

CS 423

Operating System Design: TLBs and More Page Tables

Feb 17

Ram Kesavan

Logistics

MP0 grades released (on Canvas)

MP1 due 2/25 - Utilize TA office hours if you need help with MP1

AGENDA / LEARNING OUTCOMES

Continue discussion on TLBs - solves the performance problem with address translation

Smaller page tables: tackle memory overheads of page tables

RECAP

Disadvantages of Paging

Additional memory reference to page table entry

- Extra memory access needed for each memory access!
- Inefficient, even if page table is stored in memory
- MMU stores only base address of page table
 - Solution: TLBs

Space needed for page tables is too large

- Simple page table: requires PTE for each virtual page number
PTE needed even if page not allocated
- Page tables must be contiguously allocated
 - Solution: paging the page tables!

Disadvantages of Paging

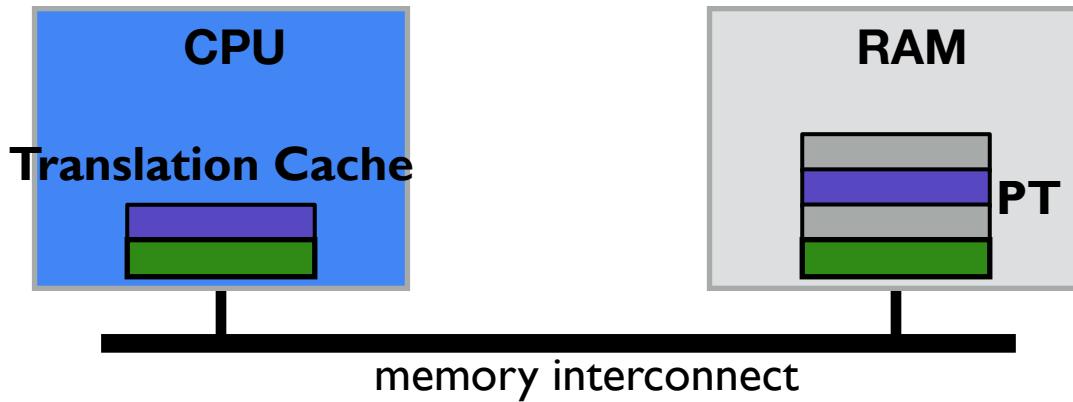
Additional memory reference to page table □ Very inefficient

- Every memory reference results in two references
- One to PT and one to actual address
- How to solve this problem? TLB

Storage for page tables may be substantial

- Simple page table: Requires PTE for all pages in address space
- Better ways to save space...

TLB: CACHE PAGE TRANSLATIONS



TLB: TRANSLATION LOOKASIDE BUFFER

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

HW AND OS ROLES

Who handles TLB hit?

Who handles TLB miss?

TLB MISS - HW AND OS ROLES

If HW, then HW must know where the PT is in memory

CR3 in x86

Page table structure agreed upon between OS and HW

Hardware “walks” page table and fills in TLB

If OS (“software managed TLB”)

HW traps into OS upon TLB miss

OS walks page table (any data structure chosen by OS)

OS fills in TLB (TLB operations are privileged instructions)

How to replace TLB entries?

Standard policy problem in any caching solution

LRU: Evict least-recently-used TLB entry

Needs per-entry bits to track that

Another simple option: random

TLB Summary

Paging is great, but accessing page tables for every memory access is slow
Cache recent page translations in TLB

- MMU performs TLB lookup on every memory access

TLB performance depends strongly on workload

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

TLB increases cost of context switches

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

TLB miss handling: hardware or OS

With trade-offs

END RECAP

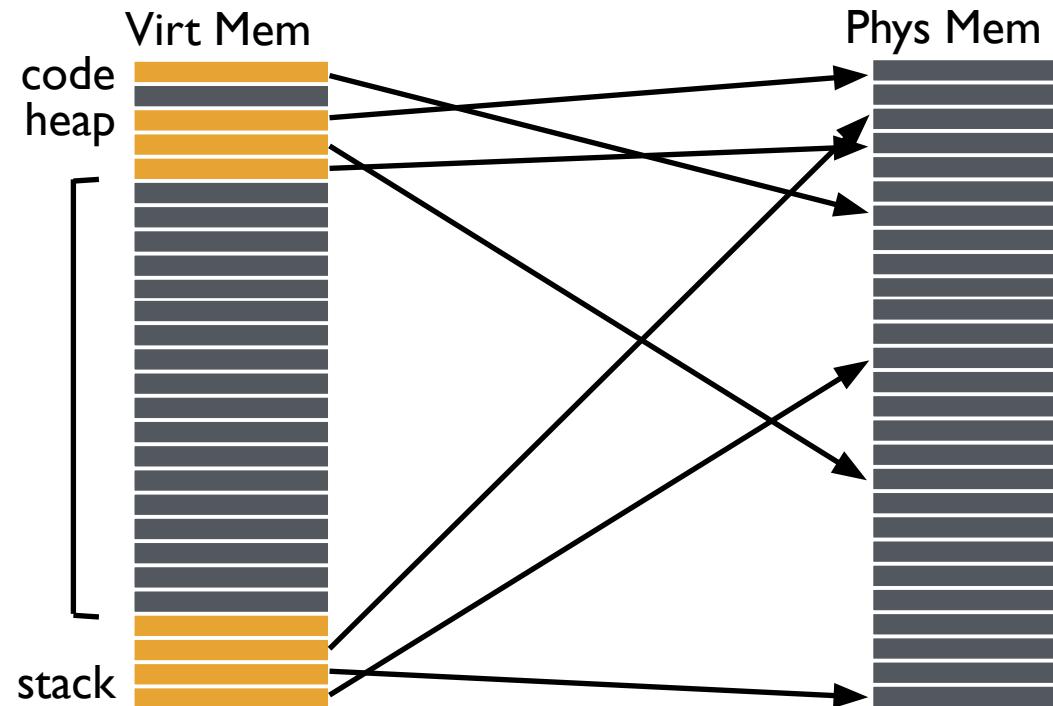
Solve the Space Problem Now

Storage for page tables may be substantial

- Simple page table: Requires PTE for all pages in address space
- Better ways to save space...

Why are Page Tables so Large?

Unallocated
Unused
But PT entries exist



MANY INVALID PTEs

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

how to avoid
storing these?

Problem: linear PT must still allocate PTE for
each page (even unallocated ones)

AVOID SIMPLE LINEAR PAGE TABLES?

Note: page table is just a data structure—doesn't have to be a flat table!

Use more complex page tables, instead of just big array

Any data structure is possible if software handles TLB miss

- 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 inserts vpn -> ppn into TLB

OTHER APPROACHES

1. Segmented paging
2. Multi-level page tables
 - o Page the page tables
 - o Page the page tables of page tables...
3. Inverted page tables

VALID PTES ARE CONTIGUOUS

PFN	valid	prot
10	0	r-x
-	0	-
23	0	rw-
-	0	-
-	0	-
-	0	-
-	0	-
...many more invalid...		
-	0	-
-	0	-
-	0	-
-	0	-
28	0	rw-
4	0	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?

Combination: Segmented Paging

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)
----------------	----------------------	-----------------------

- Each segment has a page table
- Track base physical address and bounds of the **page table** per segment
 - This is different from original segmentation
 - Bounds => # PTEs in segment

Segmented Paging

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

Bounds = #PTEs

0x002070 read:

0x202016 read:

0x104c84 read:

0x010424 write:

0x210014 write:

0x203568 read:

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

Advantages & Disadvantages

Advantages:

- Reduces external fragmentation (unlike Segmentation)
- Segments can grow without the need to reshuffle

Disadvantages:

- Must allocate page table for each segment *contiguously*
- Page table size?
 - Assume 2 bits for segment, 18 bits for page number, 12 bits for offset

Each page table can grow to:

$$\begin{aligned} &= \text{Number of entries} * \text{size of each entry} \\ &= \text{Number of pages} * 4 \text{ bytes} \\ &= 2^{18} * 4 \text{ bytes} = 2^{20} \text{ bytes} = 1 \text{ MB!} \end{aligned}$$

OTHER APPROACHES

1. Segmented paging
2. Multi-level page tables
 - o Page the page tables
 - o Page the page tables of page tables...
3. Inverted page tables

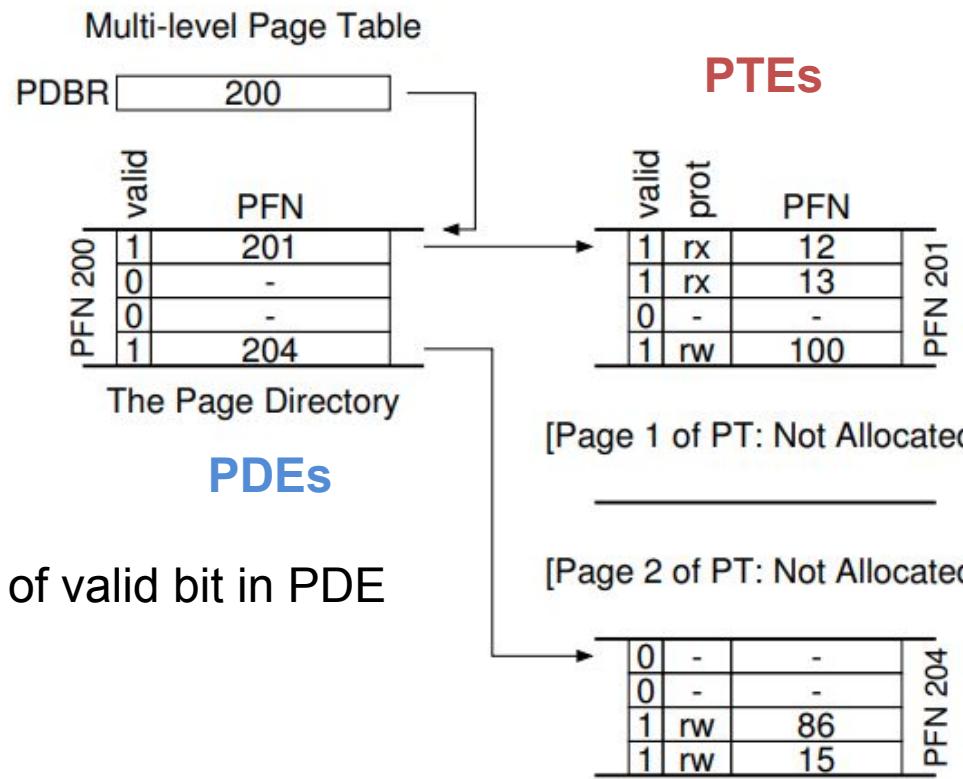
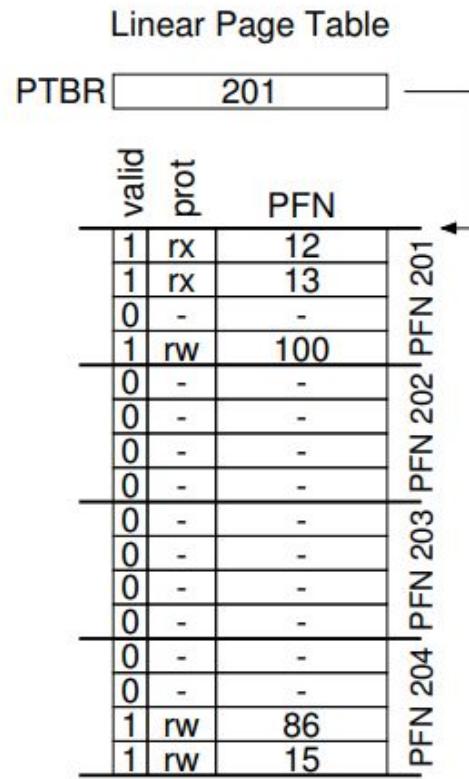
Multi-level Page Tables

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 Table – Key Idea



Meaning of valid bit in PDE
and PTE

Multilevel Page Tables

Chat for 1 minute with your neighbors...

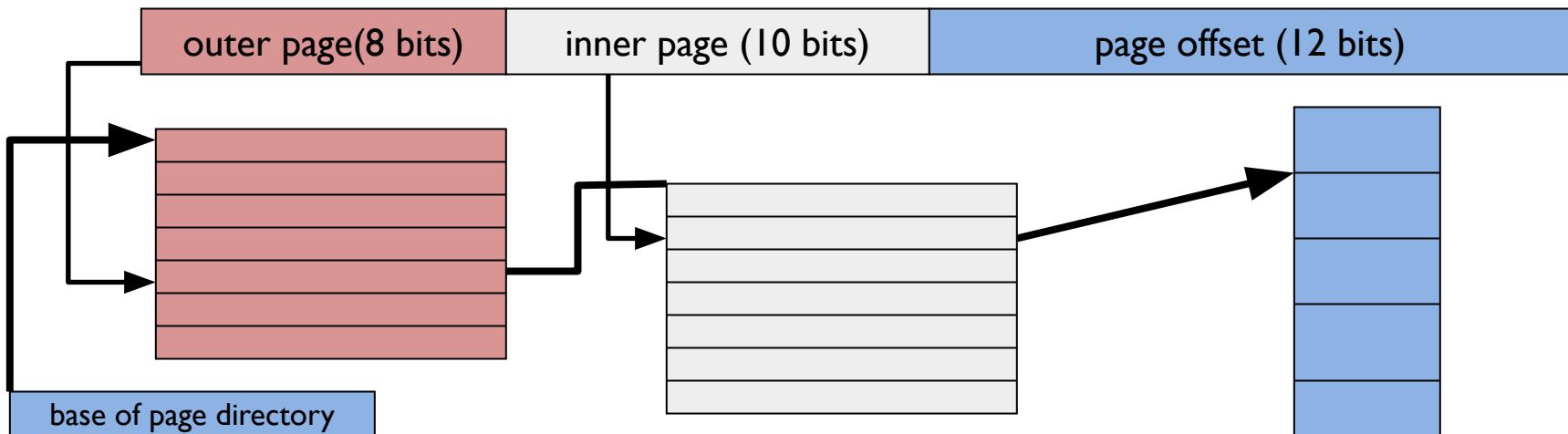
Can pages of the PT be dynamically relocated to a different physical location?

Can the PD be dynamically relocated to a different physical location?

How many memory accesses on a TLB miss with this 2-level PT?

Multilevel Page Tables

30-bit address:



Address format for multilevel Paging

30-bit address:



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

Each inner page can have 1024 PTEs

=> # bits for inner page = 10

Remaining bits for outer page:

- $30 - 12 - 10 = 8$ bits

Multilevel Translation EXAMPLE

page directory		page of PT (@PPN:0x3)		page of PT (@PPN:0x92)		
PPN	valid	PPN	valid	PPN	valid	
0x3	1	0x10	1	-	0	translate 0x01ABC
-	0	0x23	1	-	0	
-	0	-	0	-	0	
-	0	-	0	-	0	
-	0	0x80	1	-	0	Physical address?
-	0	0x59	1	-	0	
-	0	-	0	-	0	Look PD[0]
-	0	-	0	-	0	
-	0	-	0	-	0	Look PT[1] → arrive at
-	0	-	0	-	0	
-	0	-	0	-	0	page 0x23
-	0	-	0	-	0	
-	0	-	0	-	0	Concat ABC to arrive at
-	0	-	0	0x55	1	
-	0	-	0	0x45	1	0x23ABC
0x92	1					



Multilevel Translation EXAMPLE

page directory		page of PT (@PPN:0x3)		page of PT (@PPN:0x92)		
PPN	valid	PPN	valid	PPN	valid	
0x3	1	0x10	1	-	0	Translate
-	0	0x23	1	-	0	
-	0	-	0	-	0	VA: 0x04000
-	0	-	0	-	0	
-	0	0x80	1	-	0	
-	0	0x59	1	-	0	
-	0	-	0	-	0	
-	0	-	0	-	0	
-	0	-	0	-	0	VA: 0xFEED0
-	0	-	0	-	0	
-	0	-	0	-	0	
-	0	-	0	-	0	
-	0	-	0	-	0	
-	0	-	0	0x55	1	
-	0	-	0	0x45	1	
0x92	1					

outer page(4 bits)

inner page(4 bits)

page offset (12 bits)

20-bit
address:

PROBLEM WITH 2 LEVELS?

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

Consider 30 bit address with 512-byte pages

=> 9 bits for offset; leaves 21 bits for VPN

Remember our goal: each inner page should fit within a page

So how many PTE per page? With 512-byte pages and 4-byte PTE, we can have 128 entries → this means 7 bits for inner page, leaving 14 bits for outer page (or directory) → 2^{14} PDEs

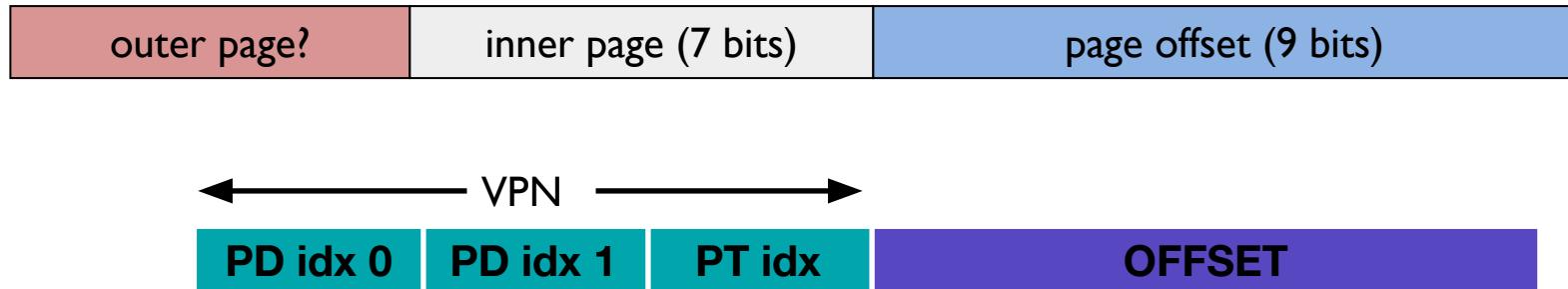
Assume 4-byte PDE, then PD itself will span _____ pages?

PD cannot be contained in one page now!

PROBLEM WITH 2 LEVELS?

Solution: page the page directory!

Add another level of page directory that points to PD pages



Can keep going recursively! Let page = 4KB (offset is 12 bits); 1K PTEs/page

2 level tree: ?

3 level tree: ?

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

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

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

INVERTED PAGE TABLE

Only store entries for virtual pages w/ valid physical mappings

Naïve approach:

Search through data structure $\langle \text{ppn}, \text{vpn+asid} \rangle$ 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

Used in IBM PowerPC

Managing inverted page table requires software-controlled TLB (although doing some of these in HW is possible)

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: Not limited by physical memory WRT #processes supported

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

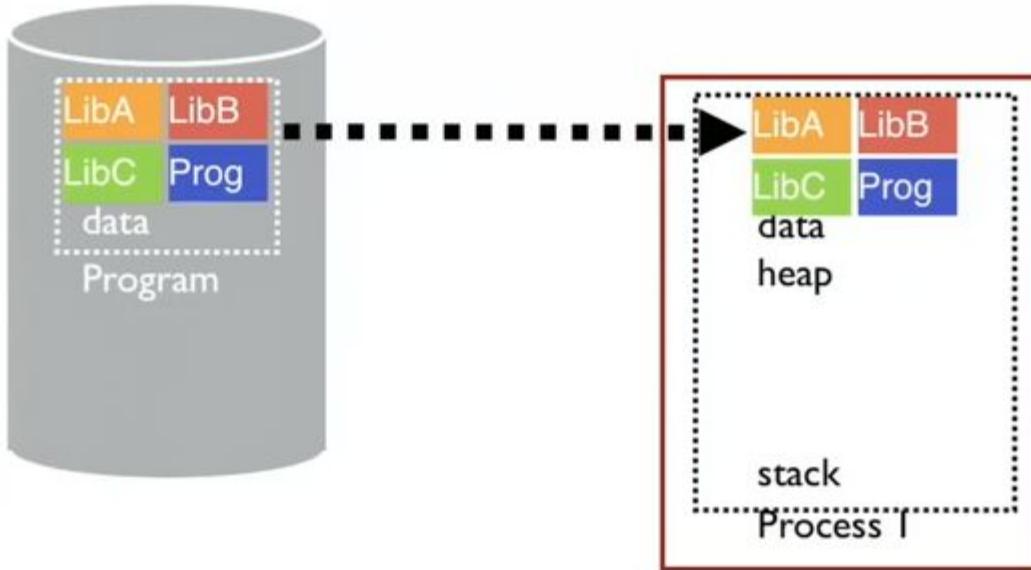
User code should be independent of amount of physical memory

- Correctness, performance

Virtual memory: OS provides illusion of more physical memory

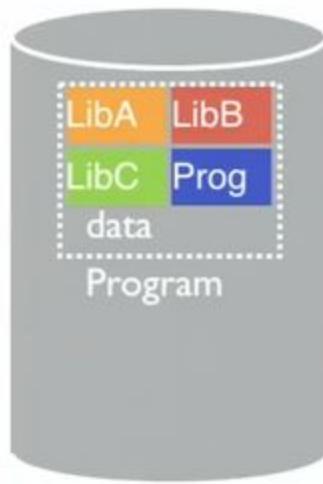
How does this work?

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



Code: many large libraries, some of which are rarely/never used

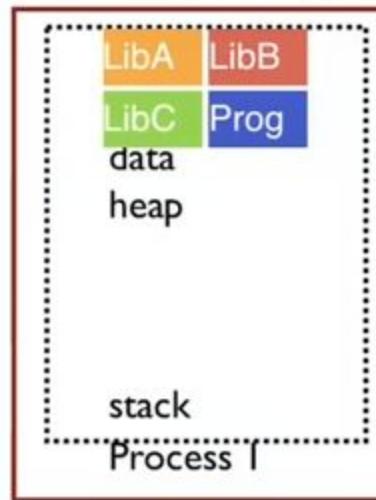
How to avoid wasting physical pages to store rarely used virtual pages?



Phys Memory



Virtual Memory



NEXT STEPS

Next class: Swapping!