

COMP SCI 3004

Operating Systems

Week 4 – TLBs, smaller tables &
Swapping

**make
history.**



THE UNIVERSITY
of ADELAIDE

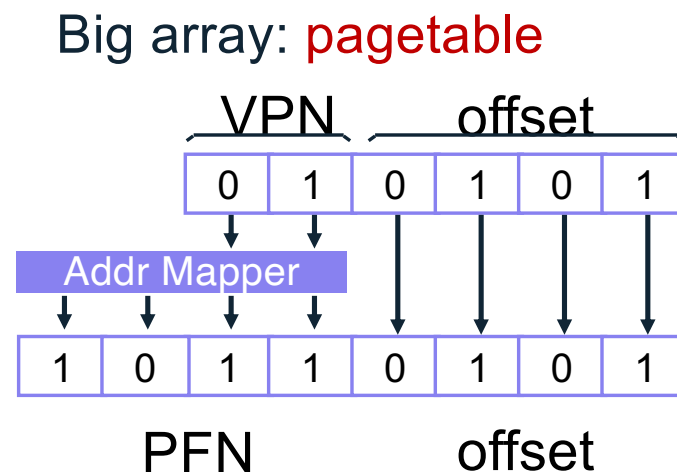
Introduction

Questions answered in this lecture:

- Review paging...
- What is the basic idea of a TLB (Translation Lookaside Buffer)?
- What types of workloads perform well with TLBs? What about context-switches?
- How to create smaller tables?
- Swapping Mechanisms
- Swapping Policies

Last week: Page Tables

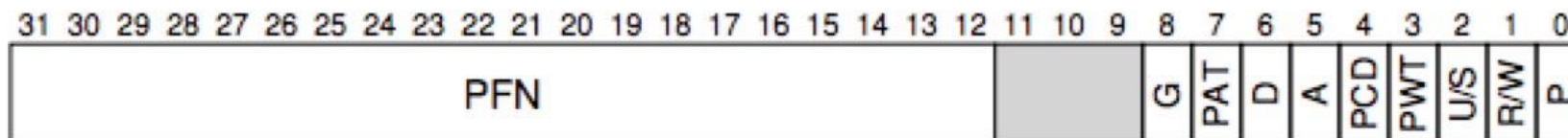
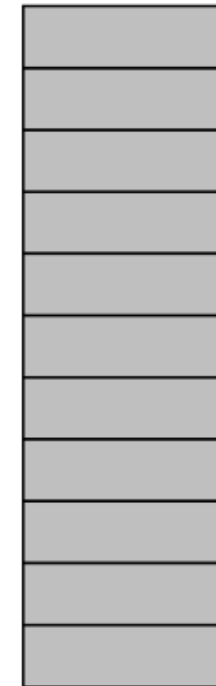
Number of bits in virtual address format does not need to equal number of bits in physical address format



VPN

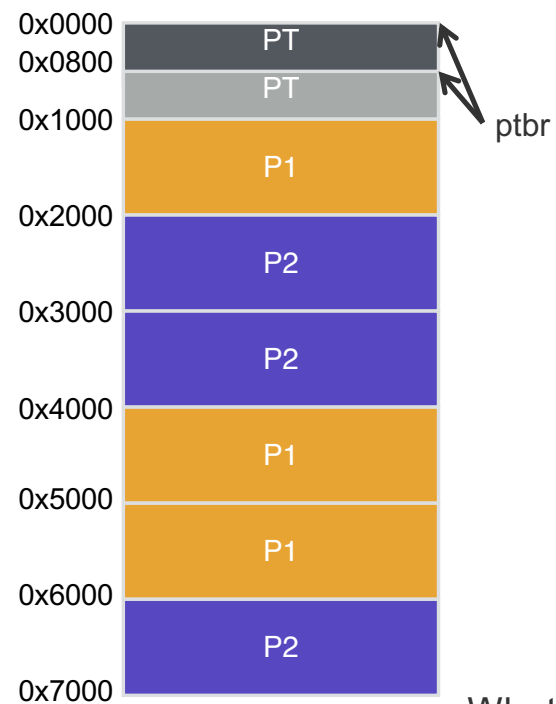
0

2^n



Review: Paging

Assume 4 KB pages



1	5	4	...	P1 pagetable
6	2	3	...	P2 pagetable

Virtual	Physical
load 0x0000	load 0x0800
	load 0x6000
load 0x1444	load 0x0808
	load 0x2444
load 0x1444	load 0x0008
	load 0x5444

What do we need to know?

Location of page table in memory (ptbr)

Size of each page table entry (assume 8 bytes)

Review: Paging PROS and CONS

- **Advantages**

- No external fragmentation – Don't need to find contiguous RAM
- All free pages are equivalent – Easy to manage, allocate, and free pages.

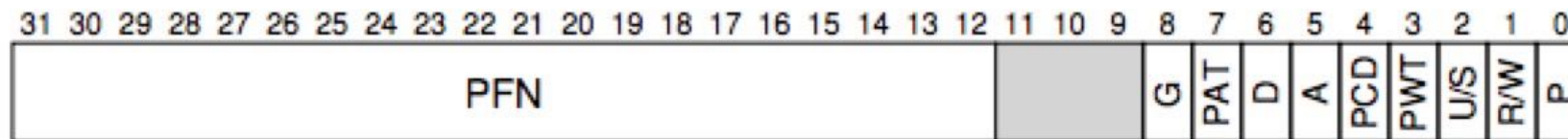
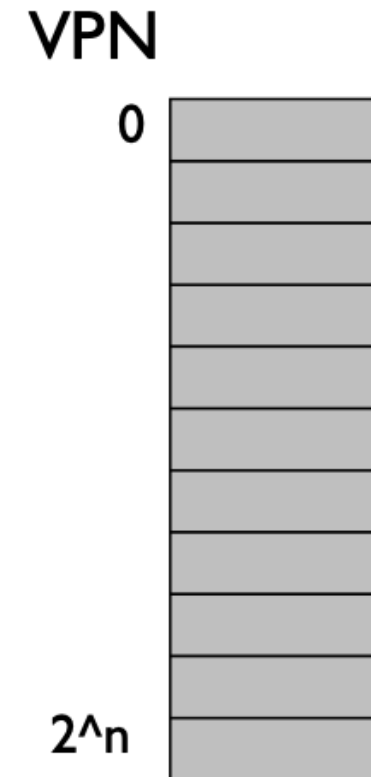
- **Disadvantages**

- Page tables are too big – Must have one entry for every page of address space
- Accessing page tables is too slow – Doubles number of memory references per instruction

Translation Steps

For each mem reference:

- (cheap) 1. extract VPN (virt page num) from VA (virt addr)
- (cheap) 2. calculate addr of PTE (page table entry)
- (expensive) 3. read PTE from memory
- (cheap) 4. extract PFN (page frame num)
- (cheap) 5. build PA (phys addr)
- (expensive) 6. read contents of PA from memory into register



Example: Array Iterator

```
int sum = 0;
for (i=0; i<N; i++){
    sum += a[i];
}
```

Assume 'a' starts at 0x1000

Ignore instruction fetches

What virtual addresses?

load 0x1000

load 0x1004

load 0x1008

load 0x100C

...

What physical addresses?

load 0x8004

load 0x5000

load 0x8004

load 0x5004

load 0x8004

load 0x5008

load 0x8004

load 0x500C

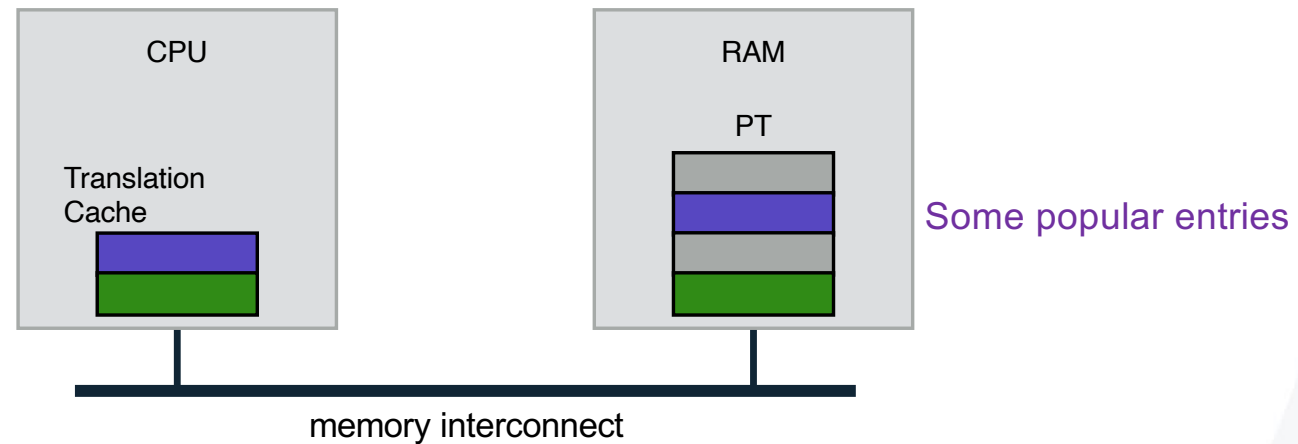
Observation:

Repeatedly access same PTE because program repeatedly accesses same virtual page

Aside: What can you infer?

- ptbr: 0x8000; PTE 4 bytes each
- VPN 1 -> PFN 5

Strategy: Cache Page Translations



TLB: Translation Lookaside Buffer

Array Iterator (w/ TLB)

```
int sum = 0;  
for (i = 0; i < 2048; i++) {  
    sum += a[i];  
}
```

Assume following virtual address stream:

load 0x1000

load 0x1004

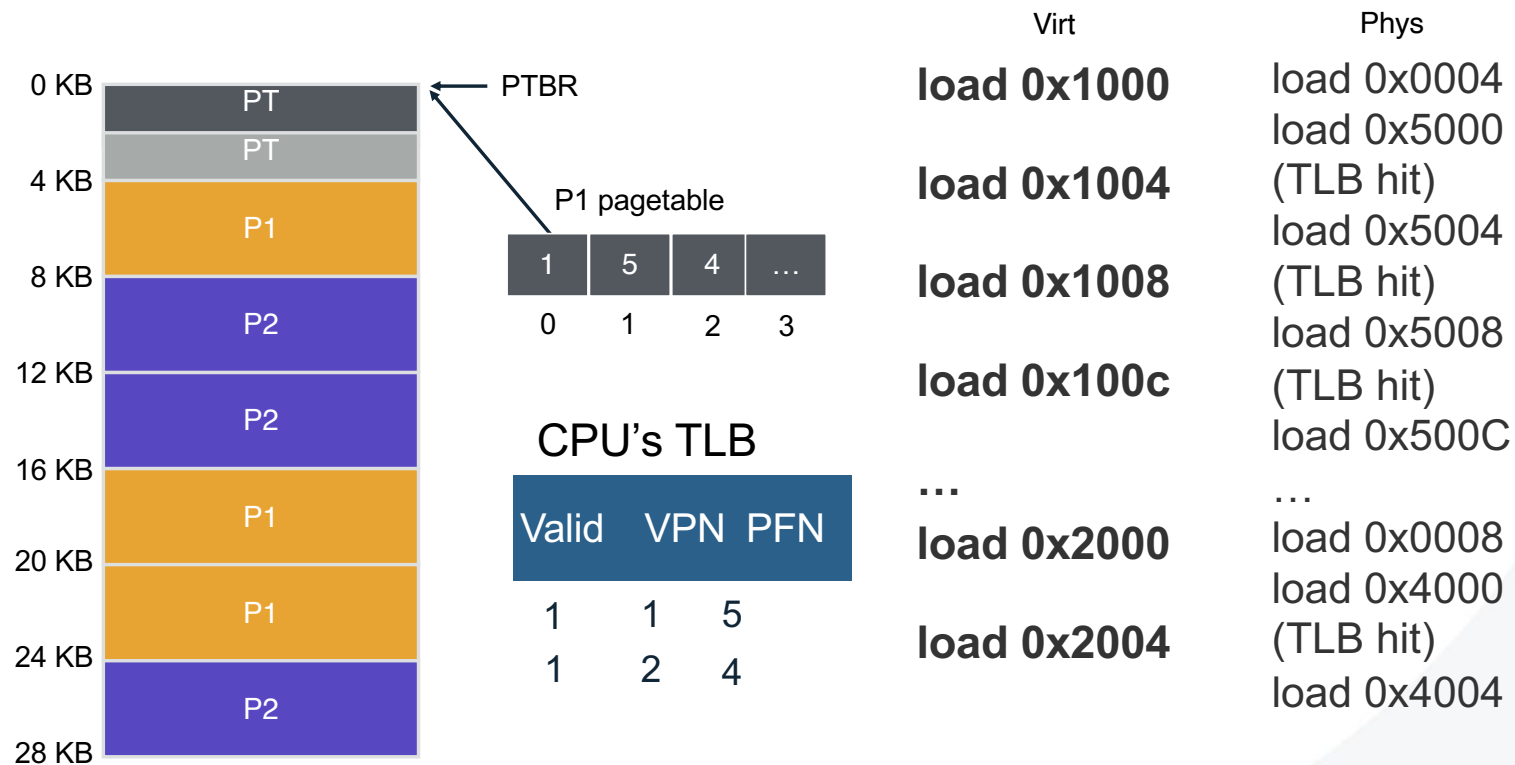
load 0x1008

load 0x100C

...

What will TLB behavior look like?

TLB Accesses: SEQUENTIAL Example



Performance of TLB?

```
int sum = 0;
for (i=0; i<2048; i++) {
    sum += a[i];
}
```

Calculate miss rate of TLB for data:

TLB misses / # TLB lookups

TLB lookups?

= number of accesses to a = 2048

TLB misses?

= number of unique pages accessed

= 2048 / (elements of 'a' per 4K page)

= 2K / (4K / sizeof(int)) = 2K / 1K

= 2

Miss rate?

$2/2048 = 0.1\%$

Hit rate? (1 – miss rate)

99.9%

Would hit rate get better or worse with smaller pages?

Worse

TLB Performance

- How can system improve TLB performance (hit rate) given fixed number of TLB entries?
- Increase page size
 - Fewer unique page translations needed to access same amount of memory
- TLB Reach:
 - Number of TLB entries * Page Size

TLB Performance with Workloads

- **Sequential array accesses almost always hit in TLB**
 - Very fast!
- **What access pattern will be slow?**
 - Highly random, with no repeat accesses

Workload access patterns

Workload A

```
int sum = 0;
for (i=0; i<2048; i++) {
    sum += a[i];
}
```

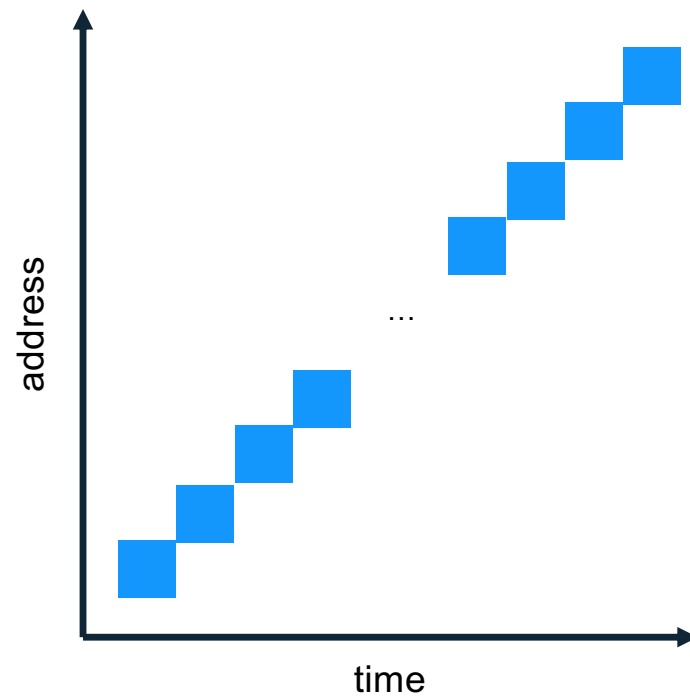
Workload B

```
int sum = 0;
srand(1234);
for (i=0; i<1000; i++) {
    sum += a[rand() % N];
}
srand(1234);
for (i=0; i<1000; i++) {
    sum += a[rand() % N];
}
```

Workload Access Patterns

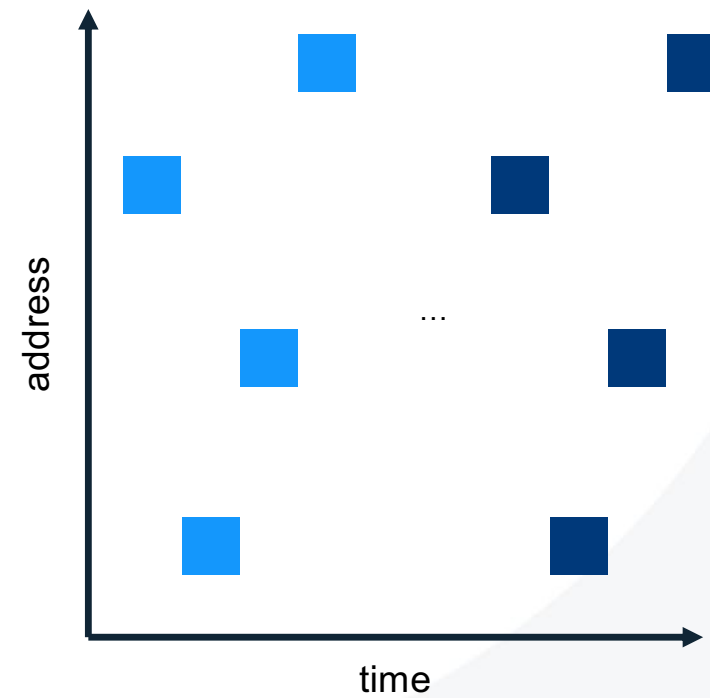
Spatial Locality

Sequential Accesses



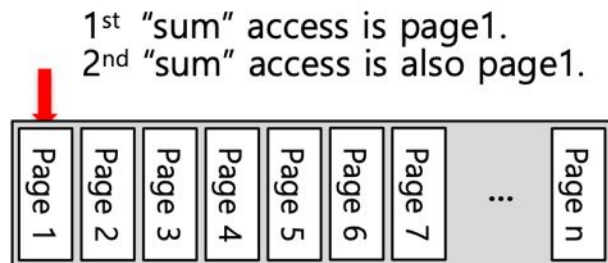
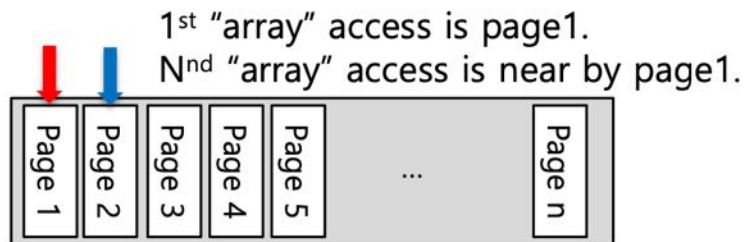
Temporal Locality

Repeated Random Accesses



Workload Locality

- What TLB characteristics are best?



- **Spatial:**
Access same page repeatedly; need same
vpn->pfn translation
Same TLB entry re-used
- **Temporal:**
Access same address near in future
Same TLB entry re-used in near future
How near in future? How many TLB entries
are there?

TLB Replacement policies

- **LRU:**
 - evict Least-Recently Used TLB slot when needed
- **Random: Evict randomly chosen entry**
 - Which is better?



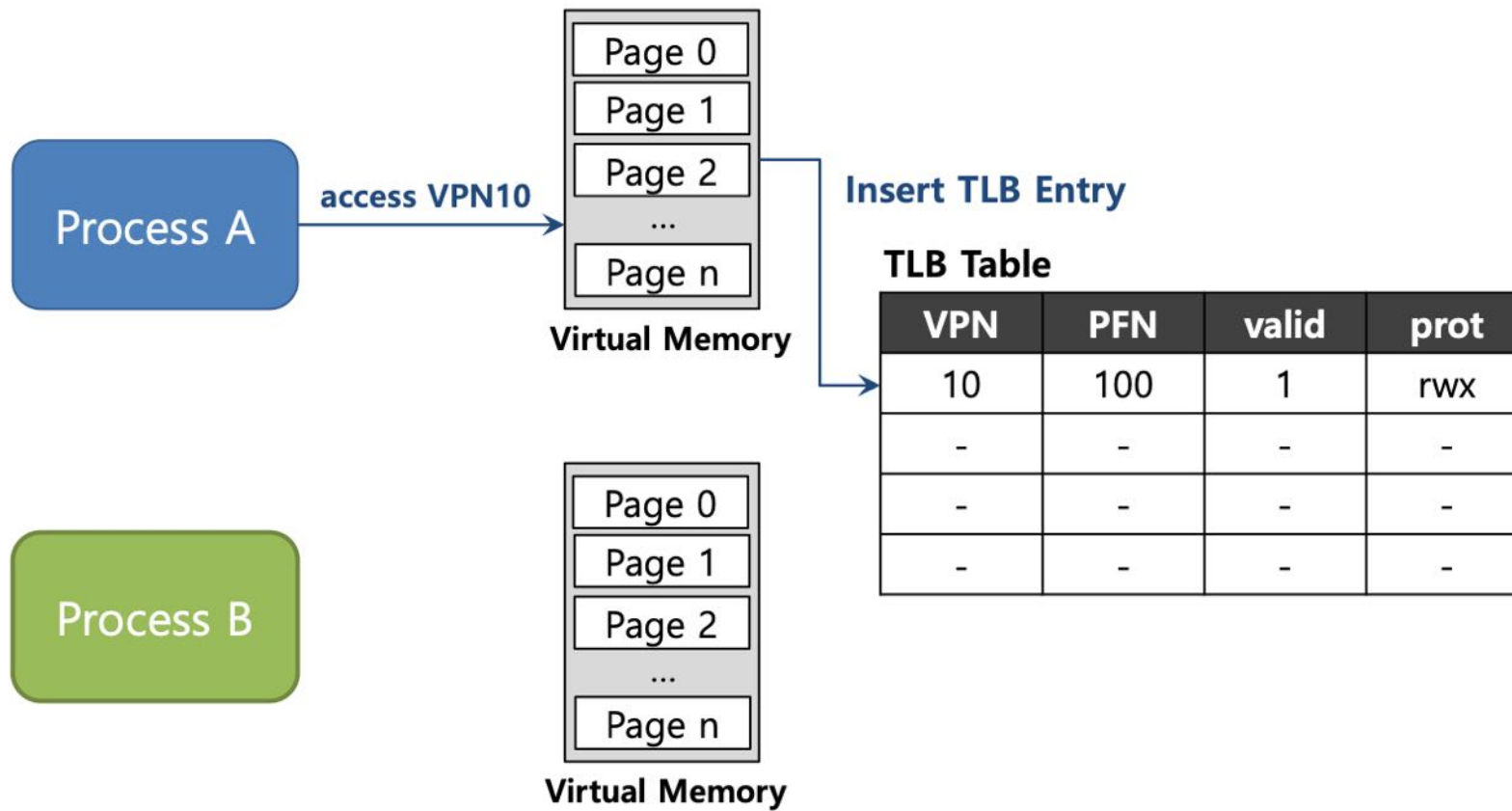
TLB Performance

- How can system improve TLB performance (hit rate) given fixed number of TLB entries?
- Increase page size
 - Fewer unique translations needed to access same amount of memory

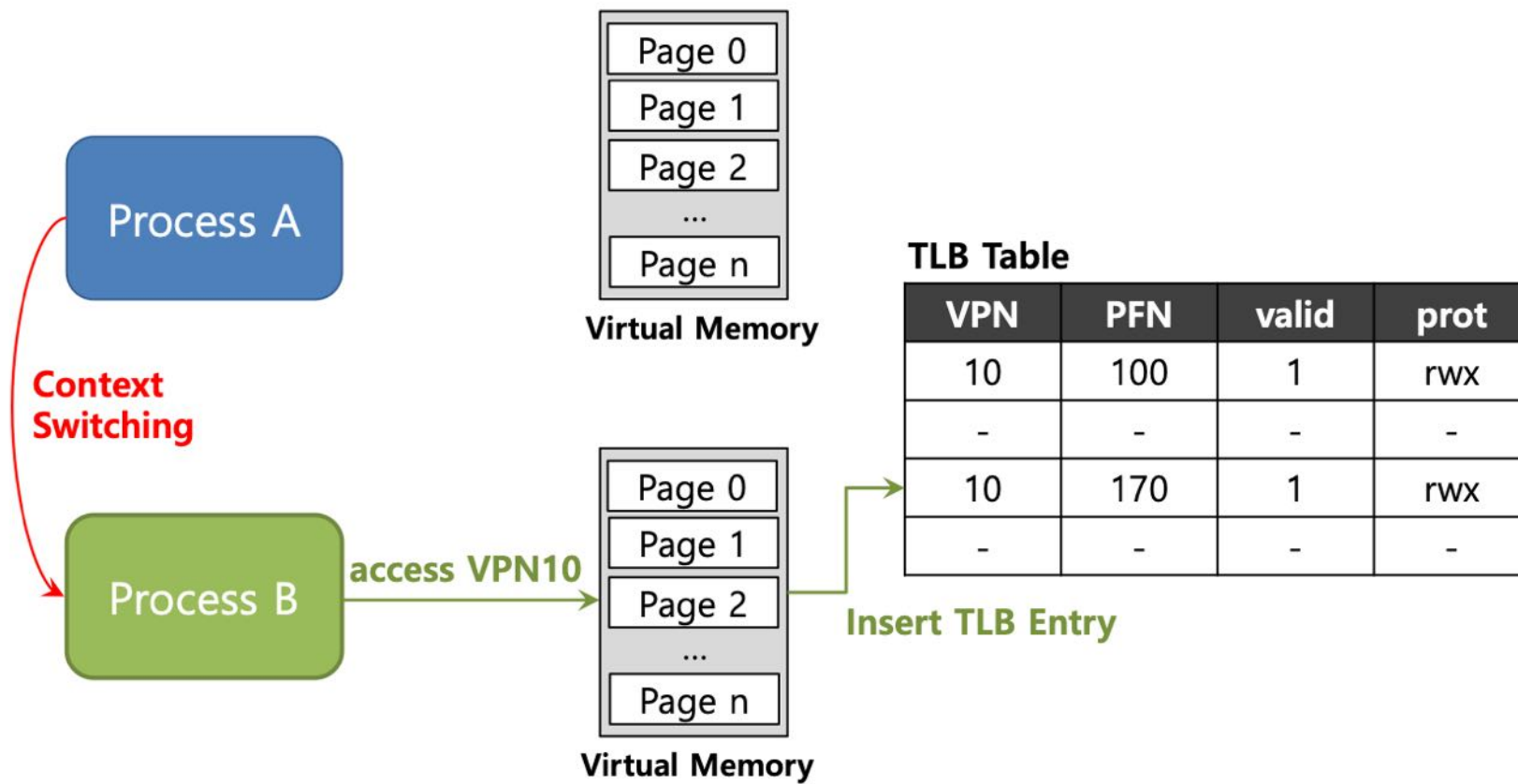
Context Switches

- What happens if a process uses cached TLB entries from another process?
- Solutions?
 1. Flush TLB on each switch
 - **Costly** → lose all recently cached translations
 2. Track which entries are for which process
 - Address Space Identifier
 - Tag each TLB entry with an 8-bit ASID

Context Switches

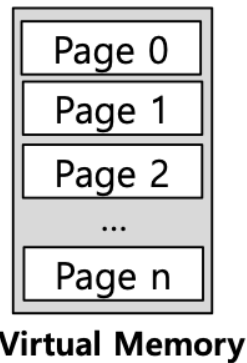


Context Switches

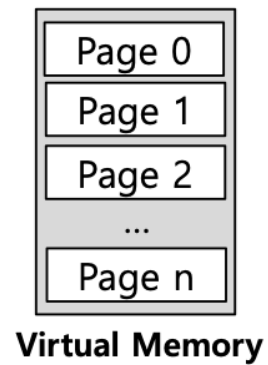


Context Switches

Process A



Process B

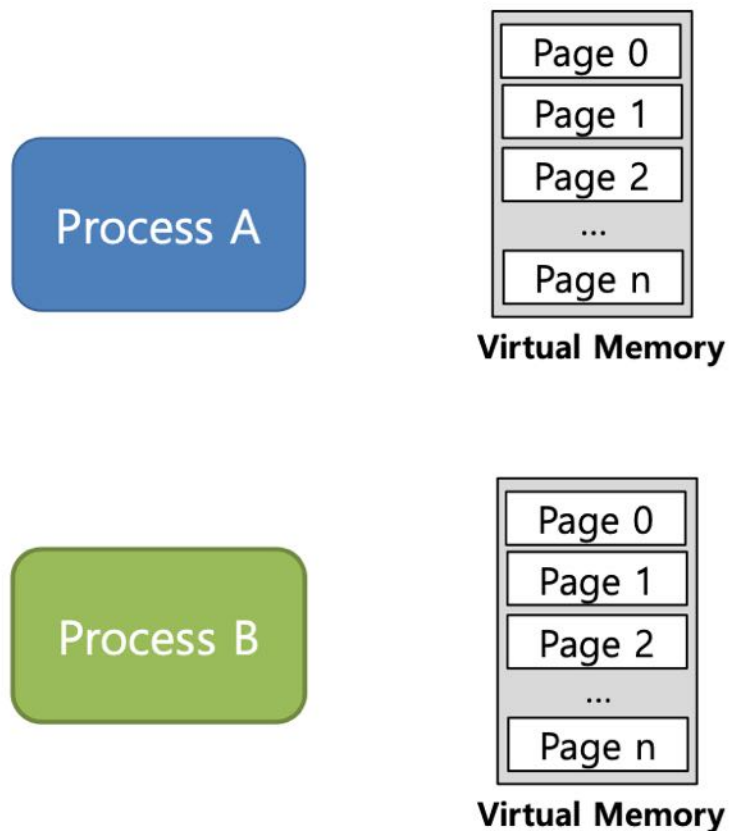


TLB Table

VPN	PFN	valid	prot
10	100	1	rwX
-	-	-	-
10	170	1	rwX
-	-	-	-

Can't **Distinguish** which entry is meant for which process

Address space identifier (ASID) in the TLB



TLB Table

VPN	PFN	valid	prot	ASID
10	100	1	rwX	1
-	-	-	-	-
10	170	1	rwX	2
-	-	-	-	-

Two processes share a page

- Process 1 is sharing physical page 101 with Process2.
- P1 maps this page into the 10th page of its address space.
- P2 maps this page to the 50th page of its address space.

VPN	PFN	valid	prot	ASID
10	101	1	rwX	1
-	-	-	-	-
50	101	1	rwX	2
-	-	-	-	-

Sharing of pages is **useful** as it reduces the number of physical pages in use.

TLB Performance


- **Context switches are expensive**
- **Even with ASID, other processes “pollute” TLB**
 - Discard process A’s TLB entries for process B’s entries
- **Architectures can have multiple TLBs**
 - 1 TLB for data, 1 TLB for instructions
 - 1 TLB for regular pages, 1 TLB for “super pages”

HW and OS Roles

- **Who Handles TLB MISS? H/W or OS?**
 - H/W: CPU must know where pagetables are
 - OS: CPU traps into OS upon TLB miss
- **Need same protection bits in TLB as pagetable**
 - rwx

Summary

- **Pages are great, but accessing page tables for every memory access is slow**
- **Cache recent page translations: TLB**
 - Hardware performs TLB lookup on every memory access
- **TLB performance depends strongly on workload**
 - Sequential workloads perform well
 - Workloads with temporal locality can perform well
 - Increase TLB reach by increasing page size
- **In different systems, hardware or OS handles TLB misses**
- **TLBs increase cost of context switches**
 - Flush TLB on every context switch
 - Add ASID to every TLB entry



COMP SCI 3004

Operating Systems

Smaller Page Tables

**make
history.**



THE UNIVERSITY
of ADELAIDE

Disadvantages of Paging

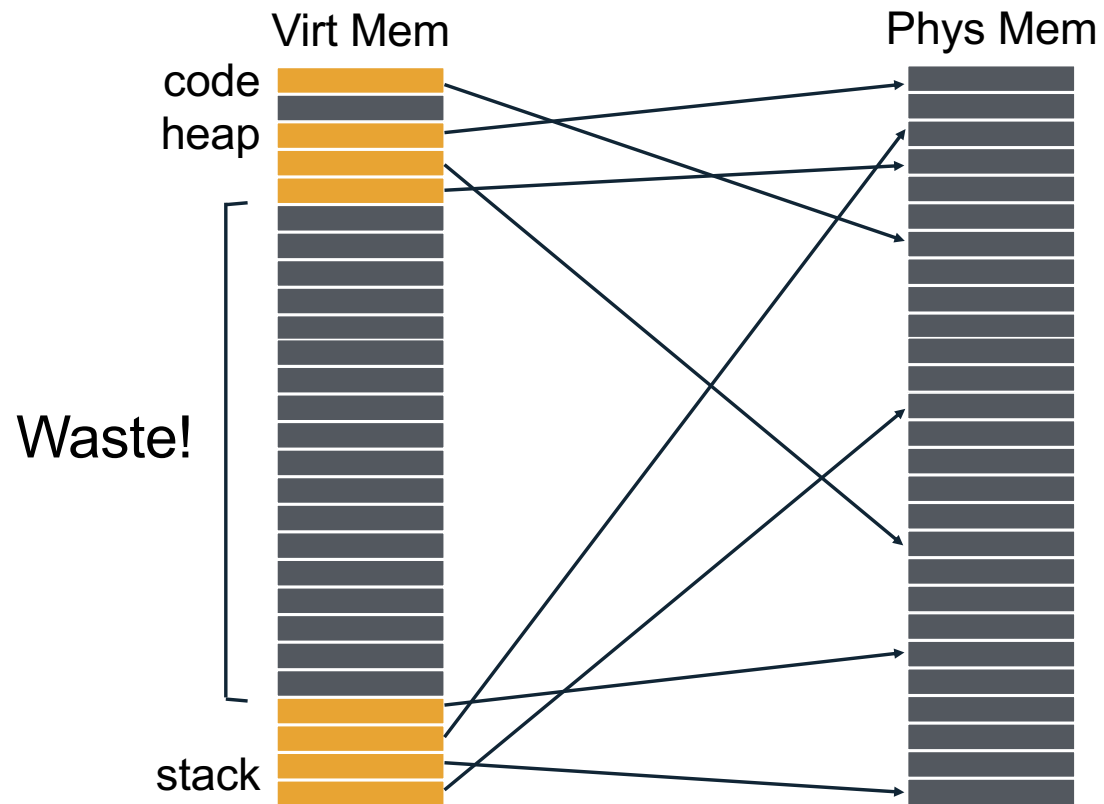
1. Additional memory reference to look up in page table

- Very inefficient
- Page table must be stored in memory
- MMU stores only base address of page table
- Avoid extra memory reference for lookup with TLBs

2. 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
- Problematic with dynamic stack and heap within address space

Why are page tables so large?



Many invalid PT entries

- Format of linear page tables:

how to avoid
storing these?

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

Other Approaches

1. Inverted Pagetables

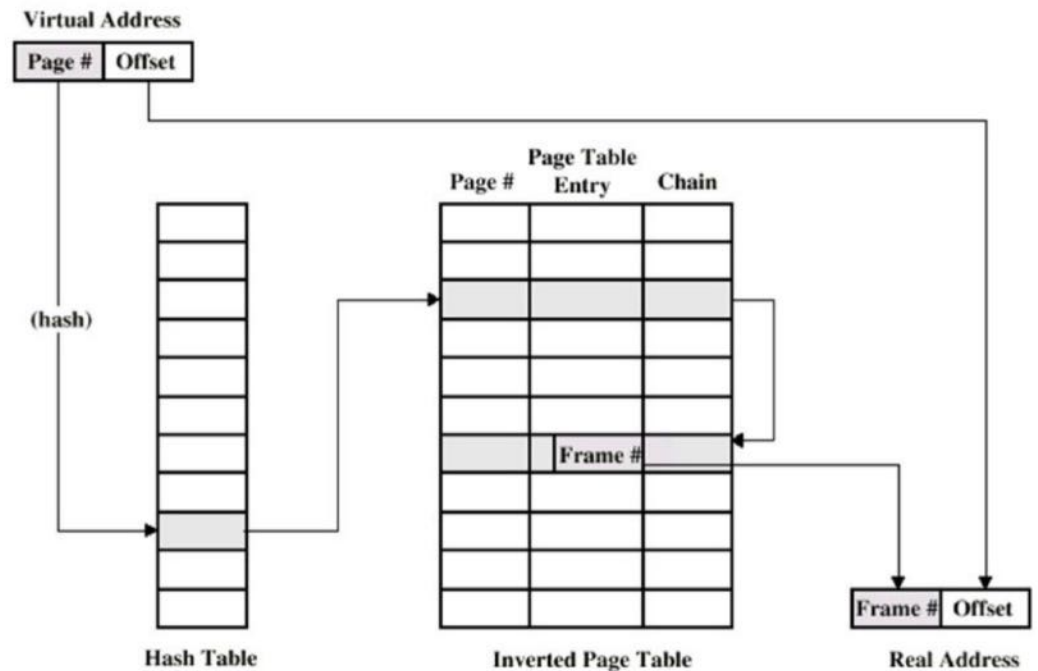
2. Segmented Pagetables

3. Multi-level Pagetables

- Page the page tables
- Page the pagetables of page tables...

1) Inverted Page Table

- Keeping a single page table that has an entry for each physical page of the system.
- The entry tells us which process is using this page, and which virtual page of that process maps to this physical page.
- Used with a hashing table (hash anchor table) in order to allow practical implementations.



2) Valid Ptes are Contiguous

how to avoid storing these?

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-

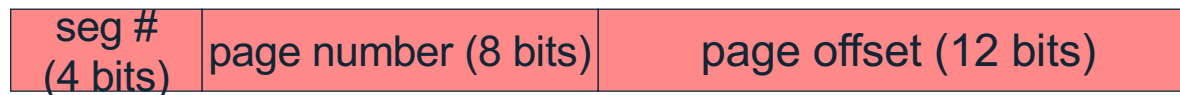
Note “hole” in addr space:
valids vs. invalids are clustered

How did OS avoid allocating holes in
phys memory?

Segmentation

2) 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**



- **Implementation – each segment**
 - has a page table
 - track base (physical address) and bounds of page table for that 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: 0x004070
 0x202016 read: 0x003016
 0x104c84 read: error
 0x010424 write: error
 0x210014 write: error
 0x203568 read: 0x02a568

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

0x001000

0x002000

Advantages of Paging and Segmentation

- **Advantages of Segments**
 - Supports sparse address spaces
 - Decreases size of page tables
 - If segment not used, not need for page table
- **Advantages of Pages**
 - No external fragmentation
 - Segments can grow without any reshuffling
 - Can run process when some pages are swapped to disk (next lecture)
- **Advantages of Both**
 - Increases flexibility of sharing
 - Share either single page or entire segment
 - How?

Disadvantages of Paging and Segmentation

- **Potentially large page tables (for each segment)**
 - Must allocate each page table 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:

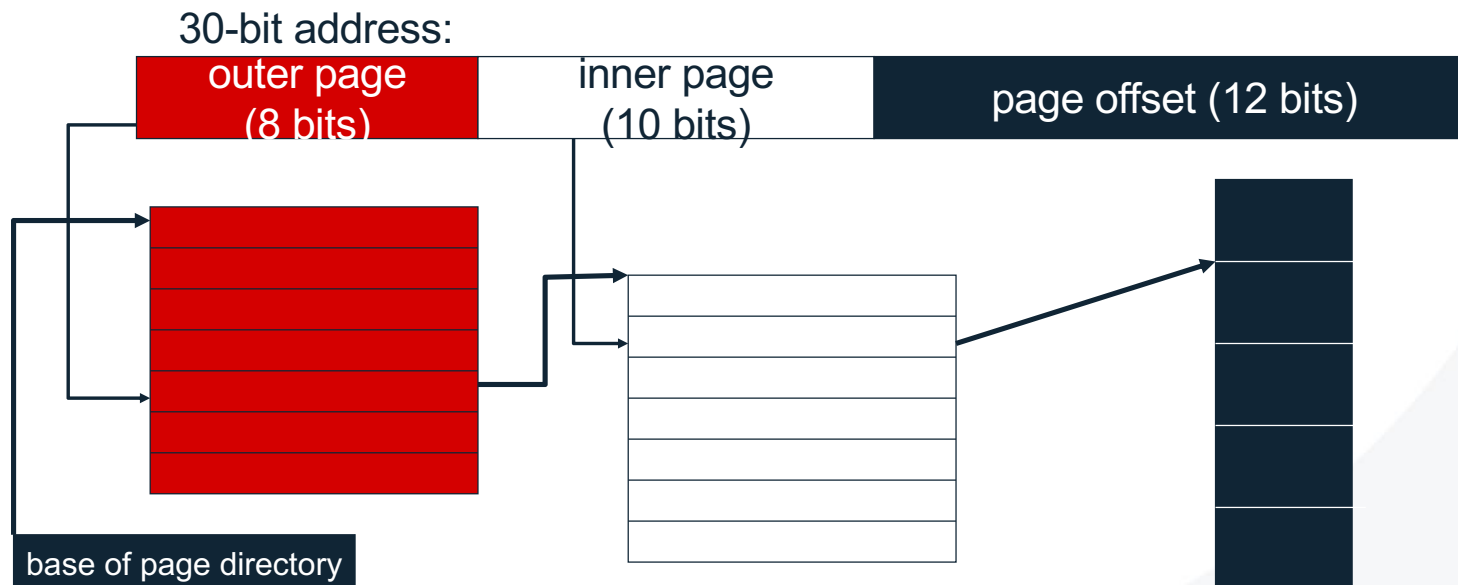
= Number of entries * size of each entry

= Number of pages * 4 bytes

= $2^{18} * 4 \text{ bytes} = 2^{20} \text{ bytes} = 1 \text{ MB!!!}$

3) Multilevel Page Tables

- Goal: Allow each page tables to be allocated non-contiguously
- Idea: Page the page tables



Example: Address format for multilevel paging

- **How should logical address be structured?**
 - How many bits for each paging level?
- **Goal**
 - Each page table fits within a page
 - PTE size * number PTE = page size
- **Remaining bits for outer page:**
 - $30 - 10 - 12 = 8$ bits

30-bit address:



Example: Address format for multilevel paging

		Page Directory		Page of PT (@PFN:100)			Page of PT (@PFN:101)		
		PFN	valid?	PFN	valid	prot	PFN	valid	prot
		100	1	10	1	r-x	—	0	—
0000 0000	code	—	0	23	1	r-x	—	0	—
0000 0001	code	—	0	—	0	—	—	0	—
0000 0010	(free)	—	0	—	0	—	—	0	—
0000 0011	(free)	—	0	—	0	—	—	0	—
0000 0100	heap	—	0	80	1	rw-	—	0	—
0000 0101	heap	—	0	59	1	rw-	—	0	—
0000 0110	(free)	—	0	—	0	—	—	0	—
0000 0111	(free)	—	0	—	0	—	—	0	—
.....	... all free ...	—	0	—	0	—	—	0	—
1111 1100	(free)	—	0	—	0	—	—	0	—
1111 1101	(free)	—	0	—	0	—	—	0	—
1111 1110	stack	—	0	—	0	—	—	0	—
1111 1111	stack	—	0	—	0	—	—	0	—
		—	0	—	0	—	—	0	—
		—	0	—	0	—	—	0	—
		—	0	—	0	—	55	1	rw-
		101	1	—	0	—	45	1	rw-

Problem with 2 levels?

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

64-bit address:



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

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
- **Next Topic: What if desired address spaces do not fit in physical memory?**

**make
history.**



COMP SCI 3004

Operating Systems

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

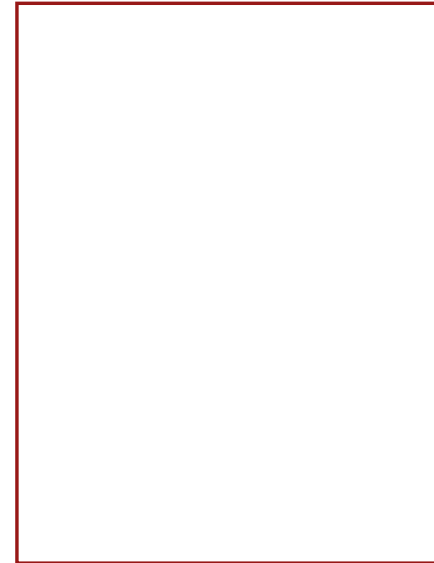
Virtual memory: OS provides illusion of more physical memory

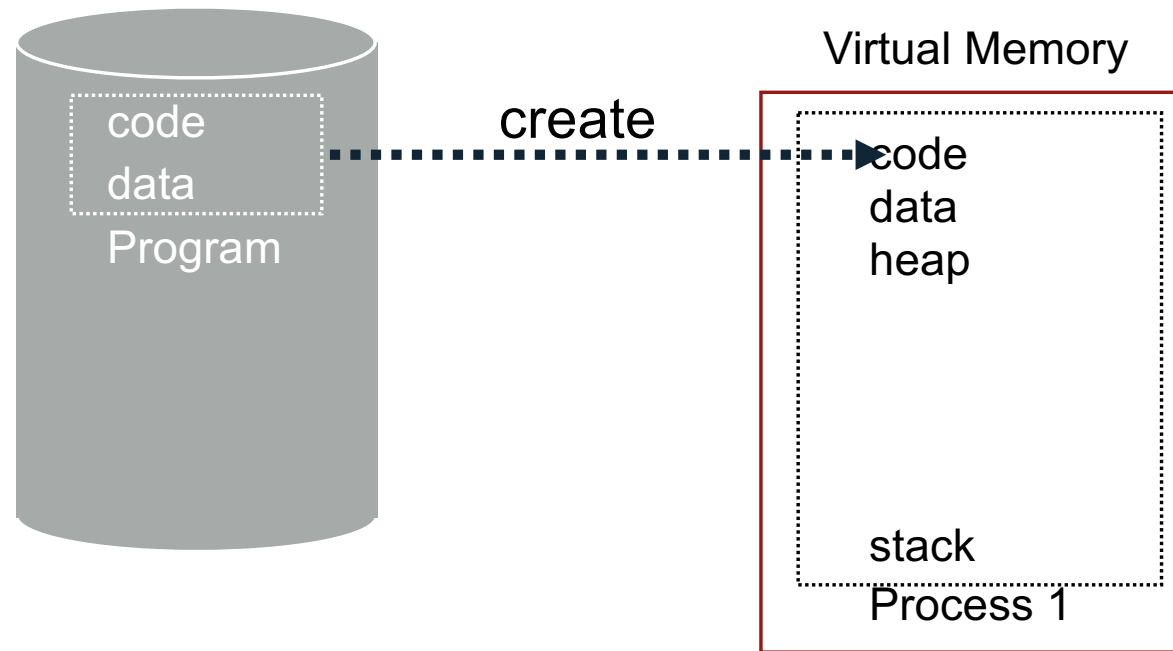
Why does this work?

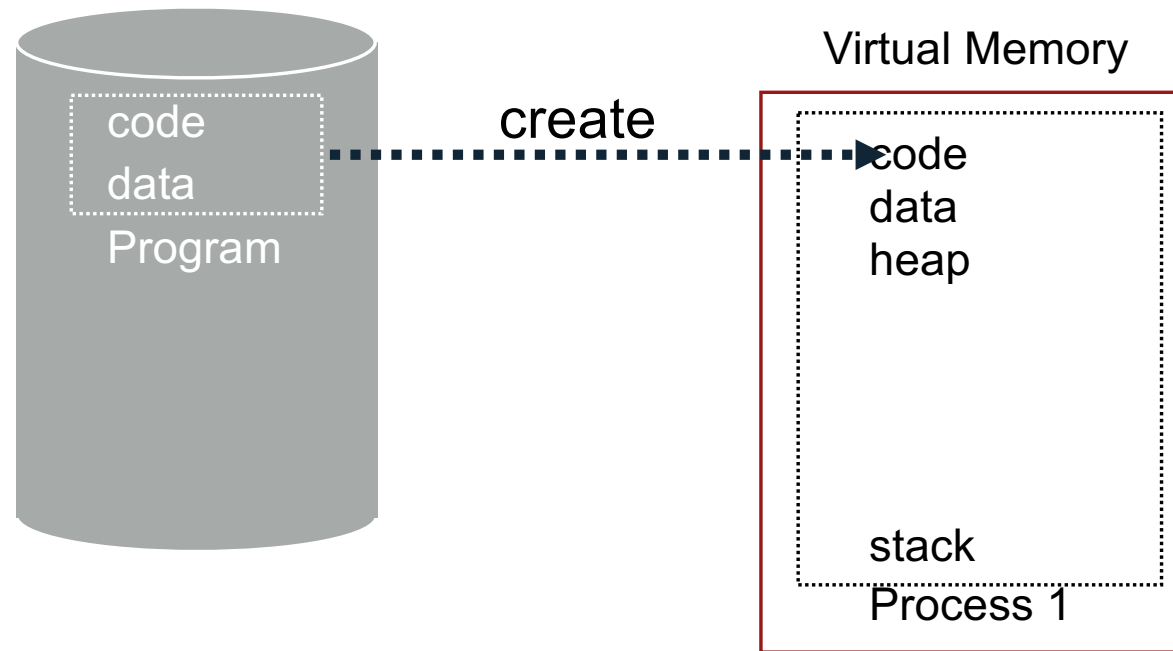
- Relies on key properties of user processes (workload) and machine architecture (hardware)



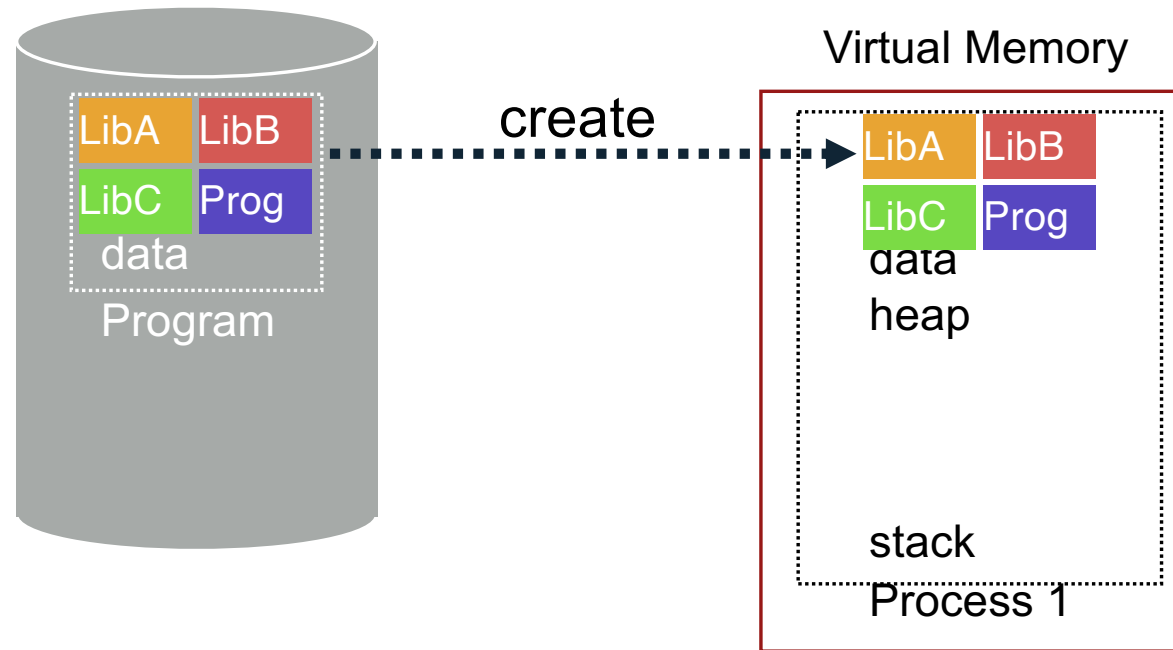
Virtual Memory





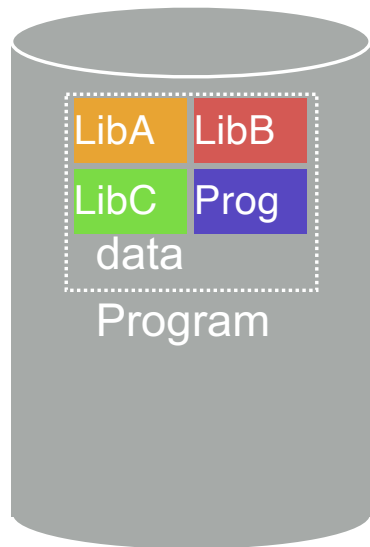


what's in code?

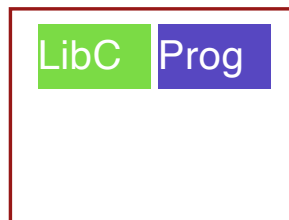


many large libraries, some
of which are rarely/never used

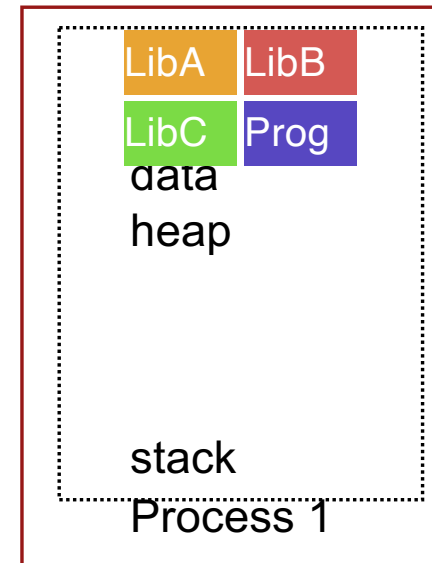
How to avoid wasting **physical pages** to back
rarely used **virtual pages**?

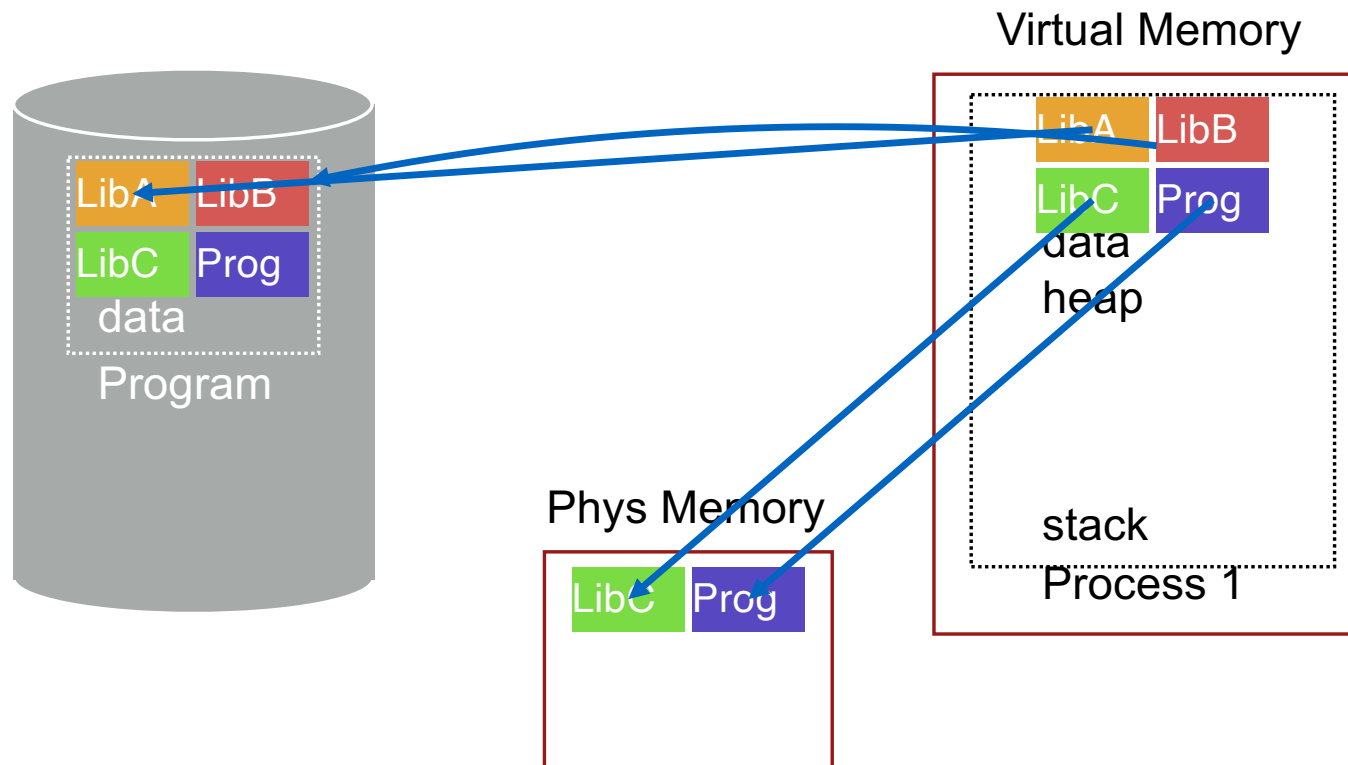


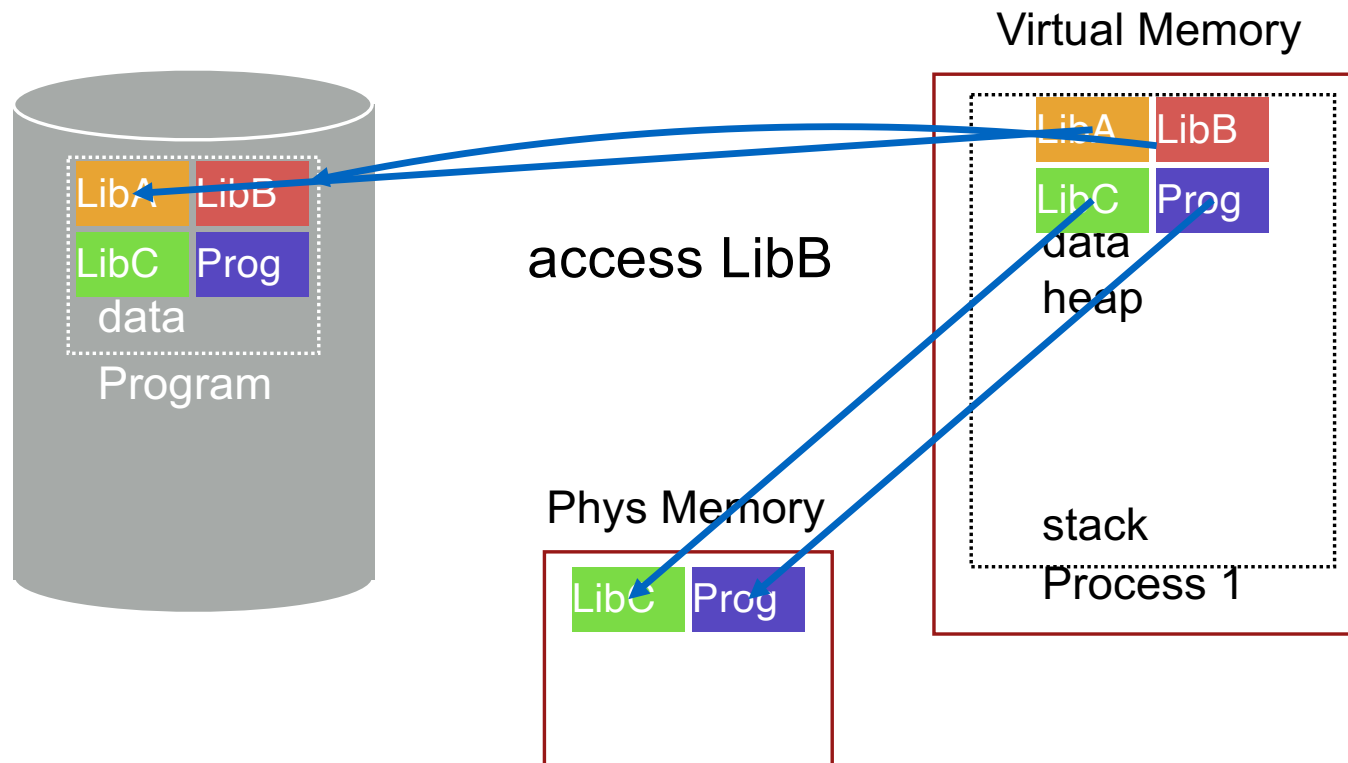
Phys Memory

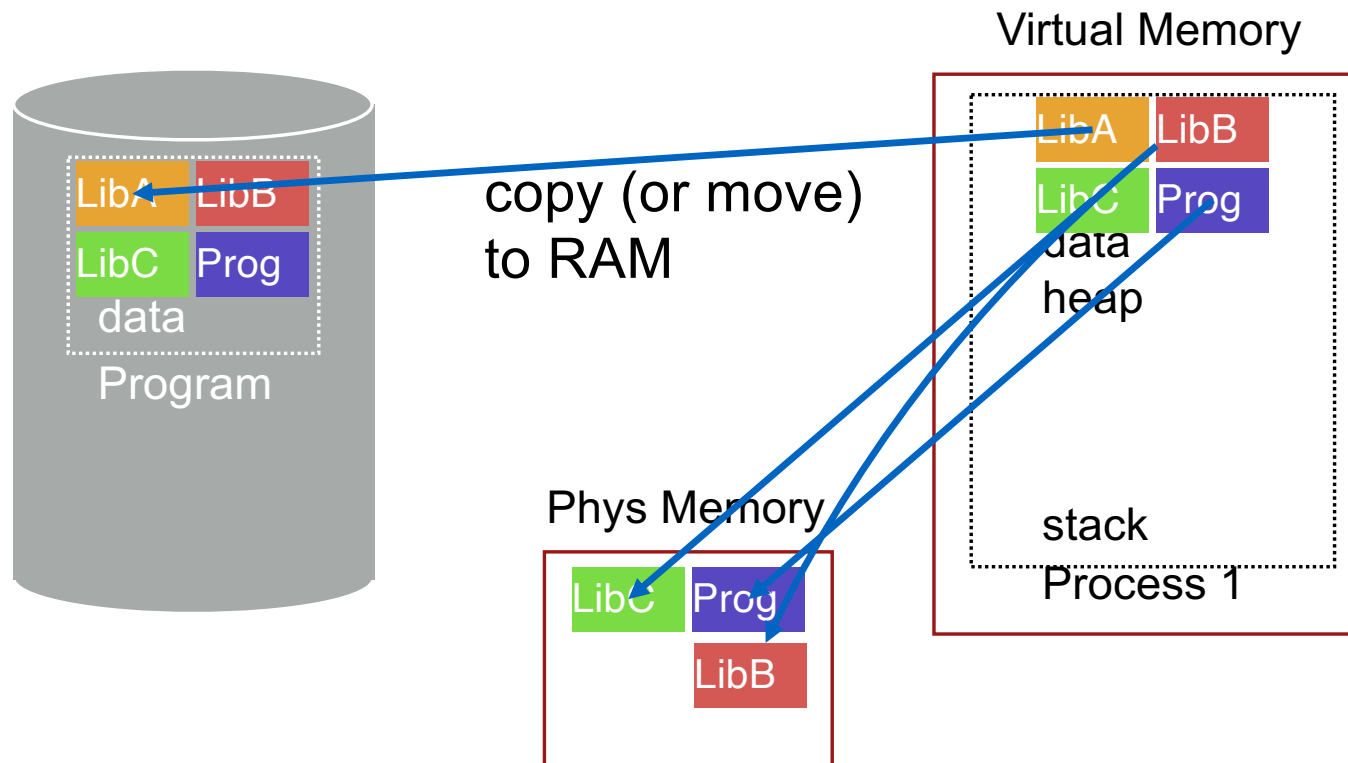


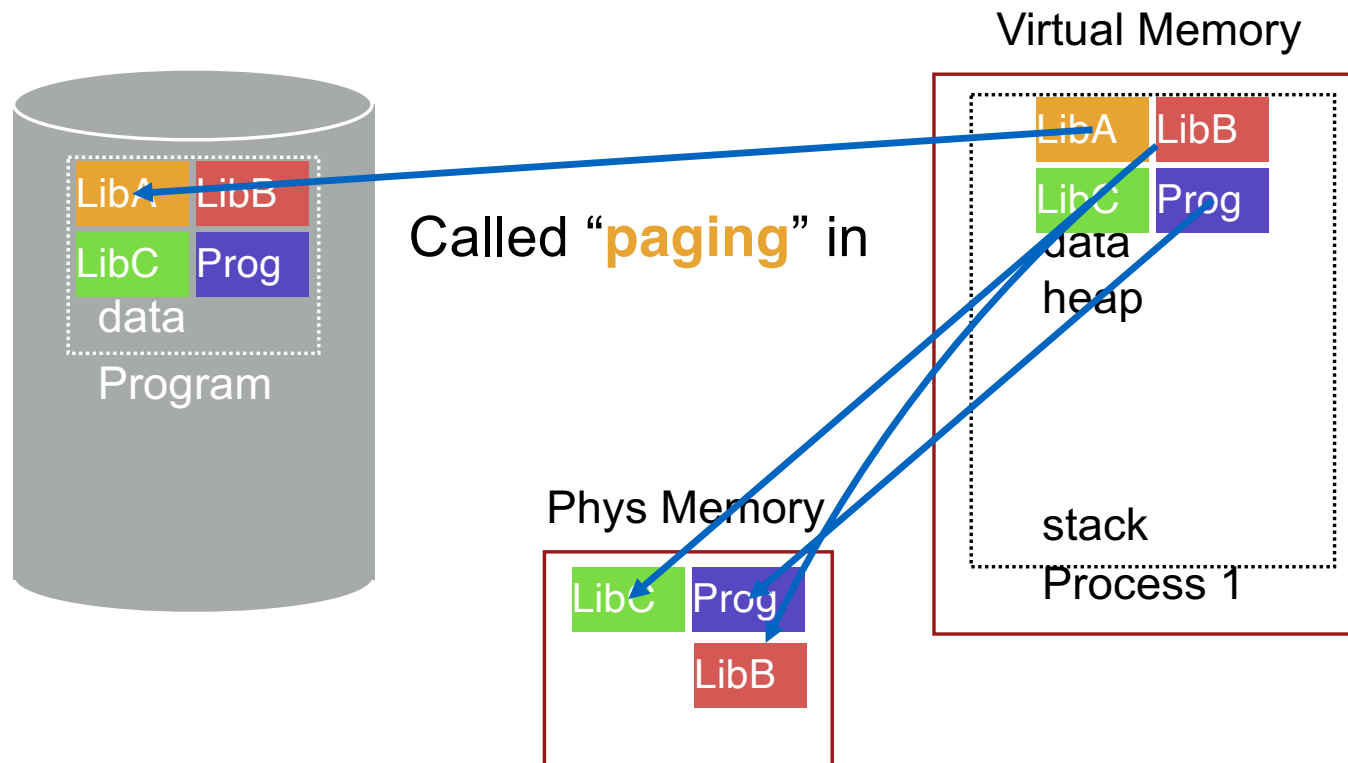
Virtual Memory









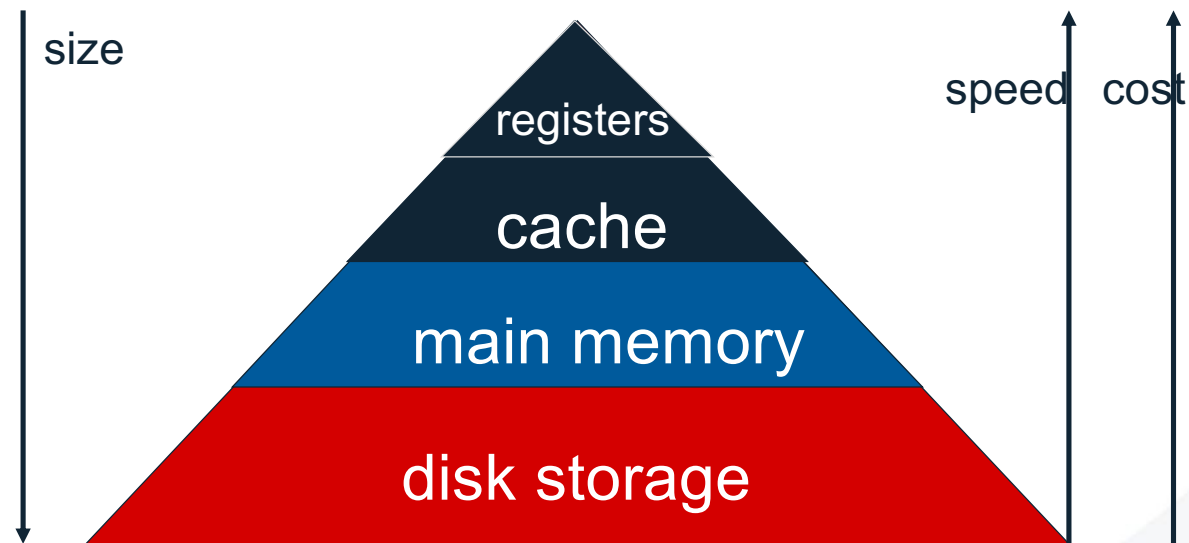


Locality of Reference

- 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



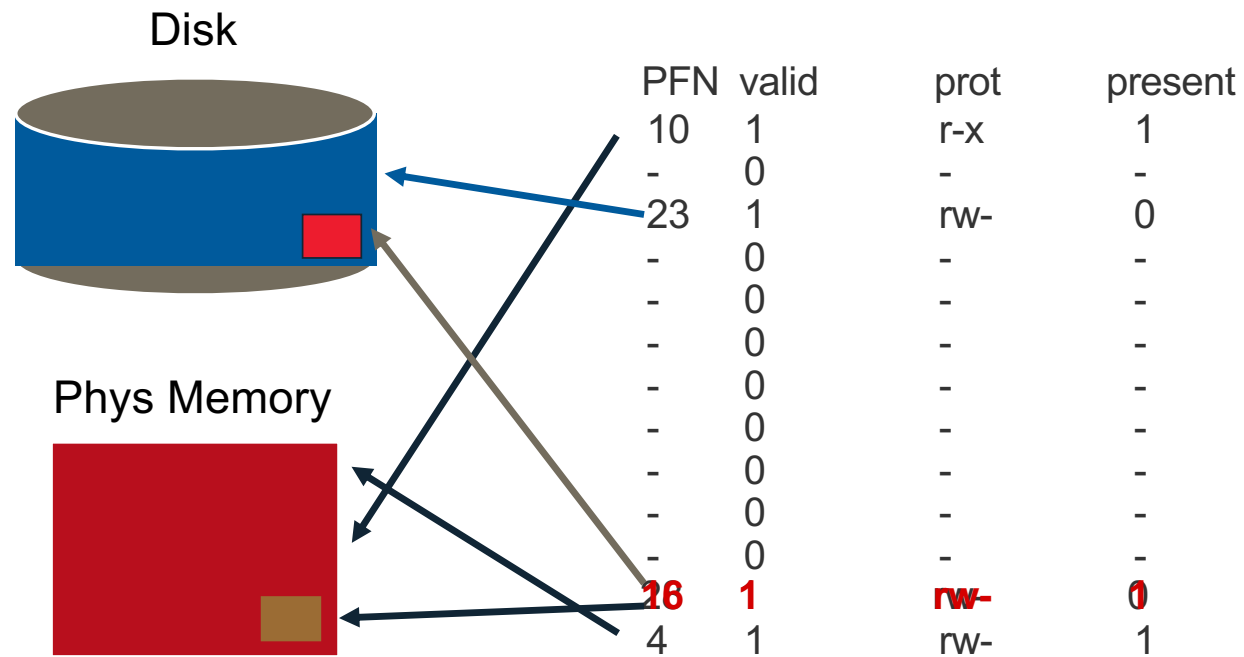
Virtual Memory Intuition

- Idea: OS need a place to stash away portions of address space that currently are not in great demand.
- Process can run when not all pages are loaded into main memory
- OS and hardware cooperate to provide illusion of large disk as fast as 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 for determining 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

Present Bit



What if access vpn 0xb?

Virtual Memory Mechanisms

- **Hardware and OS cooperate to translate addresses**
- **First, hardware checks TLB for virtual address**
 - if TLB hit, address translation is done; page in physical memory
- **If TLB miss...**
 - Hardware or OS walk page tables
 - If PTE designates page is present, then page in physical memory

Virtual Memory Mechanisms

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

Mechanism for continuing a process

- **Continuing a process after a page fault is tricky**
 - Want page fault to be transparent to user
 - Page fault may have occurred in middle of instruction
 - Requires hardware support
- **Complexity depends upon instruction set**
 - Can faulting instruction be restarted from beginning?

Virtual Memory Policies

- **Goal: Minimize number of page faults**
 - Page faults require milliseconds to handle (reading from disk)
 - Implication: Plenty of time for OS to make good decision
- **OS has two decisions**
 - Page selection
 - When should a page (or pages) on disk be brought into memory?
 - Page replacement
 - Which resident page (or pages) in memory should be thrown out to disk?

Page Selection

- **When should a page be brought from disk into memory?**
 - **Demand paging: Load page only when page fault occurs**
 - Intuition: Wait until page must absolutely be in memory
 - When process starts: No pages are loaded in memory
 - Problems: Pay cost of page fault for every newly accessed page
 - **Prepaging (anticipatory, prefetching): Load page before referenced**
 - OS predicts future accesses (oracle) and brings pages into memory early
 - Works well for some access patterns (e.g., sequential)

Page Replacement

- **Which page in main memory should selected as victim?**
 - Write out victim page to disk if modified (dirty bit set)
 - If victim page is not modified (clean), just discard
- **OPT: Replace page not used for longest time in future**
 - Advantages: Guaranteed to minimize number of page faults
 - Disadvantages: Requires that OS predict the future; Not practical, but good for comparison

Page Replacement Example

Reference Row

0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		0,1,2
1	Hit		0,1,2
3	Miss	2	0,1,3
0	Hit		0,1,3
3	Hit		0,1,3
1	Hit		0,1,3
2	Miss	3	0,1,2
1	Hit		0,1,2

Compulsory
Capacity
Conflict

Hit rate is $\frac{\text{Hits}}{\text{Hits} + \text{Misses}} = 54.6\%$

Future is not known.

Page Replacement

- **FIFO: Replace page that has been in memory the longest**
 - Intuition: First referenced long time ago, done with it now
 - Advantages: Fair: All pages receive equal residency; Easy to implement (circular buffer)
 - Disadvantage: Some pages may always be needed

Page Replacement: FIFO

Reference Row

0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		0,1,2
1	Hit		0,1,2
3	Miss	0	1,2,3
0	Miss	1	2,3,0
3	Hit		2,3,0
1	Miss		3,0,1
2	Miss	3	0,1,2
1	Hit		0,1,2

Hit rate is $\frac{\text{Hits}}{\text{Hits} + \text{Misses}} = 36.4\%$

Even though page 0 had been accessed a number of times, **FIFO still kicks it out.**

Page Replacement Comparison

- **Add more physical memory, what happens to performance?**
 - LRU, OPT: Add more memory, guaranteed to have fewer (or same number of) page faults
 - Smaller memory sizes are guaranteed to contain a subset of larger memory sizes
 - Stack property: smaller cache always subset of bigger
- FIFO: Add more memory, usually have fewer page faults
 - Belady's anomaly: May actually have more page faults!

Page Replacement

- **LRU: Least-recently-used: Replace page not used for longest time in past**
 - Intuition: Use past to predict the future
 - Advantages: With locality, LRU approximates OPT
 - Disadvantages: Harder to implement, must track which pages have been accessed, and does not handle all workloads well

Page Replacement: LRU

Reference Row

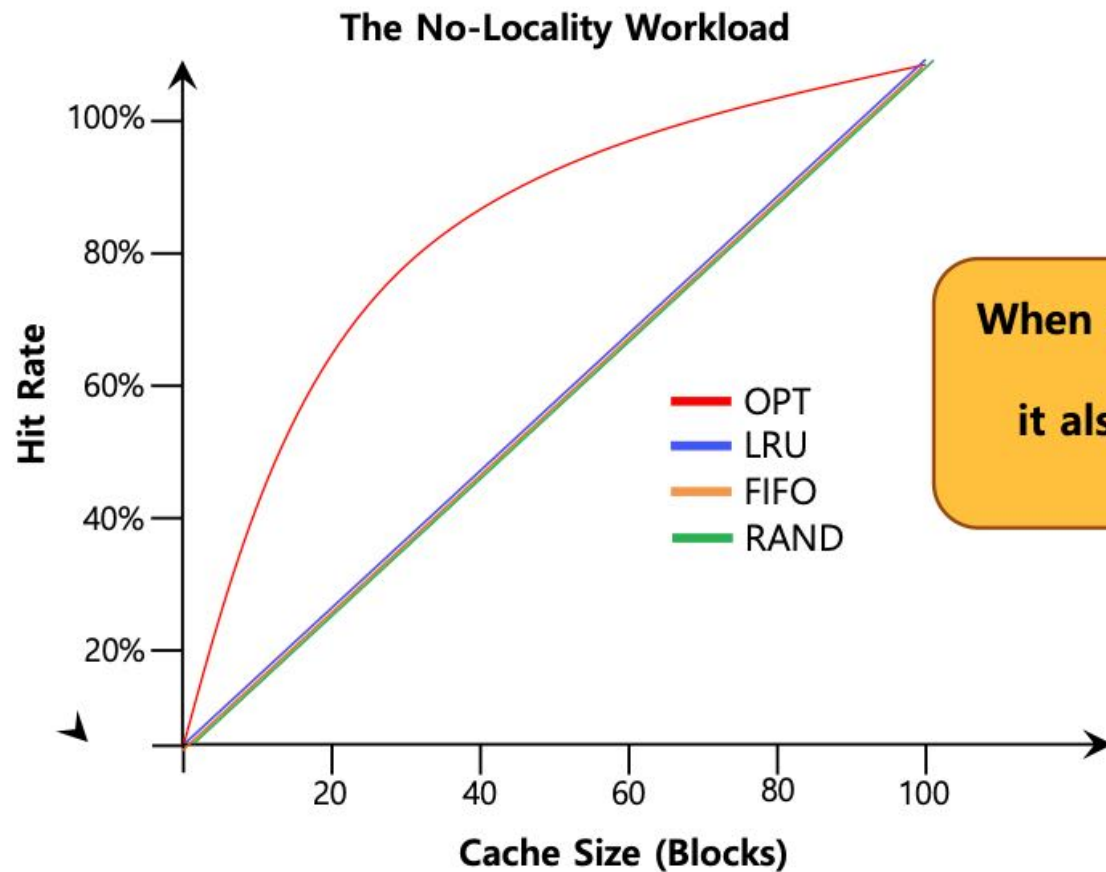
0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		1,2,0
1	Hit		2,0,1
3	Miss	2	0,1,3
0	Hit		1,3,0
3	Hit		1,0,3
1	Hit		0,3,1
2	Miss	0	3,1,2
1	Hit		3,2,1

Problems with LRU-based Replacement

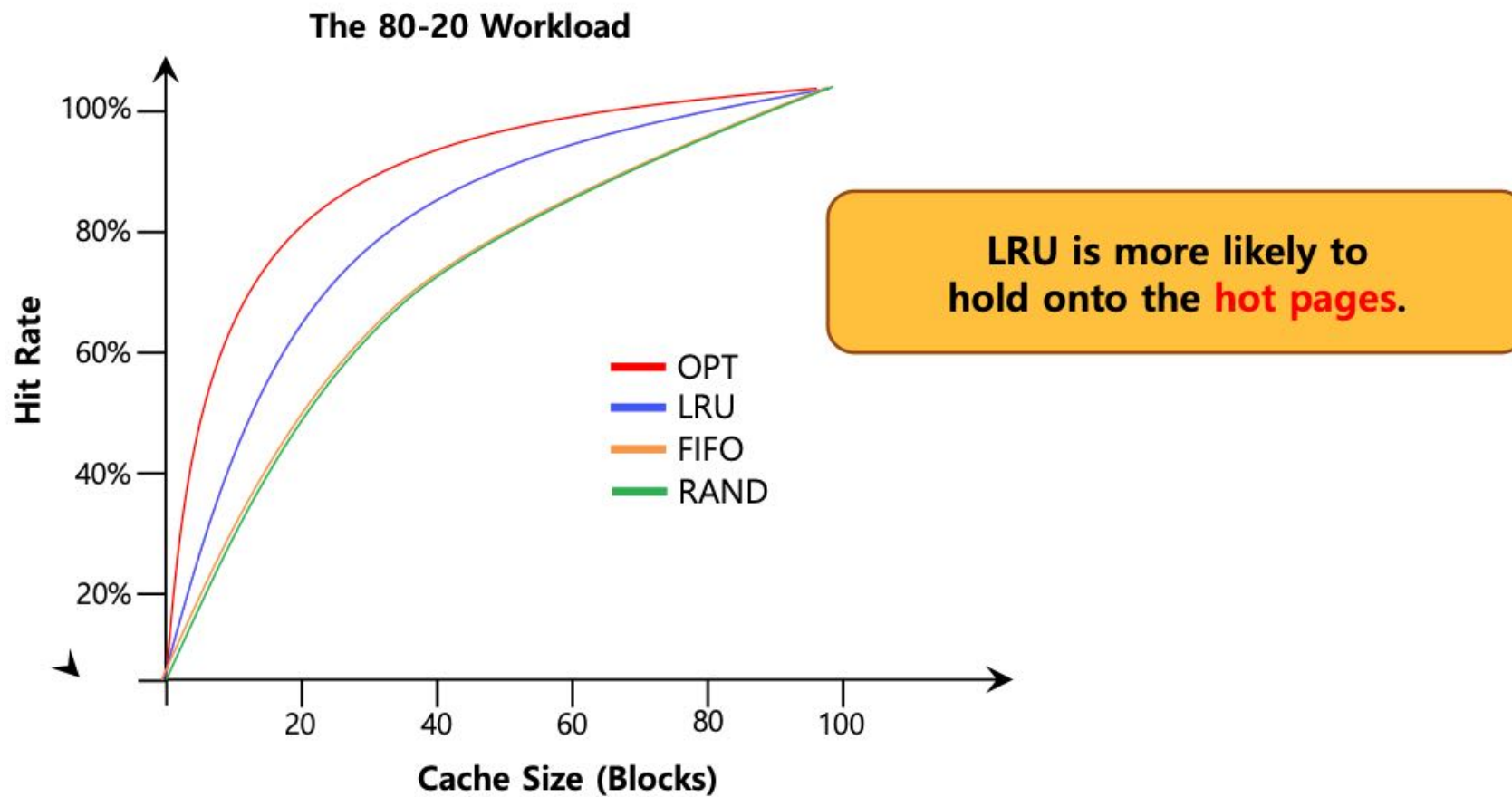
- **LRU does not consider frequency of accesses**
 - Is a page accessed once in the past equal to one accessed N times?
 - Common workload problem - Scan (sequential read, never used again) one large data region flushes memory
 - Solution: Track frequency of accesses to page
- **Pure LFU (Least-frequently-used) replacement**
 - Problem: LFU can never forget pages from the far past
- **Examples of other more sophisticated algorithms:**
 - LRU-K and 2Q: Combines recency and frequency attributes
 - Expensive to implement, LRU-2 used in databases

Workload Example : The No-Locality Workload



When the cache is large enough to fit the entire workload, it also **doesn't matter** which policy you use.

Workload Example : The 80-20 Workload



Implementing LRU

- **Software Perfect LRU**
 - OS maintains ordered list of physical pages by reference time
 - When page is referenced: Move page to front of list
 - When need victim: Pick page at back of list
 - Trade-off: Slow on memory reference, fast on replacement

Implementing LRU

- **Hardware Perfect LRU**
 - Associate timestamp register with each page
 - When page is referenced: Store system clock in register
 - When need victim: Scan through registers to find oldest clock
 - Trade-off: Fast on memory reference, slow on replacement (especially as size of memory grows)
- **In practice, do not implement Perfect LRU**
 - LRU is an approximation anyway, so approximate more
 - Goal: Find an old page, but not necessarily the very oldest

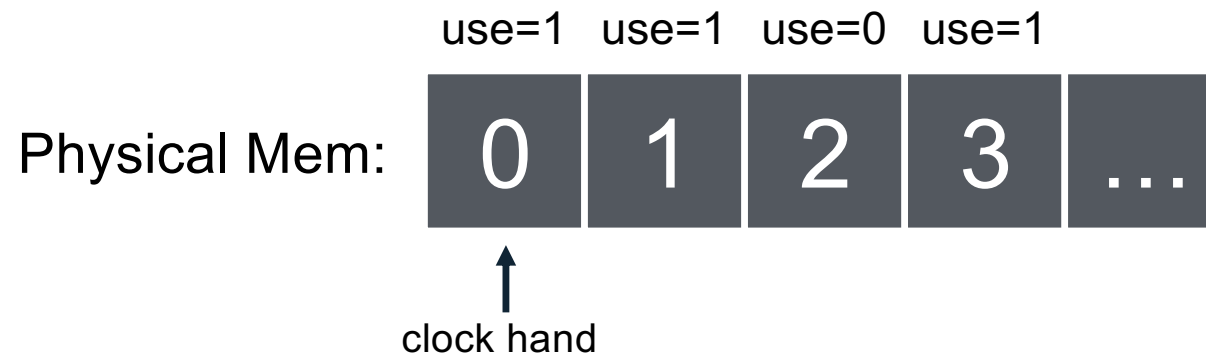
Clock Algorithm

- **Hardware**
 - Keep use (or reference) bit for each page frame
 - When page is referenced: set use bit

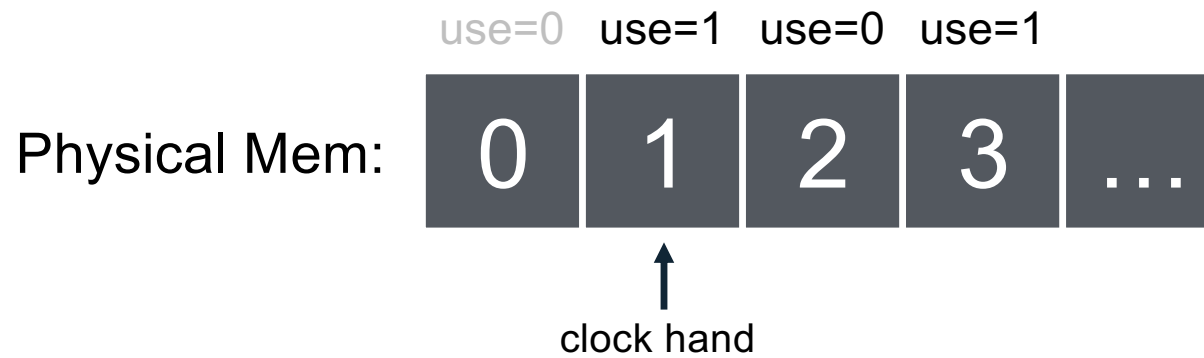
Clock Algorithm

- **Operating System**
 - Page replacement: Look for page with use bit cleared (has not been referenced for awhile)
 - Implementation:
 - Keep pointer to last examined page frame
 - Traverse pages in circular buffer
 - Clear use bits as search
 - Stop when find page with already cleared use bit, replace this page

Clock: Look For a Page



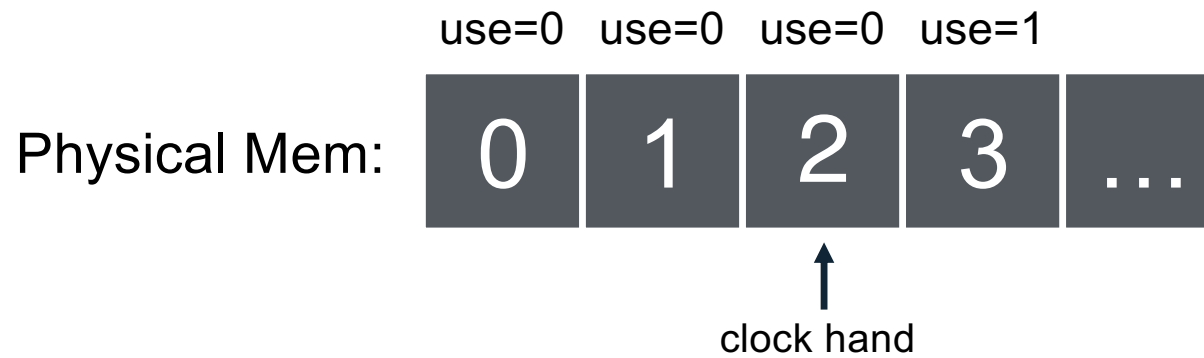
Clock: Look For a Page



Clock: Look For a Page

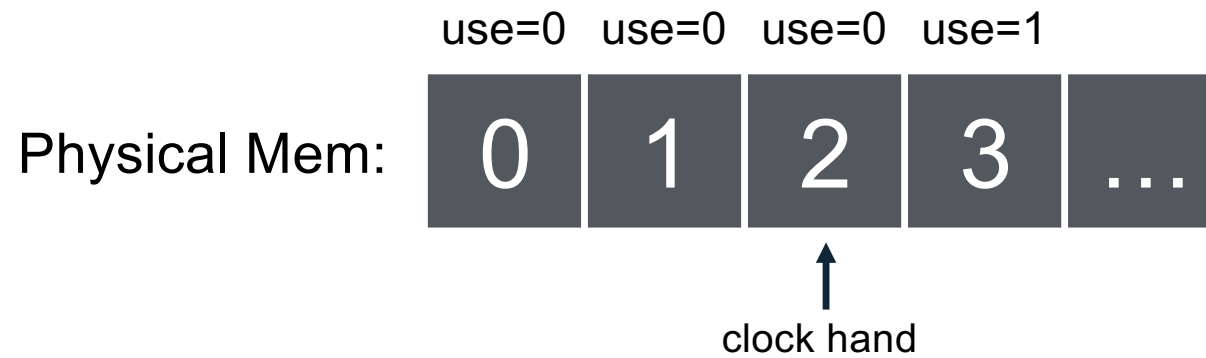


Clock: Look For a Page



evict **page 2** because it has not been recently used

Clock: Look For a Page

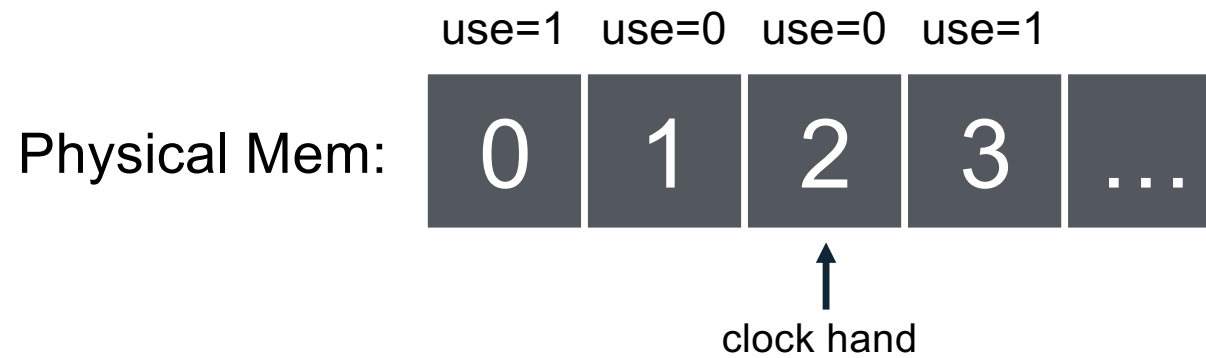


page 0 is accessed...

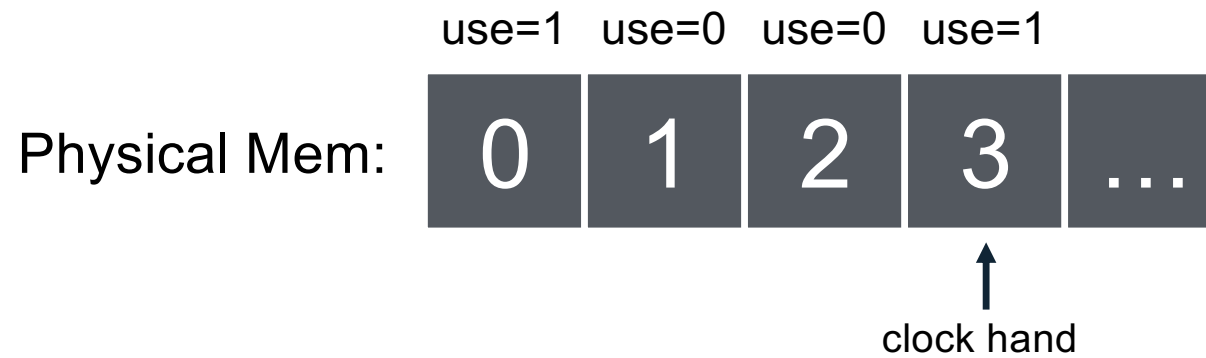
Clock: Look For a Page



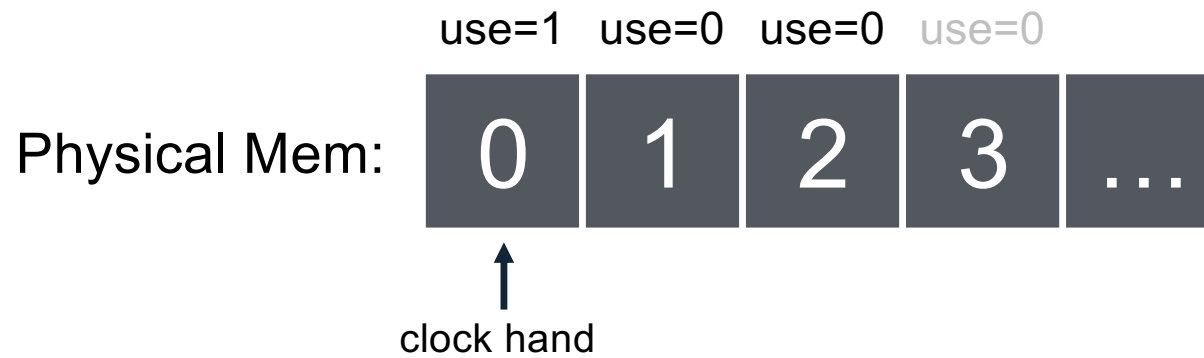
Clock: Look For a Page



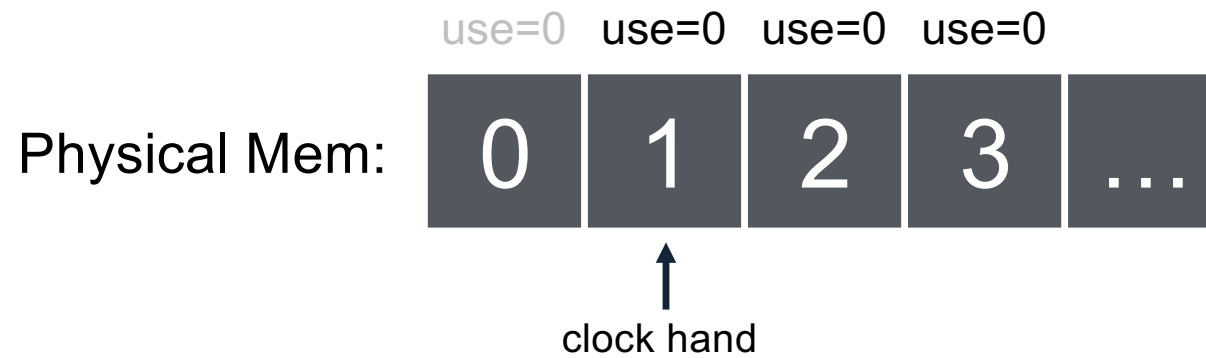
Clock: Look For a Page



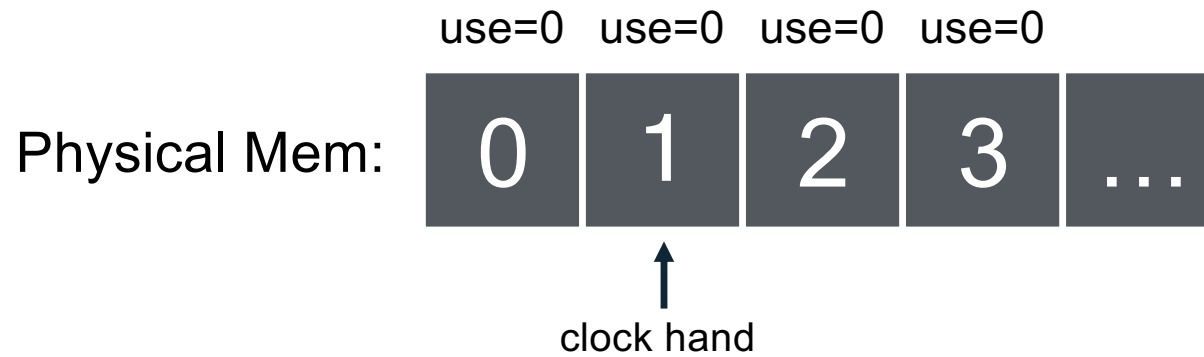
Clock: Look For a Page



Clock: Look For a Page

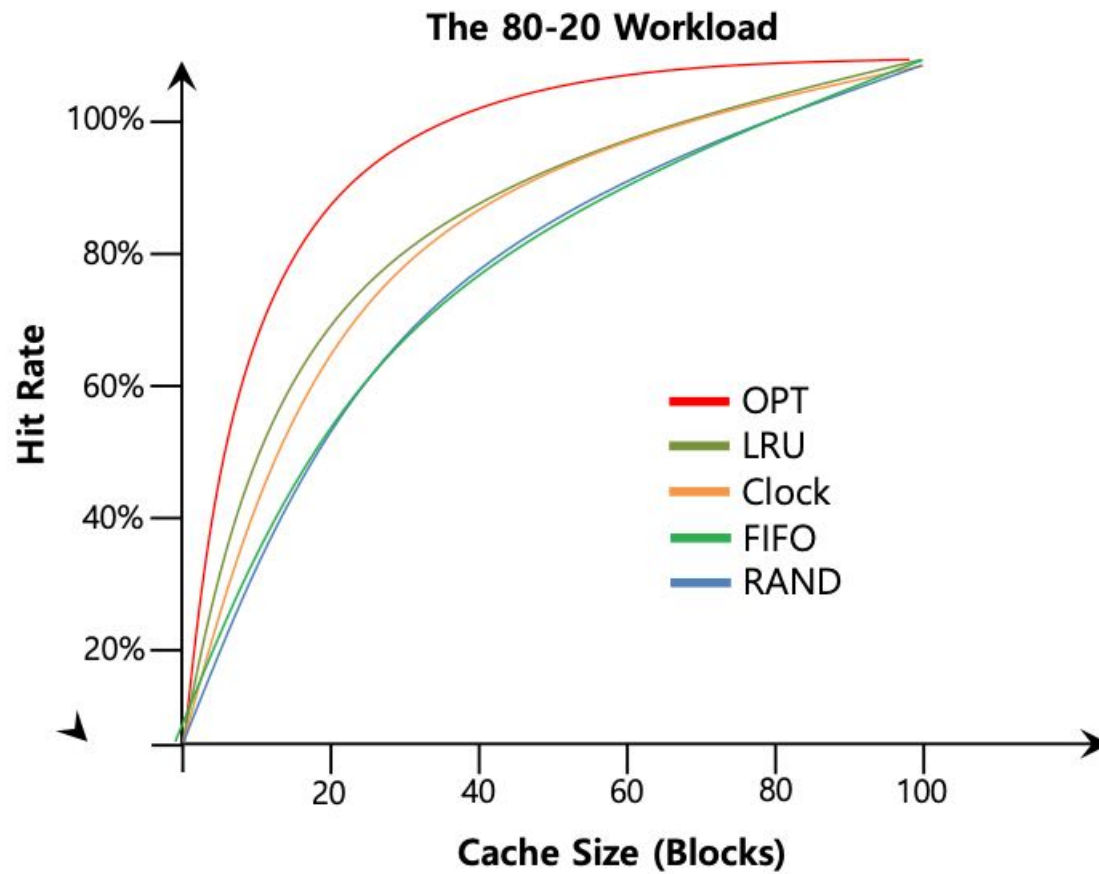


Clock: Look For a Page



evict **page 1** because it has not been recently used

Workload with Clock Algorithm



Clock Extensions

- **Replace multiple pages at once**
 - Intuition: Expensive to run replacement algorithm and to write single block to disk
 - Find multiple victims each time and track free list
- **Add software counter (“chance”)**
 - Intuition: Better ability to differentiate across pages (how much they are being accessed)
 - Increment software counter if use bit is 0
 - Replace when chance exceeds some specified limit

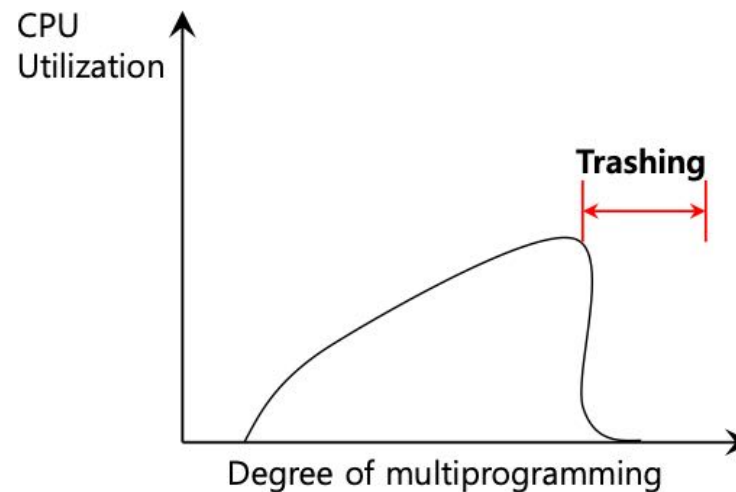
Clock Extensions

- **Use dirty bit to give preference to dirty pages**
 - Intuition: More expensive to replace dirty pages
 - Dirty pages must be written to disk, clean pages do not
 - Replace pages that have use bit and dirty bit cleared

Thrashing

Memory is **oversubscribed** and the memory demands of the set of running processes **exceeds** the available physical memory.

- Decide not to run a subset of processes.
- Reduced set of processes working sets fit in memory. w Linux take : OOM (**out-of-memory killer**)



Summary

- **Illusion of virtual memory:**
Processes can run when sum of virtual address spaces > amount of physical memory
- **Mechanism:**
 - Extend page table entry with “present” bit
 - OS handles page faults (or page misses) by reading in desired page from disk
- **Policy:**
 - Page selection – demand paging, prefetching, hints
 - Page replacement – OPT, FIFO, LRU, others
- **Implementations (clock) perform approximation of LRU**