Virtual Memory: Policy

Questions answered in this lecture:

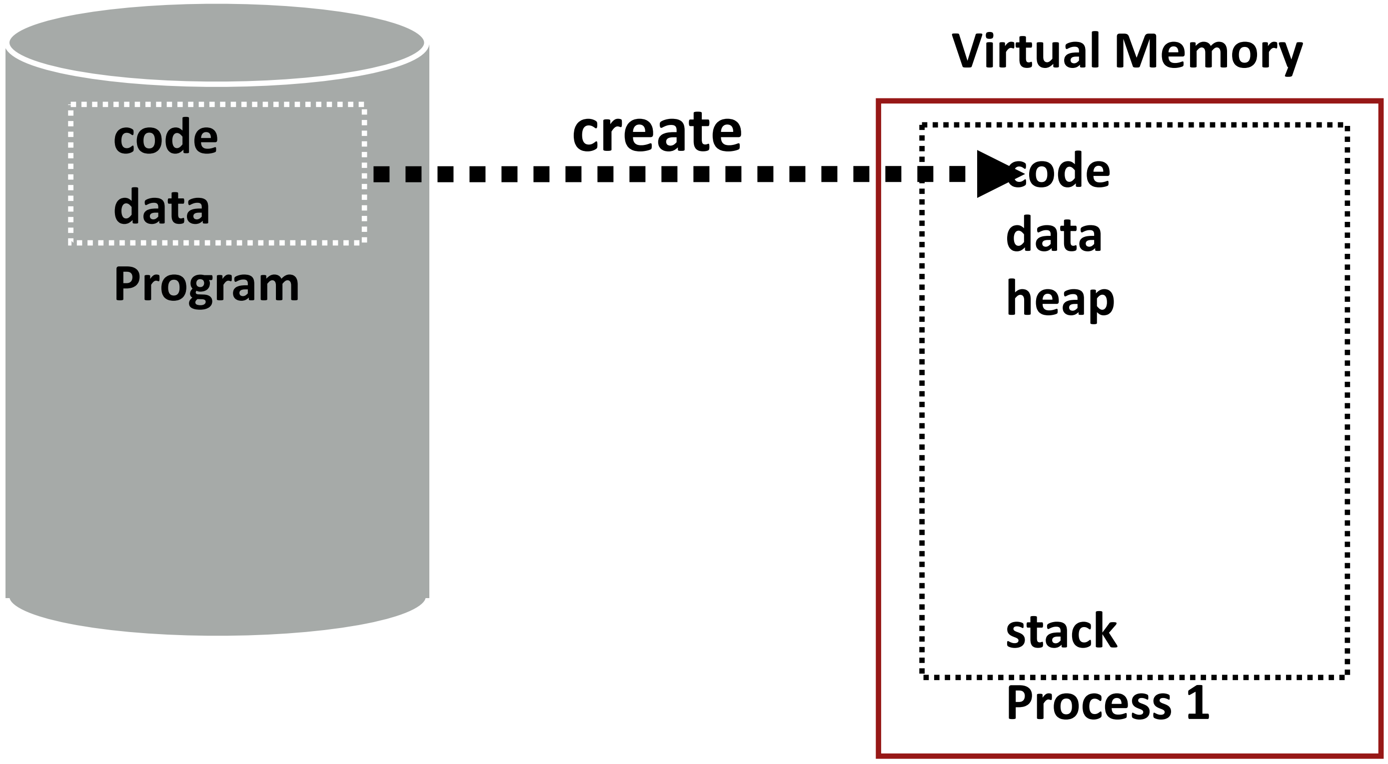
- How to run process when not enough physical memory?
- When should a page be moved from disk to memory?
- What page in memory should be replaced?
- How can the LRU page be approximated efficiently?

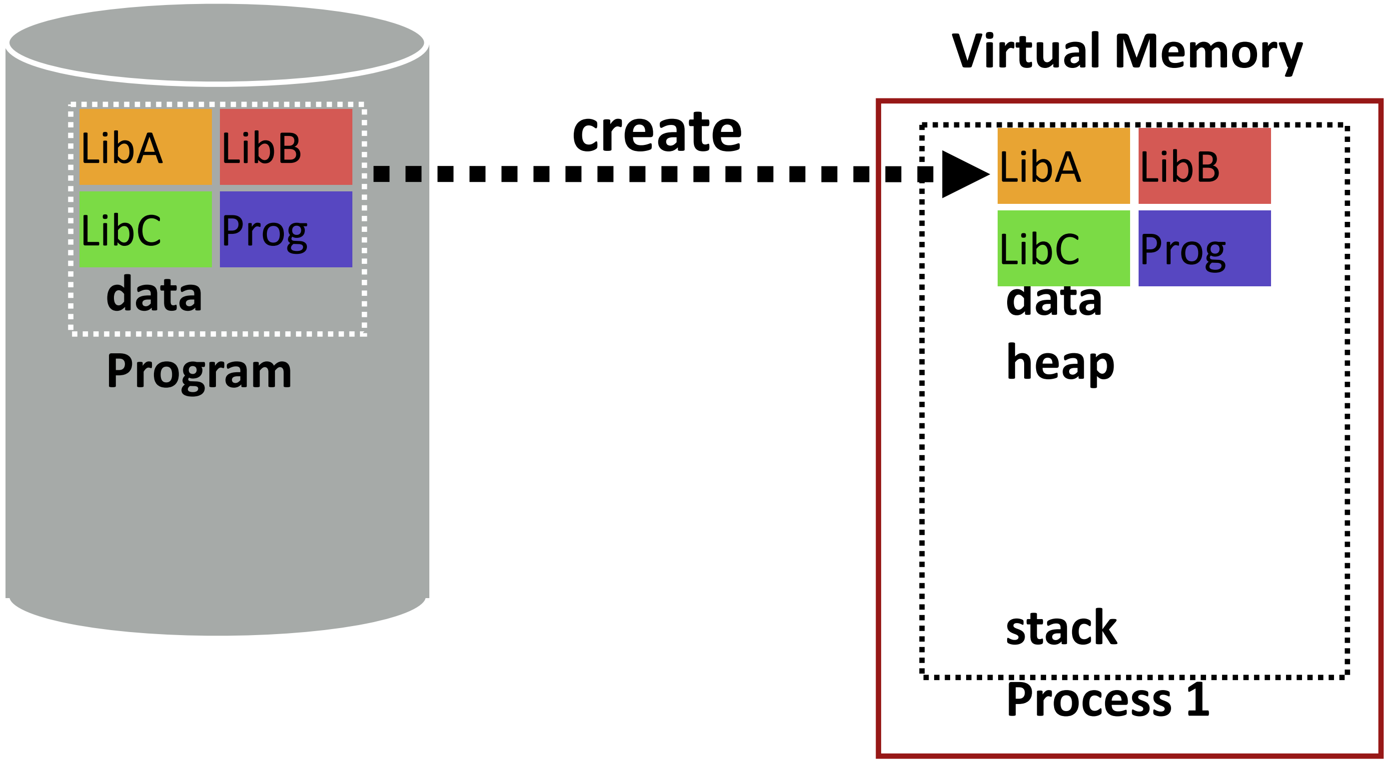
Motivation

- **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

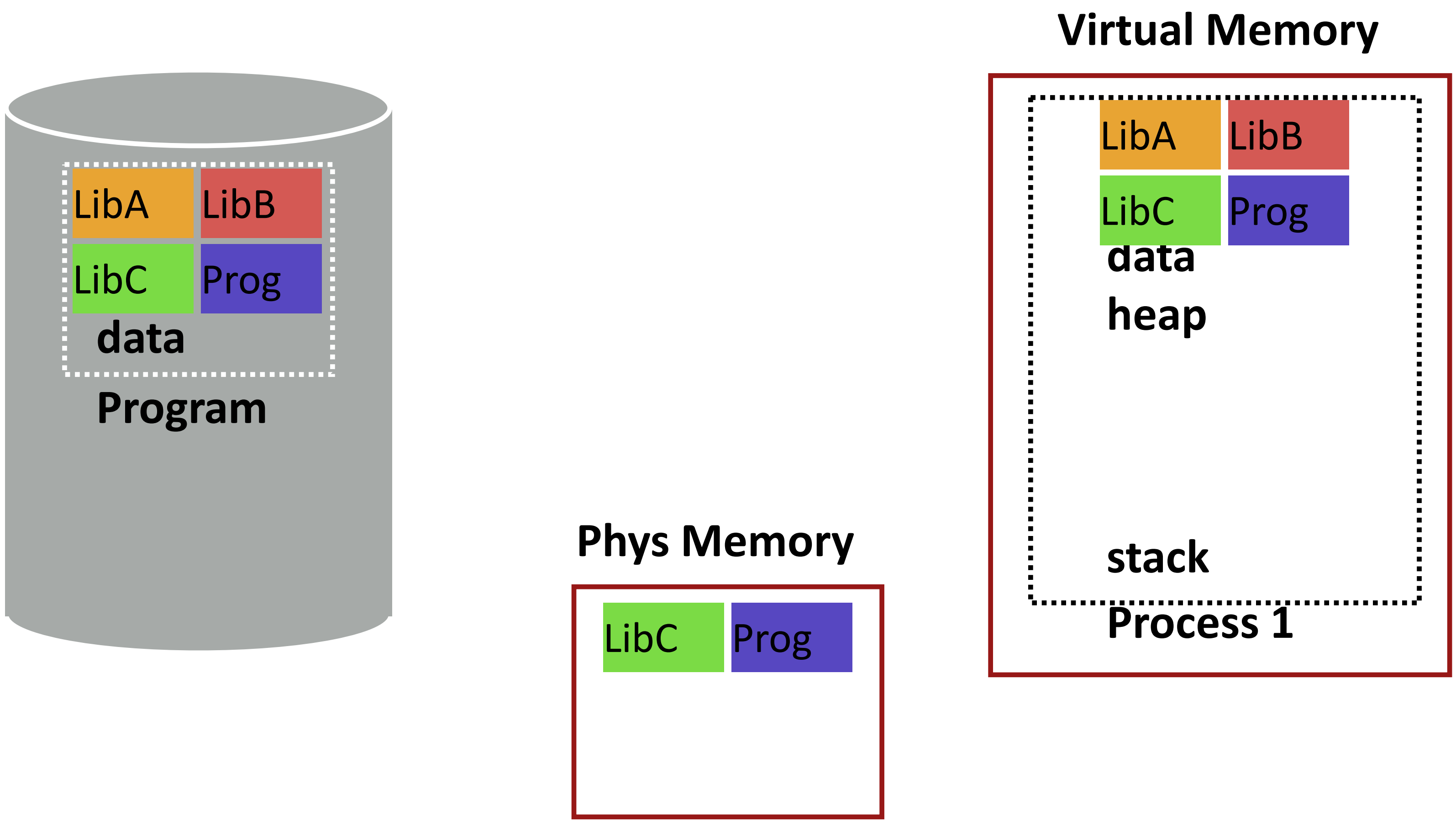
- **Virtual memory: OS provides illusion of more physical memory**
- **Why does this work?**
  - Relies on key properties of user processes (workload) and machine architecture (hardware)


code
data
Program
create
Virtual Memory
code
data
heap
stack
Process 1

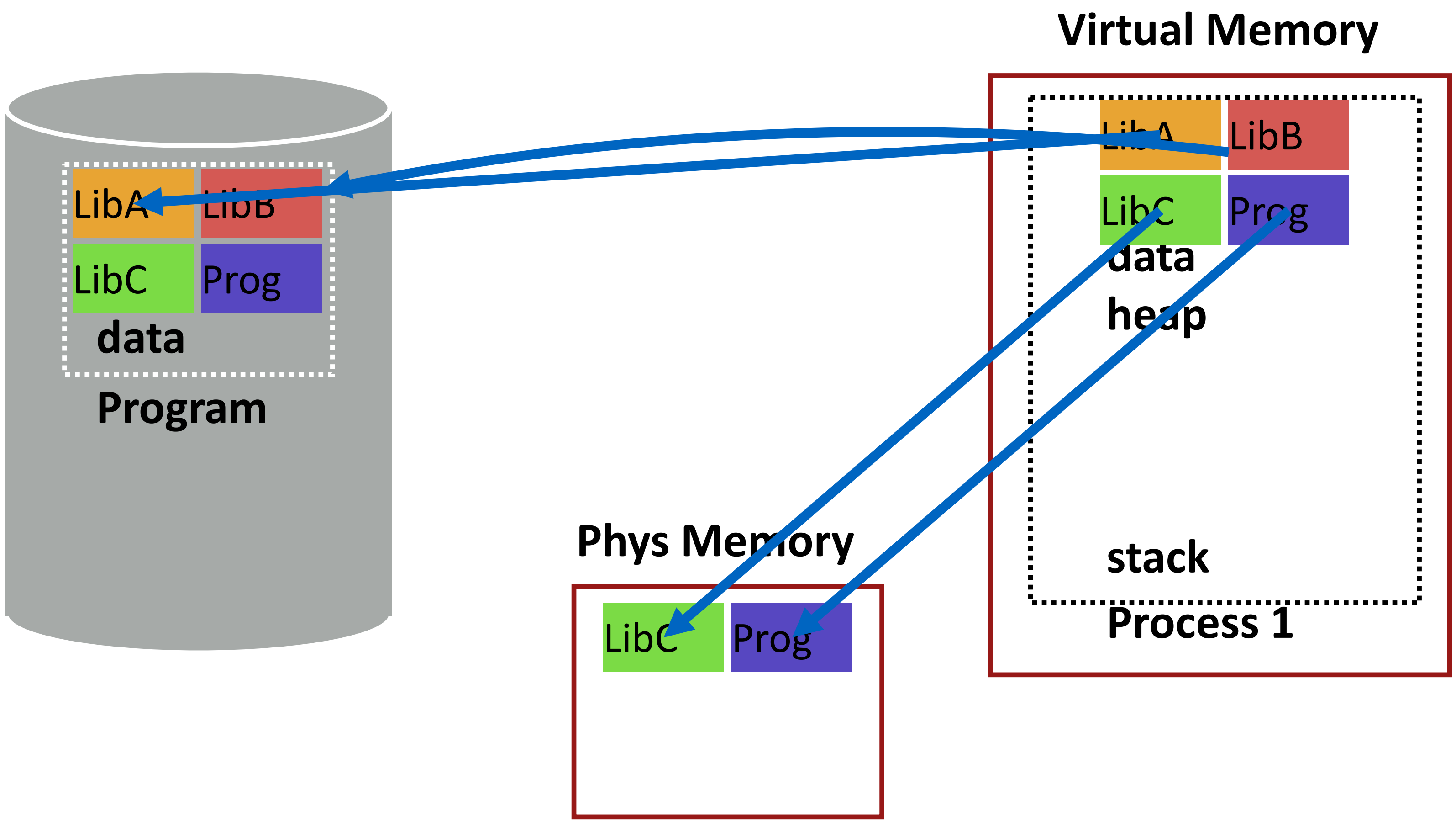


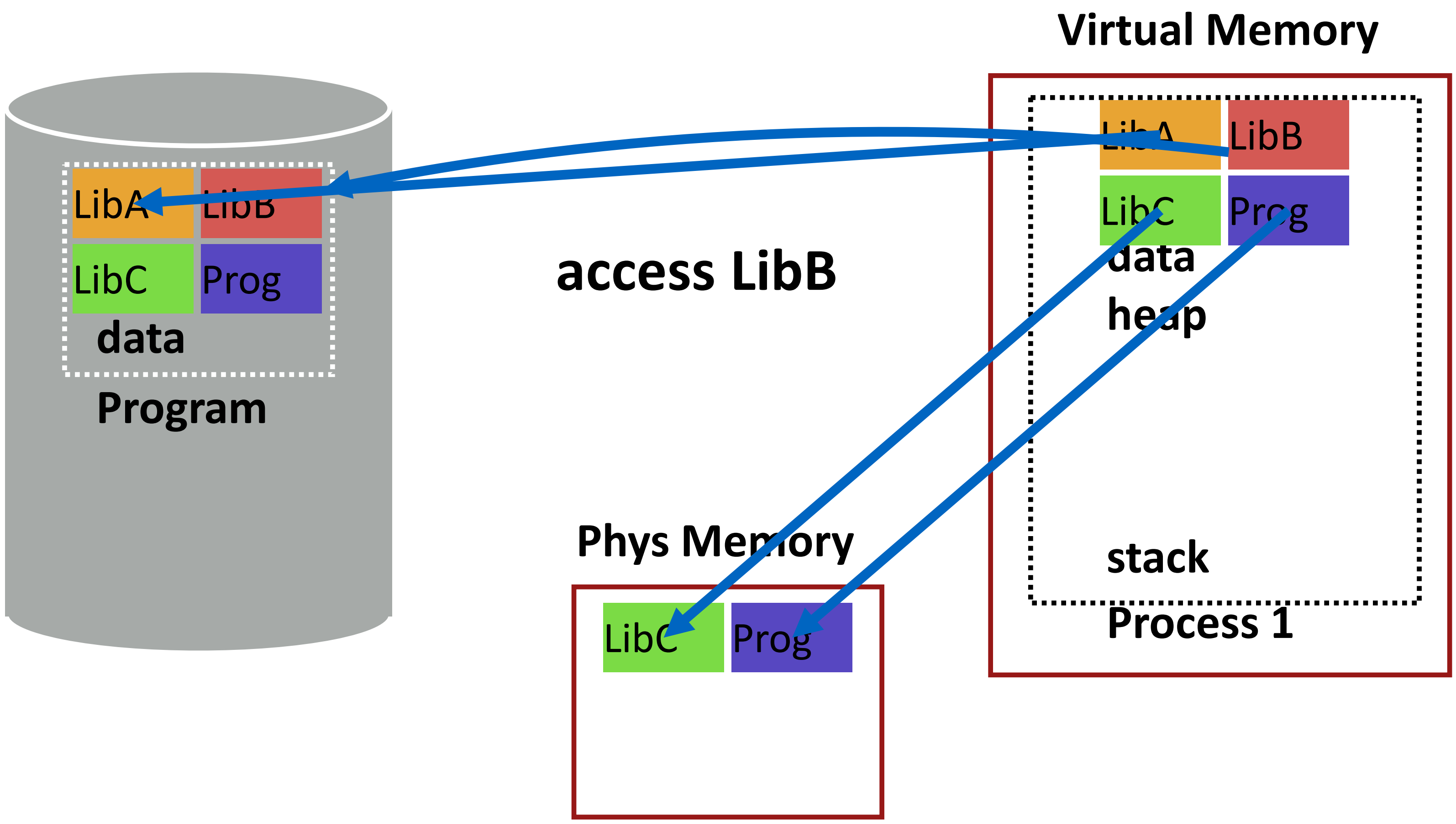
many large libraries, some
of which are rarely/never used

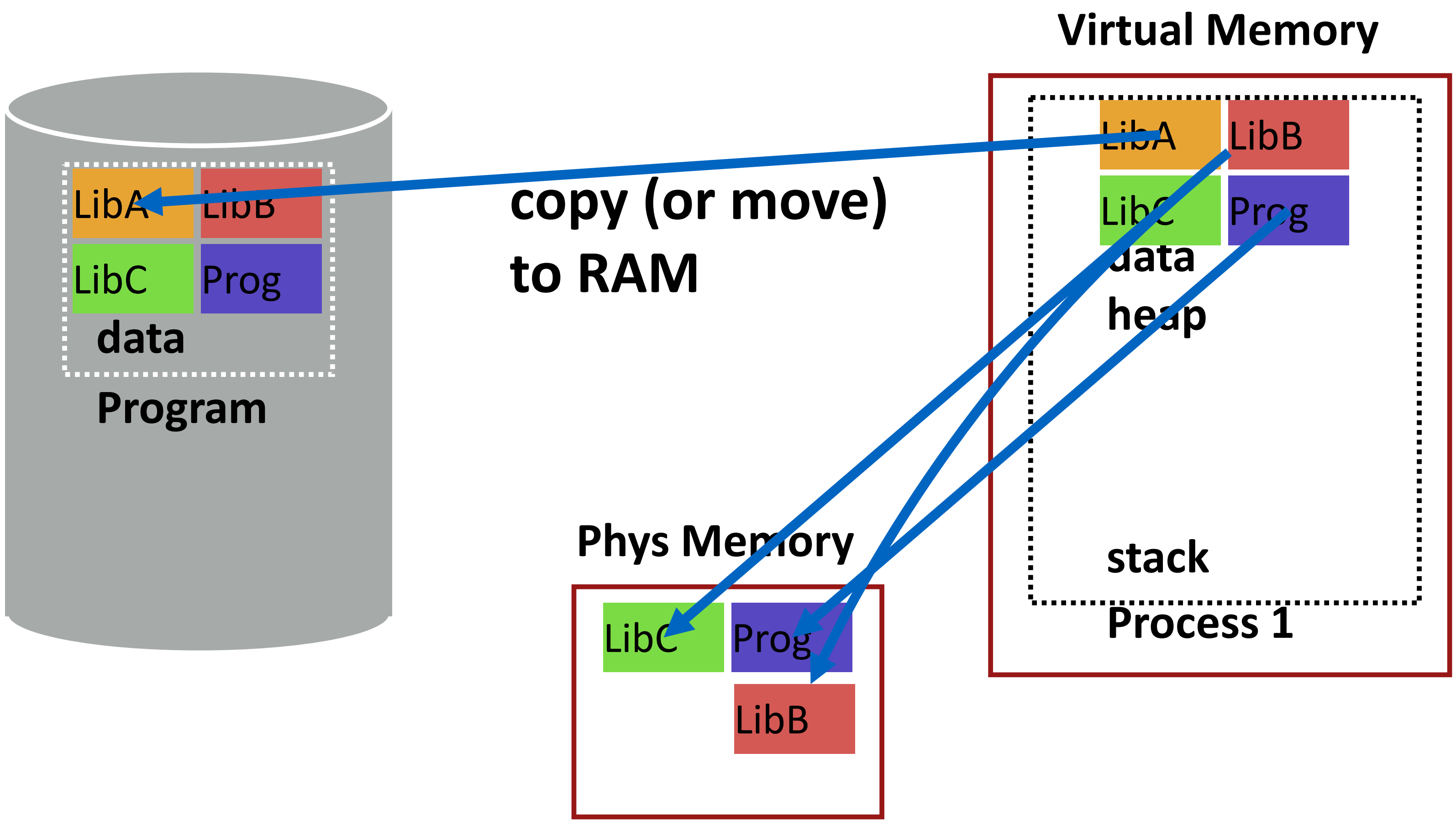
How to avoid wasting physical pages to back rarely used virtual pages?



LibA
LibB
LibC
Prog
data
Program
Phys Memory
LibC
Prog
Virtual Memory
LibA
LibB
LibC
Prog
data
heap
stack
Process 1


Virtual Memory
LibA
LibB
LibC
Prog
data
Program
LibA
LibB
LibC
Prog
data
heap
stack
Process 1
Phys Memory
LibC
Prog


Virtual Memory
LibA
LibB
LibC
Prog
data
heap
stack
Process 1
LibA
LibB
LibC
Prog
data
Program
access LibB
Phys Memory
LibC
Prog

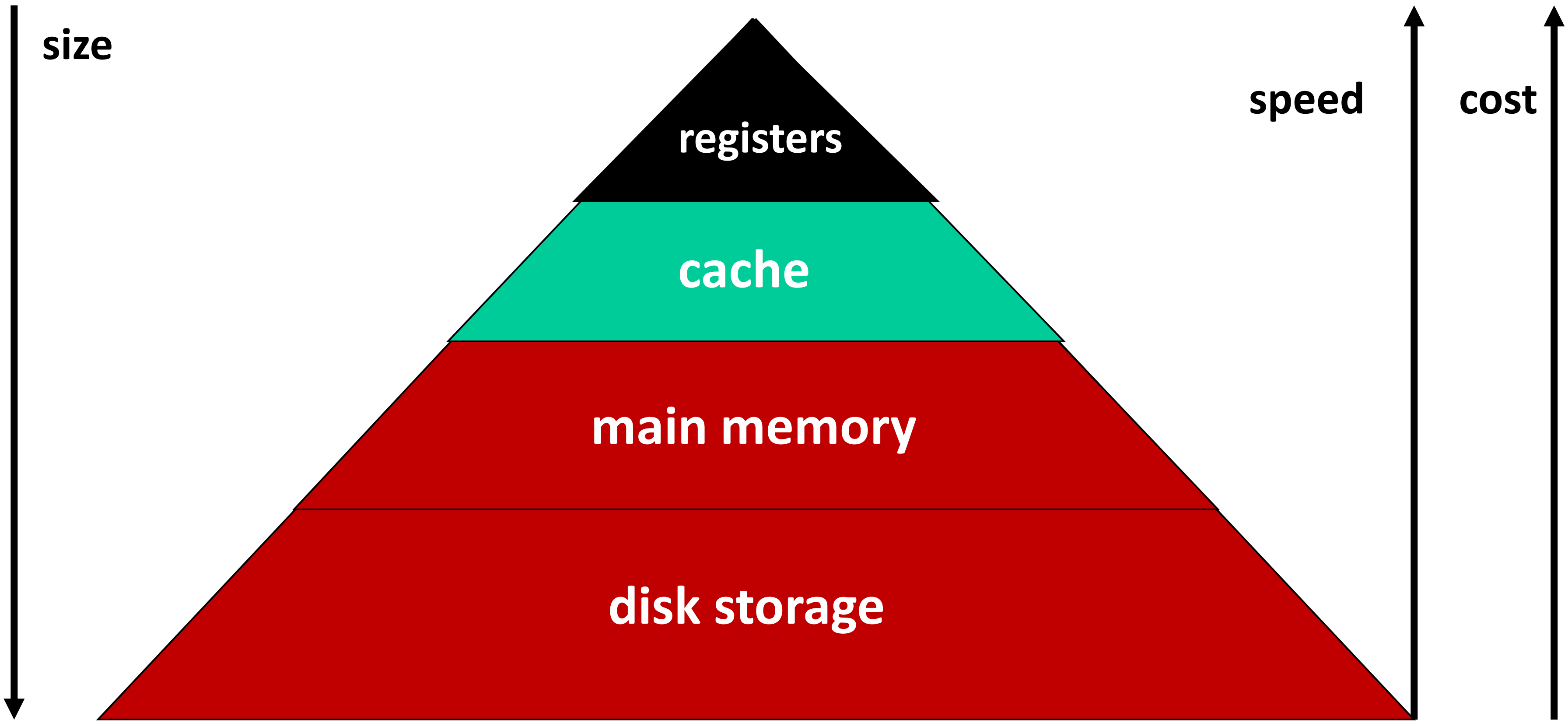

Virtual Memory
LibA
LibB
LibC
Prog
data
heap
stack
Process 1
copy (or move)
to RAM
LibA
LibB
LibC
Prog
data
Program
Phys Memory
LibC
Prog
LibB

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

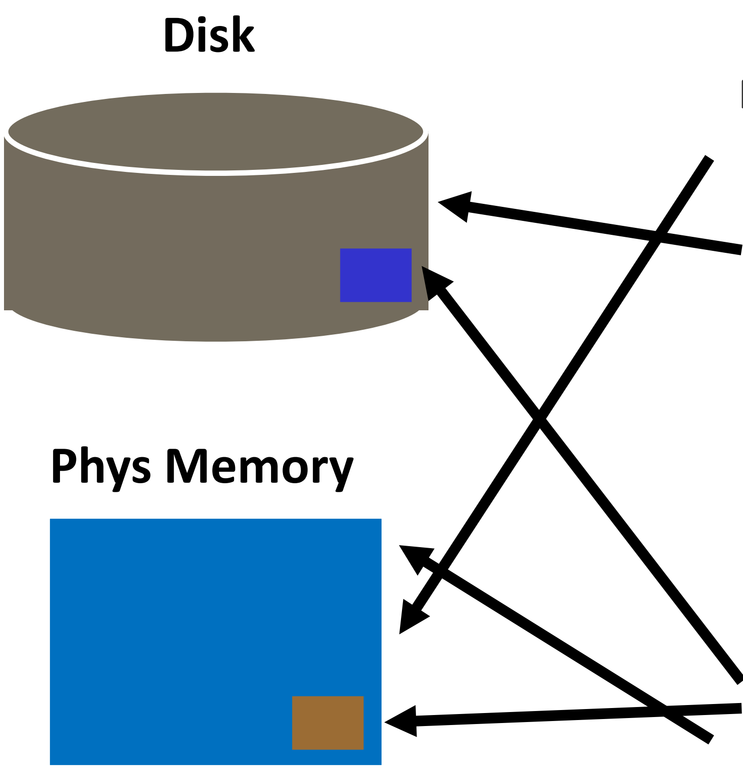
- **Idea: OS keeps unreferenced pages on disk**
  - Slower, cheaper backing store than memory
- **Process can run when not all pages are loaded into main memory**
- **OS and hardware cooperate to provide illusion of large memory**
  - Same behavior as if all of address space in main memory
  - Hopefully have similar performance
  - What should be done?
- **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
    - PTE points to block on disk
    - Causes trap into OS when page is referenced

Present Bit



PFN	valid	prot	present
10	1	r-x	1
-	0	-	-
23	1	rw-	0
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
xx 16	1 1	rw- rw-	0 1
4	1	rw-	1

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
- **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 (`dirty` bit set in 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
    - When instruction is being fetched
    - When data is being loaded or stored
  - Requires hardware support
    - precise interrupts: stop CPU pipeline such that instructions before faulting instruction have completed, and those after can be restarted

- **Complexity depends upon instruction set**
  - Can faulting instruction be restarted from beginning?
    - Example: PUSH `[eax]`
      `(stack pointer changes before accessing, if physical address)`
    - Must track side effects so hardware can undo

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)
  - Problems?
- **Hints: Combine above with user-supplied hints about page references**
  - User specifies: may need page in future, don't need this page anymore, or sequential access pattern, ...
  - Example: `madvise()` in Unix (**MADV_RANDOM, MADV_SEQUENTIAL, MADV_WILLNEED, MADV_DONTNEED**)

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
- **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
- **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
    - Does not handle all workloads well

Page Replacement Example

- **Page reference string: ABCABDADBCB**

Three pages of physical memory

Metric: Miss count

5, 7, 5 misses

	OPT				FIFO				LRU			
ABC	A	B	C	3	A	B	C	3	A	B	C	3
A	A	B	C		A	B	C		A	B	C	
B	A	B	C		A	B	C		A	B	C	
D	A	B	D	1	D	B	C	A	A	B	D	1
A	A	B	D		D	A	C	B	A	B	D	
D	A	B	D		D	A	C		A	B	D	
B	A	B	D		D	A	B	C	A	B	D	
C	C	B	D	1	D	C	B	A	C	B	D	1
B	C	B	D		D	C	B		C	B	D	

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!

FIFO Performance may Decrease!

Consider access stream: ABCDABEABCDE

Consider physical memory size: 3 pages vs. 4 pages

How many misses with FIFO?

3 pages: 9 misses
4 pages: 10 misses

Page Replacement Example

- **Page reference string: ABCDABEABCDE**

Three pages of physical memory

Metric: Miss count

5, 7, 5 misses

	3				4				
ABC	A	B	C	3	A	B	C		
D	B	C	D	A	A	B	C	D	4
A	C	D	A	B	A	B	C	D	
B	D	A	B	C	A	B	C	D	
E	A	B	E	D	B	C	D	E	A
A	A	B	E		C	D	E	A	B
B	A	B	E		D	E	A	B	C
C	B	E	C	A	E	A	B	C	D
D	E	C	D	B	A	B	C	D	E
E	E	C	D		B	C	D	E	A

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

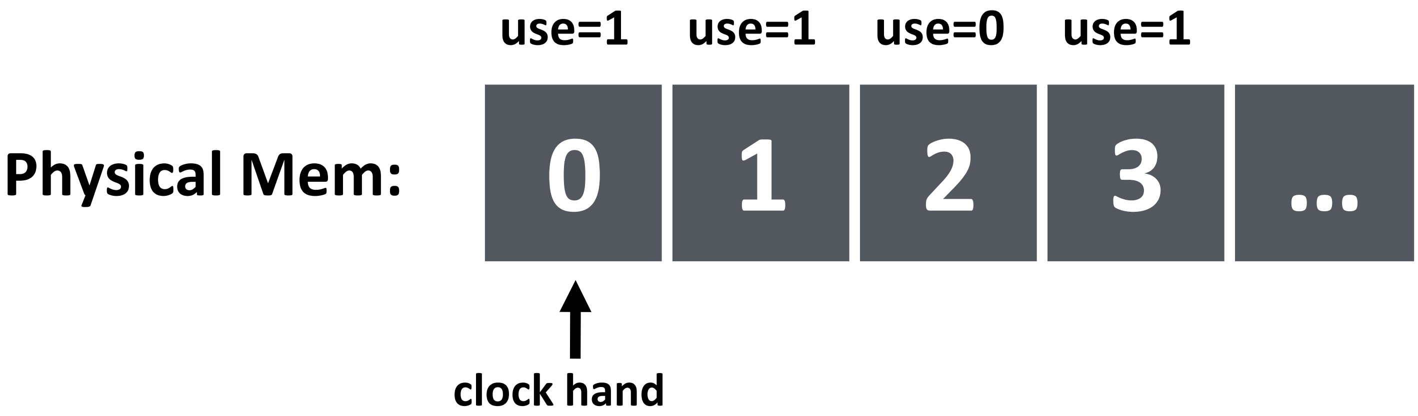
Implementing LRU

- **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
- **Hardware Perfect LRU**
  - Associate timestamp with each page
  - When page is referenced: Store system clock
  - When need victim: Scan through pages to find oldest clock
  - Trade-off: Fast on memory reference, slow on replacement (especially as size of memory grows)
- **Practical LRU implementation (in-memory kv)**
  - A page array with fields reserved for LRU double-linked list
- **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

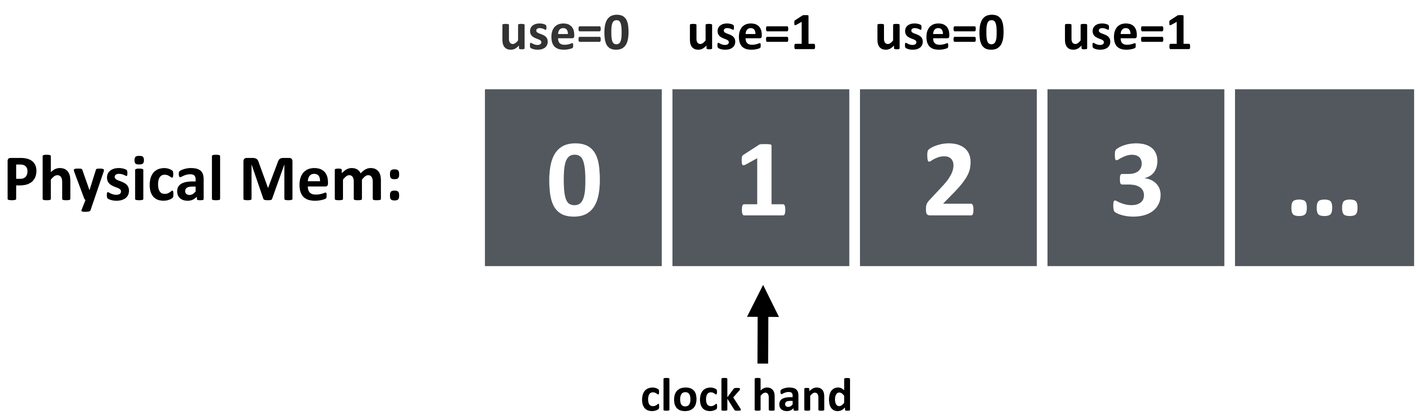
Clock Algorithm

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

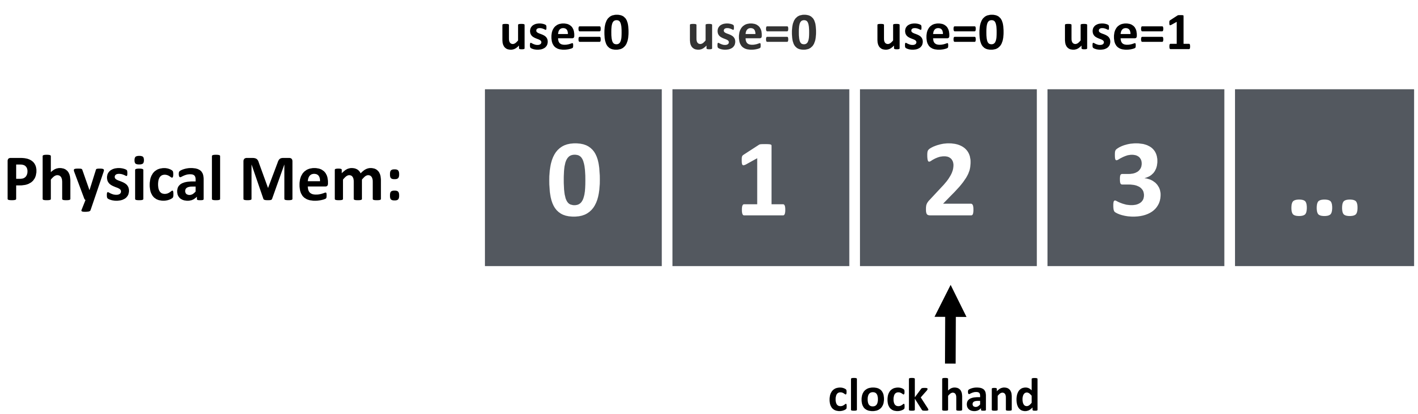
Clock: Look For a Page



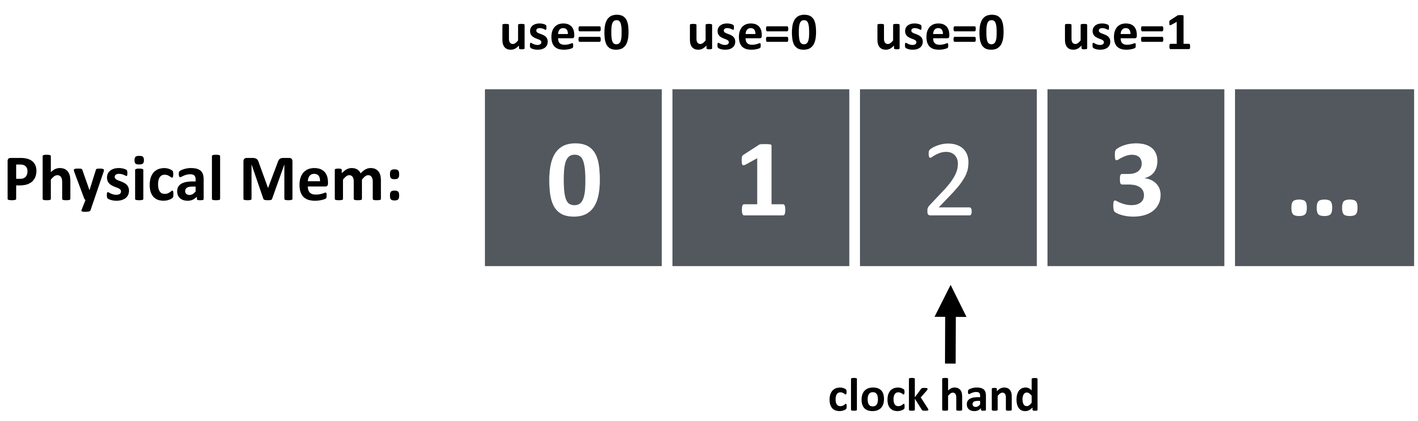
Clock: Look For a Page



Clock: Look For a Page



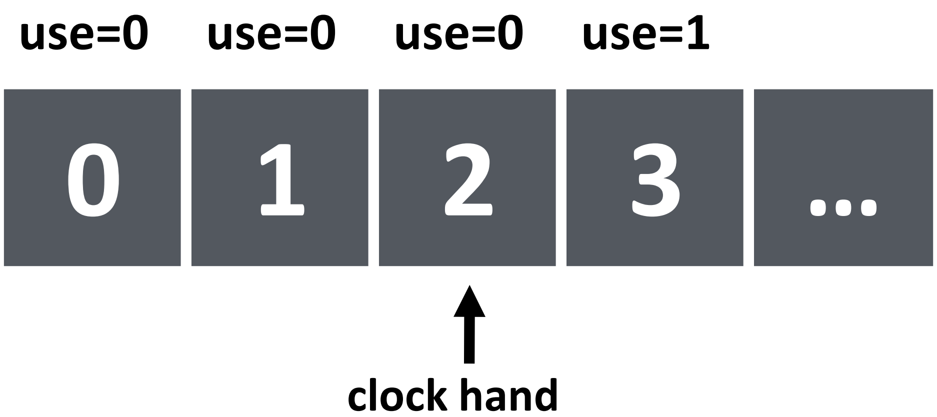
Clock: Look For a Page



evict page 2 because it has not been recently used

Clock: Look For a Page

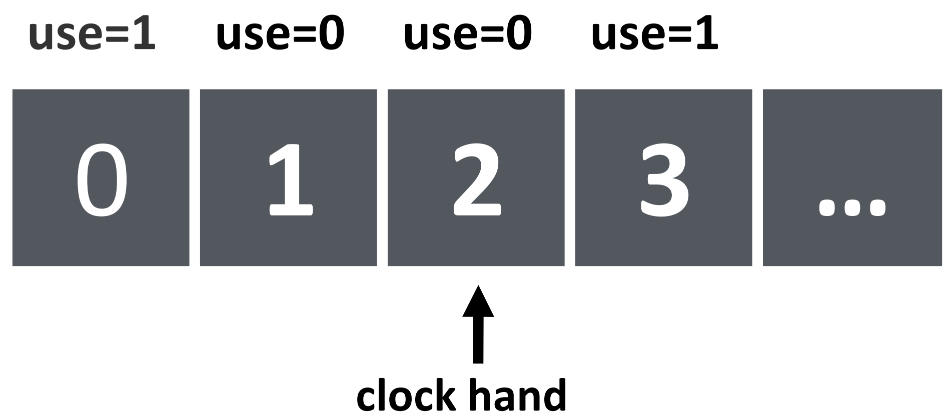
Physical Mem:



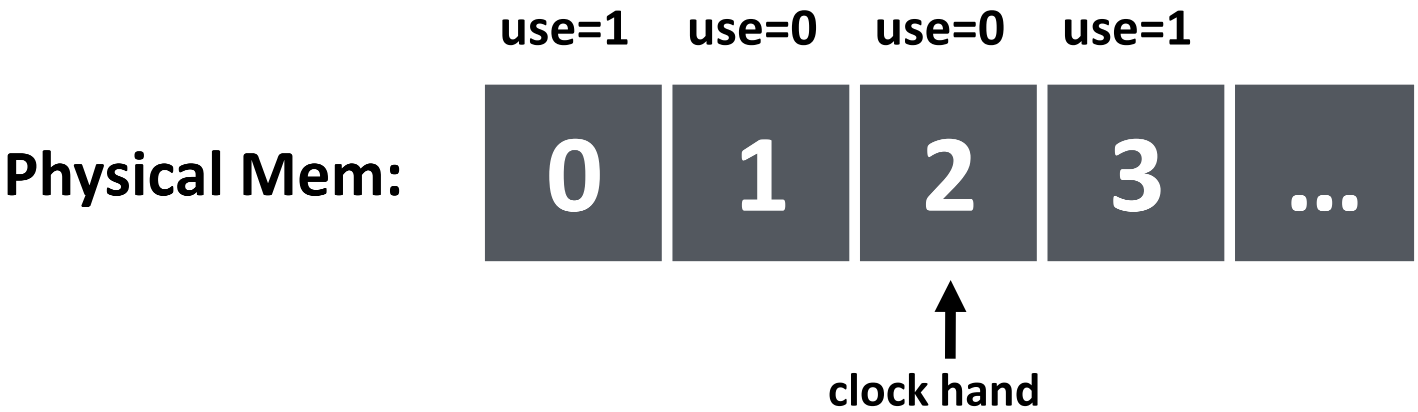
page 0 is accessed...

Clock: Look For a Page

Physical Mem:



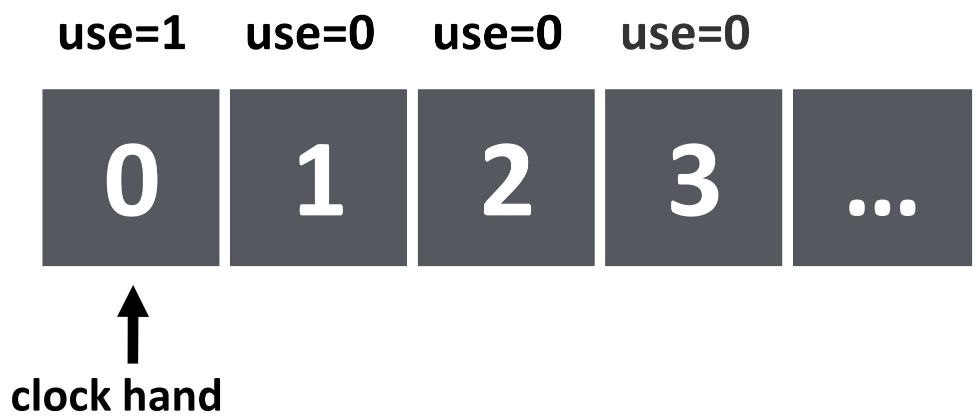
Clock: Look For a Page



Clock: Look For a Page

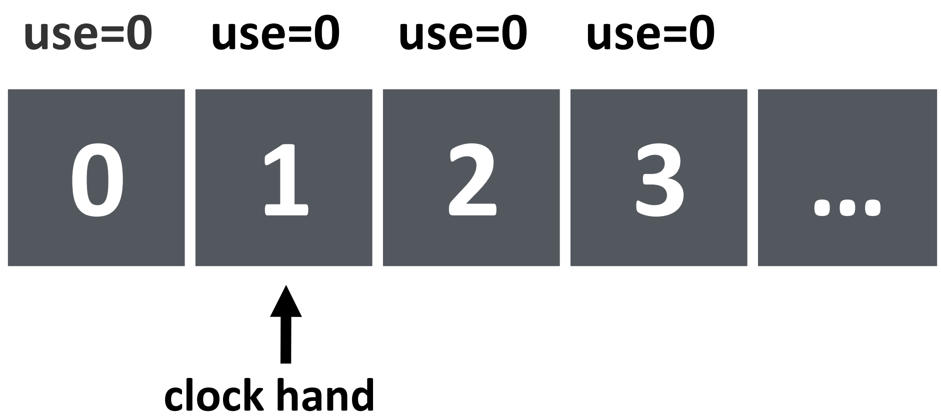
Clock: Look For a Page

Physical Mem:

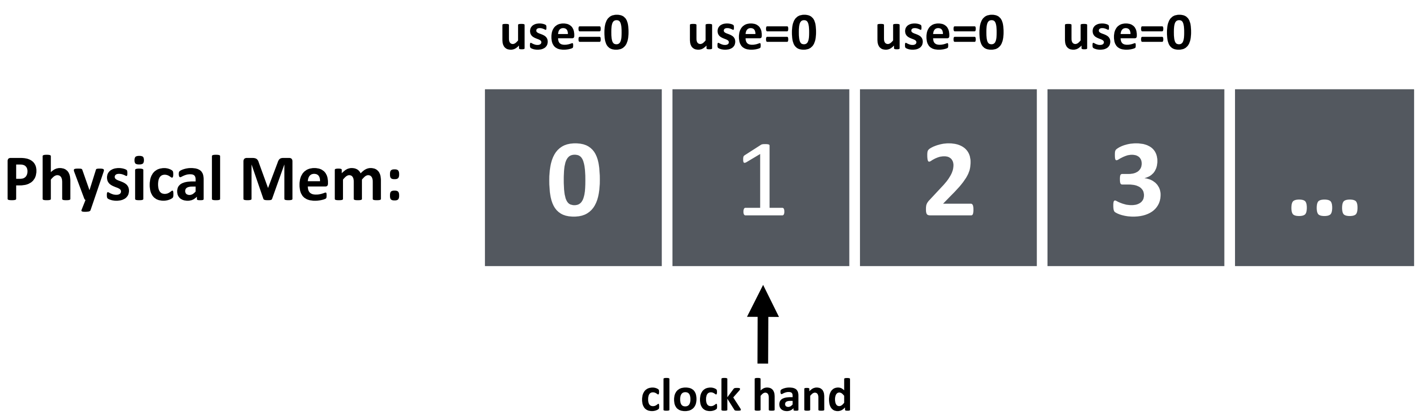


Clock: Look For a Page

Physical Mem:



Clock: Look For a Page



evict page 1 because it has not been recently used

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
    (not visited for more than k clock rounds)
- **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

What if no Hardware Support?

- **What can the OS do if hardware does not have `use` bit (or `dirty` bit)?**
  - Can the OS “emulate” these bits?
  - Yes
- **Leading question:**
  - How can the OS get control every time `use` bit should be set?
  - i.e., generate a trap when a page is accessed?
- **implement `use` bit with page fault:**
  - Scan page table for pages with present bit = 1
  - Clear the bit to 0 so that OS can get a page fault when accessed
  - OS use another data structure to know whether the page is in memory

Conclusions

- **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**