MEMORY: SMALLER PAGETABLES

Shivaram Venkataraman

CS 537, Spring 2020

ADMINISTRIVIA

- Project 2a is due Friday
- Project 1b grades this week

- Discussion today: xv6 scheduler walk through for P2b

OFFICE HOURS

1. One question per student at a time
2. Please be prepared before asking questions
3. The TAs might not be able to fix your problem
4. Limited time per student

Search Piazza?

AGENDA / LEARNING OUTCOMES

Memory virtualization

- How we reduce the size of page tables?
- What can we do to handle large address spaces?

RECAP

PAGE TRANSLATION WITH TLB

For each mem reference:

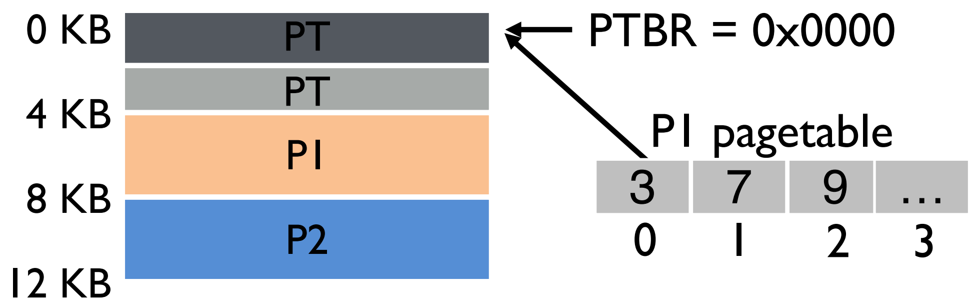
1. extract **VPN** (virt page num) from **VA** (virt addr)
2. check TLB for **VPN**

   if miss:

   3. calculate addr of **PTE** (page table entry)
   4. read **PTE** from memory, add to TLB

5. extract **PFN** from TLB (page frame num)
6. build **PA** (phys addr)
7. read contents of **PA** from memory

TLB ACCESSES: EXAMPLE



Virt	Phys
load 0x0000	
load 0x0004	
…	
…	
load 0x2000	

1. extract **VPN**
2. check TLB for **VPN**

if miss:

3. calculate **PTE** addr
4. read **PTE from mem**, add TLB

5. extract **PPN** from TLB
6. build **PA** (phys addr)
7. read **PA** from memory

CPU's TLB

Valid	VPN	PPN

TLB SUMMARY

TLB performance depends strongly on workload

- Sequential workloads perform well
- Workloads with temporal locality can perform well

TLBs increase cost of context switches

- Flush TLB on every context switch
- Add ASID to every TLB entry

In different systems, hardware or OS handles TLB misses

DISADVANTAGES OF PAGING

Additional memory reference to page table → Very inefficient

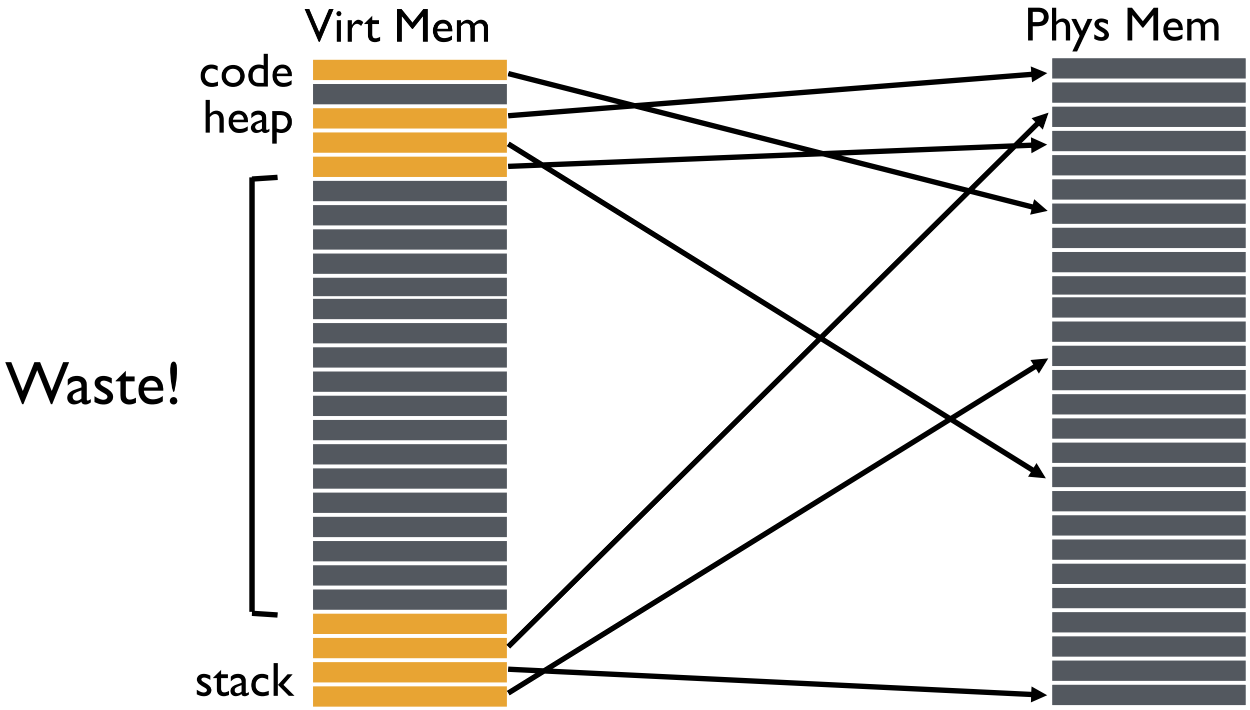
- Page table must be stored in memory
- MMU stores only base address of page table

Storage for page tables may be substantial

- Simple page table: Requires PTE for all pages in address space
  Entry needed even if page not allocated ?

SMALLER PAGE TABLES

WHY ARE PAGE TABLES SO LARGE?



AVOID SIMPLE LINEAR PAGE TABLES?

Use more complex page tables, instead of just big array

Any data structure is possible with software-managed TLB

- Hardware looks for vpn in TLB on every memory access
- If TLB does not contain vpn, TLB miss
  - Trap into OS and let OS find vpn → ppn translation
  - OS notifies TLB of vpn → ppn for future accesses

What about hardware managed TLBs?

OTHER APPROACHES

1. Segmented Pagetables
2. Multi-level Pagetables
   - Page the page tables
   - Page the pagetables of page tables…
3. Inverted Pagetables

VALID PTES ARE CONTIGUOUS

PFN	valid	prot
10	1	r-x
-	0	-
23	1	rw-
-	0	-
-	0	-
-	0	-
-	0	-
…many more invalid…		
-	0	-
-	0	-
-	0	-
-	0	-
28	1	rw-
4	1	rw-

how to avoid storing these?

Note "hole" in addr space: valids vs. invalids are clustered

How did OS avoid allocating holes in phys memory?

Segmentation

COMBINE PAGING AND SEGMENTATION

Divide address space into segments (code, heap, stack)

- Segments can be variable length

Divide each segment into fixed-sized pages.

Logical address divided into three portions

seg # (4 bits)	page number (8 bits)	page offset (12 bits)

Implementation

- Each segment has a page table
- Track base (physical address) and bounds of the **page table** per segment

EXAMPLE: PAGING AND SEGMENTATION

seg # (4 bits)	page number (8 bits)	page offset (12 bits)

seg	base	bounds	R W
0	0x002000	0xff	1 0
1	0x000000	0x00	0 0
2	0x001000	0x0f	1 1

```
0x002070 read:
0x202016 read:
0x104c84 read:
0x010424 write:
0x210014 write:
0x203568 read:
```

...	0x001000
0x01f	
0x011	
0x003	
0x02a	
0x013	
...	
0x00c	0x002000
0x007	
0x004	
0x00b	
0x006	
...	

ADVANTAGES OF PAGING AND SEGMENTATION

Advantages from using Segments

- Decreases size of page tables. If segment not used, no need for page table

Advantages from using Pages

- No external fragmentation
- Segments can grow without any reshuffling

Advantages of using both

- Increases flexibility of sharing
- Share either single page or entire segment

DISADVANTAGES OF PAGING AND SEGMENTATION

Potentially large page tables (for each segment)

- Must allocate page table for each segment contiguously
- More problematic with more address bits
- Page table size?
  - Assume 2 bits for segment, 18 bits for page number, 12 bits for offset

Each page table is:

= Number of entries * size of each entry

= Number of pages * 4 bytes

= 2^18 * 4 bytes = 2^20 bytes = 1 MB!

OTHER APPROACHES

1. Segmented Pagetables
2. Multi-level Pagetables
   - Page the page tables
   - Page the pagetables of page tables…
3. Inverted Pagetables

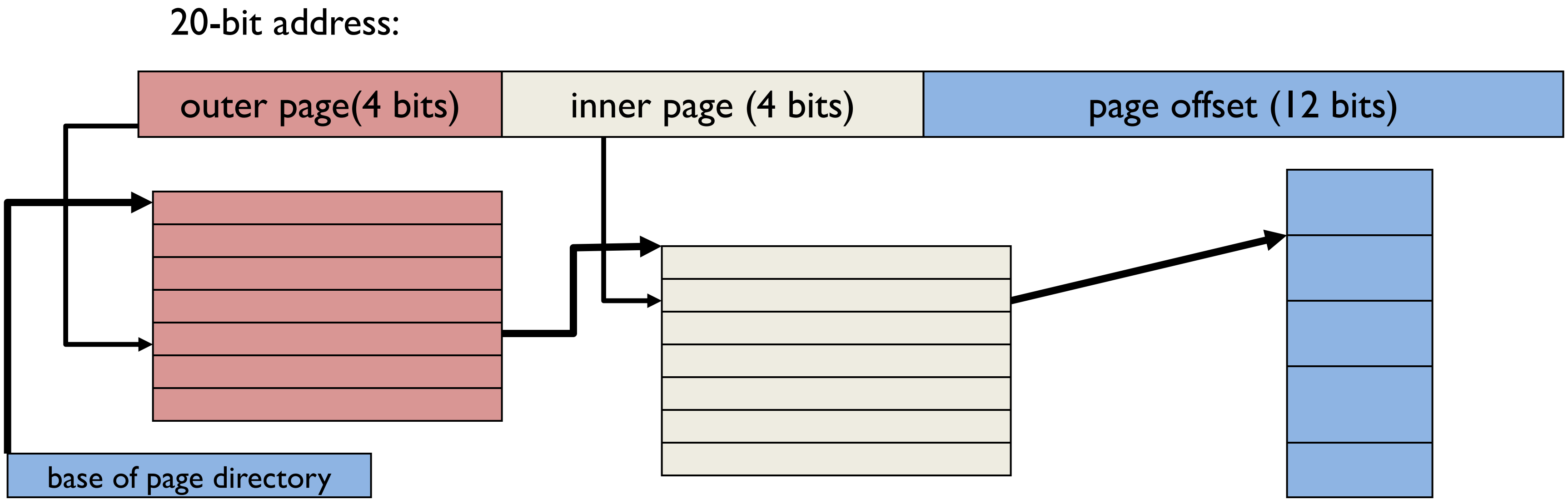
MULTILEVEL PAGE TABLES

Goal: Allow page table to be allocated non-contiguously

Idea: Page the page tables

- Creates multiple levels of page tables; outer level "page directory"
- Only allocate page tables for pages in use
- Used in x86 architectures (hardware can walk known structure)

MULTILEVEL PAGE TABLES



ADDRESS FORMAT FOR MULTILEVEL PAGING

30-bit address:

outer page	inner page	page offset (12 bits)

How should logical address be structured? How many bits for each paging level?

Goal?

- Each inner page table fits within a page
- PTE size * number PTE = page size

  Assume PTE size = 4 bytes

  Page size = 2^12 bytes = 4KB

→ # bits for selecting inner page =

Remaining bits for outer page:

- 30 – ____ – ____ = ____ bits

MULTILEVEL TRANSLATION EXAMPLE

page directory

PPN	valid
0x3	1
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
0x92	1

page of PT (@PPN:0x3)

PPN	valid
0x10	1
0x23	1
-	0
-	0
0x80	1
0x59	1
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0

page of PT (@PPN:0x92)

PPN	valid
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
0x55	1
0x45	1

translate 0x01ABC

20-bit address:

outer page(4 bits)	inner page(4 bits)	page offset (12 bits)

QUIZ 12

https://tinyurl.com/cs537-sp20-quiz12

Consider a virtual address space of 16KB with 64-byte pages.

1. How many bits will we have in our virtual address for this address space?

2. What is the total number of entries in the Linear Page Table for such an address space?

3. Consider a two-level page table now with a page directory. How many bits will be used to select the inner page assuming PTE size = 4 bytes?

QUIZ12

translate 0xFEED0

page directory

PPN	valid
0x3	1
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
0x92	1

page of PT (@PPN:0x3)

PPN	valid
0x10	1
0x23	1
-	0
-	0
0x80	1
0x59	1
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0

page of PT (@PPN:0x92)

PPN	valid
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
0x55	1
0x45	1

20-bit address:

outer page(4 bits)	inner page(4 bits)	page offset (12 bits)

PROBLEM WITH 2 LEVELS?

Problem: page directories (outer level) may not fit in a page

64-bit address:

outer page?	inner page (10 bits)	page offset (12 bits)

Solution:

- Split page directories into pieces
- Use another page dir to refer to the page dir pieces.

← VPN →

PD idx 0	PD idx 1	PT idx	OFFSET

How large is virtual address space with 4 KB pages, 4 byte PTEs,
(each page table fits in page)

4KB / 4 bytes → 1K entries per level

1 level:

2 levels:

3 levels:

FULL SYSTEM WITH TLBS

On TLB miss: lookups with more levels more expensive

Assume 3-level page table
Assume 256-byte pages
Assume 16-bit addresses
Assume ASID of current process is 211

ASID	VPN	PFN	Valid
211	0xbb	0x91	1
211	0xff	0x23	1
122	0x05	0x91	1
211	0x05	0x12	0

How many physical accesses for each instruction? (Ignore ops changing TLB)

(a) 0xAA10: movl 0x1111, %edi

(b) 0xBB13: addl $0x3, %edi

(c) 0x0519: movl %edi, 0xFF10

INVERTED PAGE TABLE

Only store entries for virtual pages w/ valid physical mappings

Naïve approach:

- Search through data structure <ppn, vpn+asid> to find match
- Too much time to search entire table

Better:

- Find possible matches entries by hashing vpn+asid
- Smaller number of entries to search for exact match

Managing inverted page table requires software-controlled TLB

SUMMARY: BETTER PAGE TABLES

Problem: Simple linear page tables require too much contiguous memory

Many options for efficiently organizing page tables

If OS traps on TLB miss, OS can use any data structure

- Inverted page tables (hashing)

If Hardware handles TLB miss, page tables must follow specific format

- Multi-level page tables used in x86 architecture
- Each page table fits within a page

SWAPPING

MOTIVATION

OS goal: Support processes when not enough physical memory

- Single process with very large address space
- Multiple processes with combined address spaces

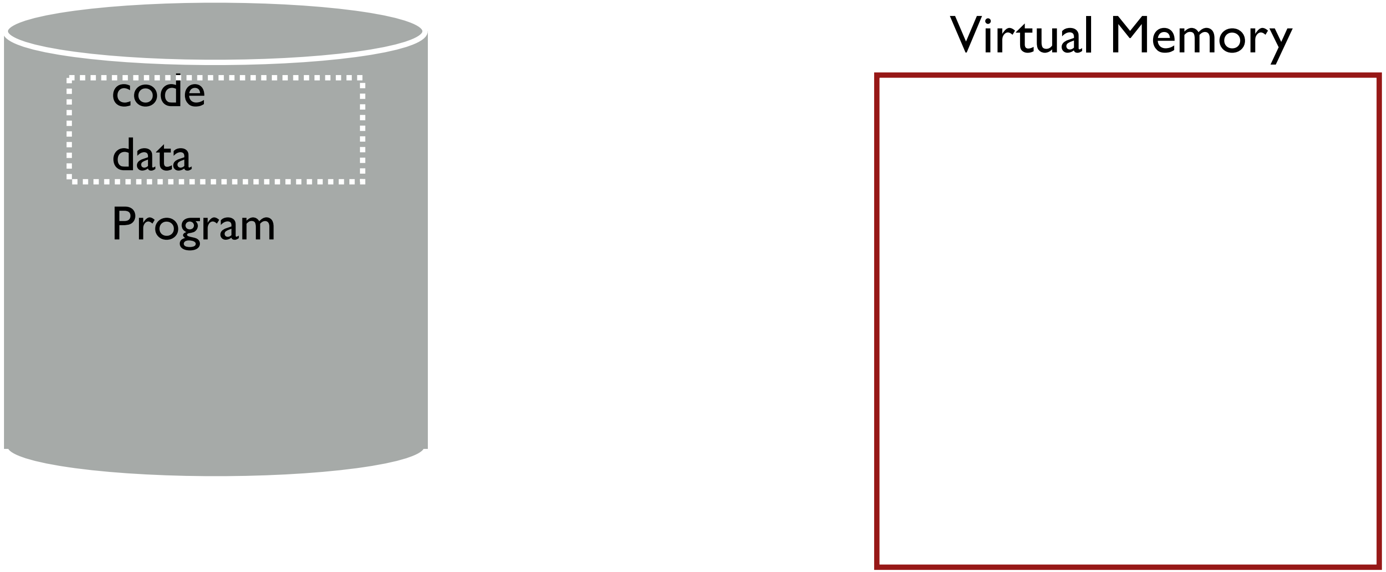
User code should be independent of amount of physical memory

- Correctness, if not performance

Virtual memory: OS provides illusion of more physical memory

Why does this work?

- Relies on key properties of user processes (workload) and machine architecture (hardware)


code
data
Program
Virtual Memory

LOCALITY OF REFERENCE

Leverage locality of reference within processes

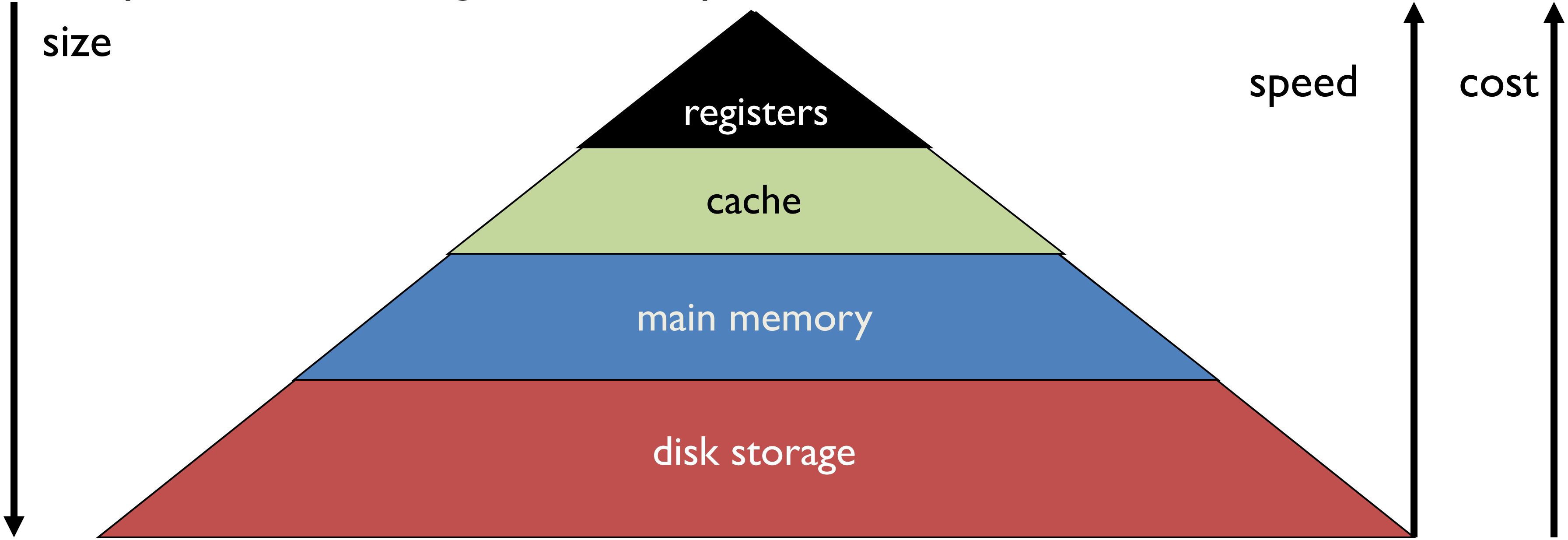
- Spatial: reference memory addresses **near** previously referenced addresses
- Temporal: reference memory addresses that have referenced in the past
- Processes spend majority of time in small portion of code
  - Estimate: 90% of time in 10% of code

Implication:

- Process only uses small amount of address space at any moment
- Only small amount of address space must be resident in physical memory

MEMORY HIERARCHY

Leverage memory hierarchy of machine architecture
Each layer acts as "backing store" for layer above



SWAPPING INTUITION

Idea: OS keeps unreferenced pages on disk

- Slower, cheaper backing store than memory

Process can run when not all pages are loaded into main memory

OS and hardware cooperate to make large disk seem like memory

- Same behavior as if all of address space in main memory

Requirements:

- OS must have **mechanism** to identify location of each page in address space → in memory or on disk
- OS must have **policy** to determine which pages live in memory and which on disk

VIRTUAL ADDRESS SPACE MECHANISMS

Each page in virtual address space maps to one of three locations:

- Physical main memory: Small, fast, expensive
- Disk (backing store): Large, slow, cheap
- Nothing (error): Free

Extend page tables with an extra bit: present

- permissions (r/w), valid, present
- Page in memory: present bit set in PTE
- Page on disk: present bit cleared
  - PTE points to block on disk
  - Causes trap into OS when page is referenced
  - Trap: page fault

Disk

Phys Memory

PFN	valid	prot	present
10	1	r-x	1
-	0	-	-
23	1	rw-	0
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
28	1	rw-	0
4	1	rw-	1

What if access vpn 0xb?

VIRTUAL MEMORY MECHANISMS

First, hardware checks TLB for virtual address

- if TLB hit, address translation is done; page in physical memory

Else ...

- Hardware or OS walk page tables
- If PTE designates page is present, then page in physical memory
  (i.e., present bit is cleared)

Else

- Trap into OS (not handled by hardware)
- OS selects victim page in memory to replace
  - Write victim page out to disk if modified (use dirty bit in PTE)
- OS reads referenced page from disk into memory
- Page table is updated, present bit is set
- Process continues execution

NEXT STEPS

Project 2a: Due Friday

Discussion section:

- Project 2b prep!
- xv6 scheduler walk through

Next class: More Swapping!