Virtual Memory & Caching

(Chapter 12-17)

CS 4410
Operating Systems

Cornell University
Founded A.D. 1865
Cornell CIS
Computing and Information Science

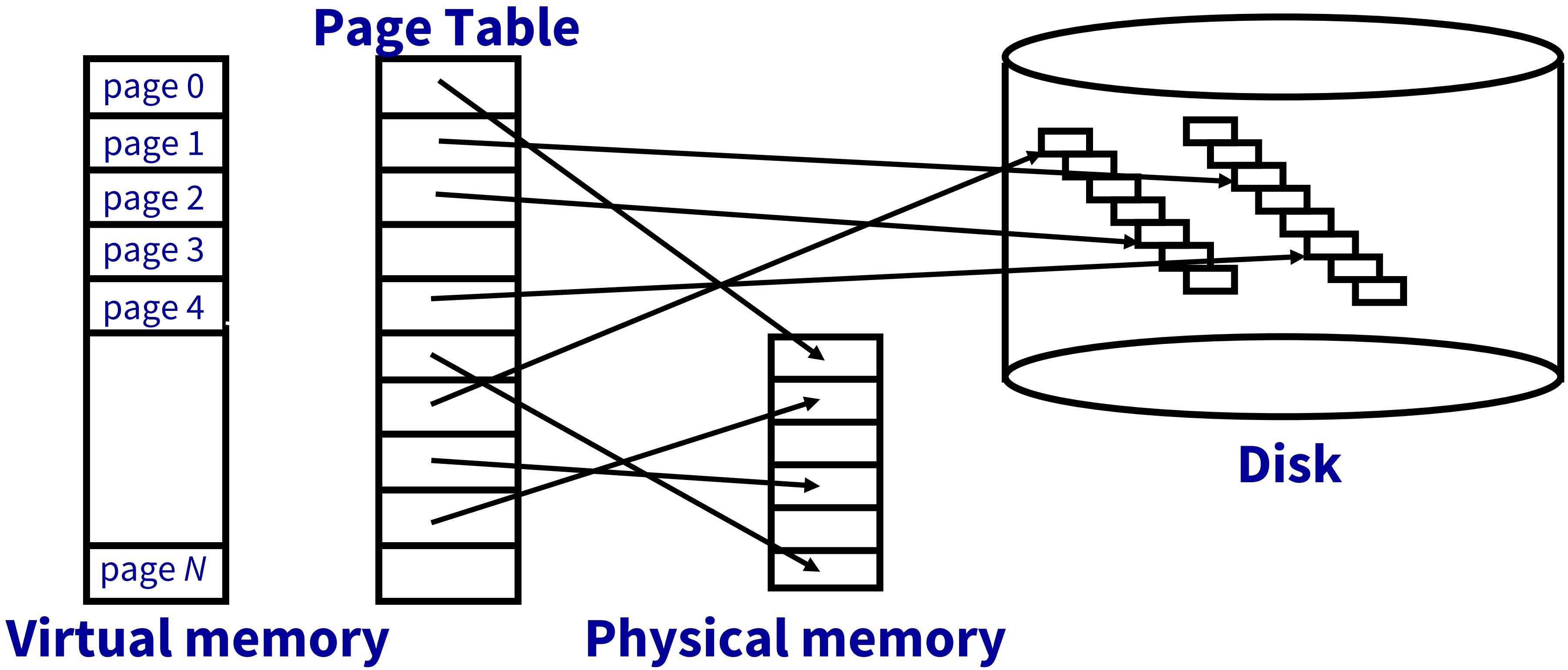
Last Time: Address Translation

- Paged Translation
- Efficient Address Translation
  - Multi-Level Page Tables
  - ~~Inverted Page Tables~~
  - TLBs

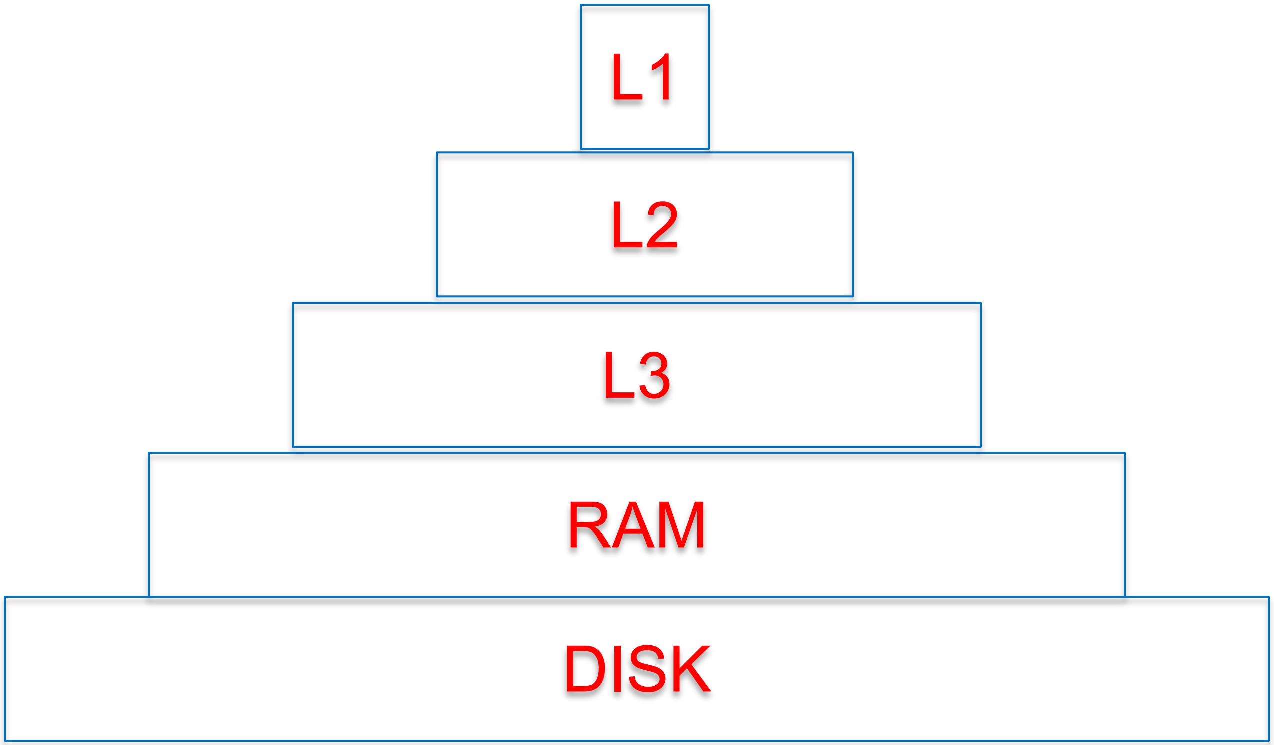
This time: **Virtual Memory & Caching**

What is Virtual Memory?

- Each process has illusion of large address space
  - 2^x bytes for x-bit addressing
- However, physical memory is usually much smaller
- How do we give this illusion to multiple processes?
  - Virtual Memory: some addresses reside in disk



Process executes from disk!



RAM is really just another layer of cache

Swapping vs. Paging

Swapping

- Loads entire process in memory
- "Swap in" (from disk) or "Swap out" (to disk) a process
- Slow (for large processes)
- Wasteful (might not require everything)
- Does not support sharing of code segments
- Virtual memory limited by size of physical memory

Paging

- Runs all processes concurrently
- A few pages from each process live in memory
- Finer granularity, higher performance
- Large virtual mem supported by small physical mem
- Certain pages (read-only ones, for example) can be shared among processes

(the contents of) **A Virtual Page Can Be**

Mapped

- to a physical frame

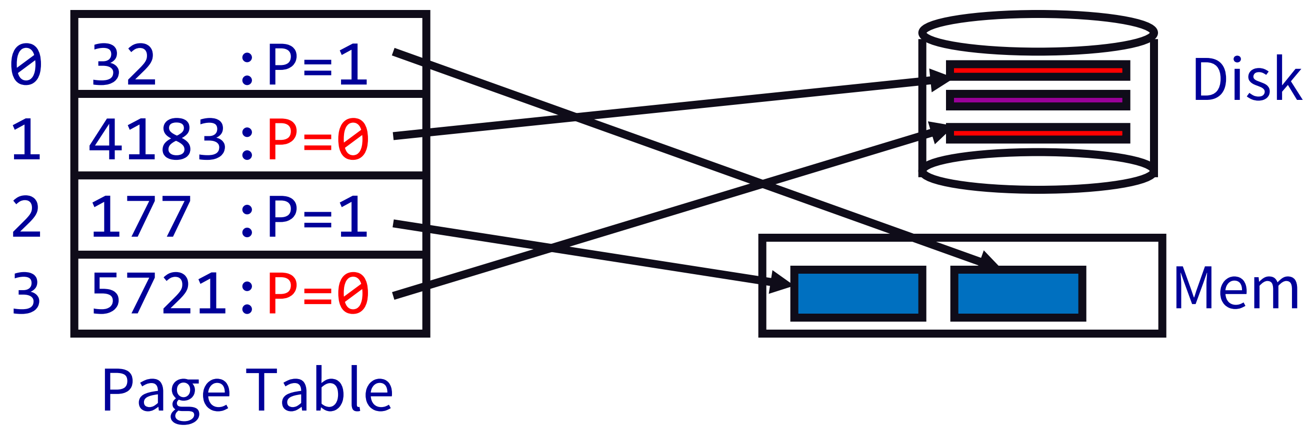
Not Mapped (→ Page Fault)

- in a physical frame, but not currently mapped
- or still in the original program file
- or zero-filled (heap/BSS, stack)
- or on backing store (“paged or swapped out”)
- or illegal: not part of a segment
  → Segmentation Fault

Supporting Virtual Memory

Modify Page Tables with a *present* bit

- Page in memory → *present = 1*
- Page not in memory → PT lookup triggers **page fault**



Handling a Page Fault

Identify page and reason (r/w/x)

- access inconsistent w/ segment access rights

  → terminate process
- access a page that is kept on disk:

  → does frame with the code/data already exist?

  No? Allocate a frame & bring page in (next slide)
- access of zero-initialized data (BSS) or stack
  - Allocate a frame, fill frame with zero bytes
- ~~access of COW page~~
  - ~~Allocate a frame and copy~~

When a page needs to be brought in…

- Find a free frame
  - evict one if there are no free frames
- Issue disk request to fetch data for page
- Block current process
- Context switch to another process
- When disk request completes, update PTE
  - frame number, present bit, RWX bits
- Put current process in ready queue

When a frame needs to be swapped out…

- Find all page table entries that refer to the frame
  - Frame might be shared
  - Maintain a *Core Map* (frames → pages)
- Set each page table entry to not present
- Remove any TLB entries
  - "TLB Shootdown"
- Write changes on page back to disk, if needed
  - Dirty/Modified bit in PTE indicates need
  - Text segments are (still) on program image on disk

Updated Context Switch

- Save current process' registers in PCB
- ***Flush TLB*** *(unless TLB is tagged)*
- Restore registers and PTBR of next process to run
- "Return from Interrupt"

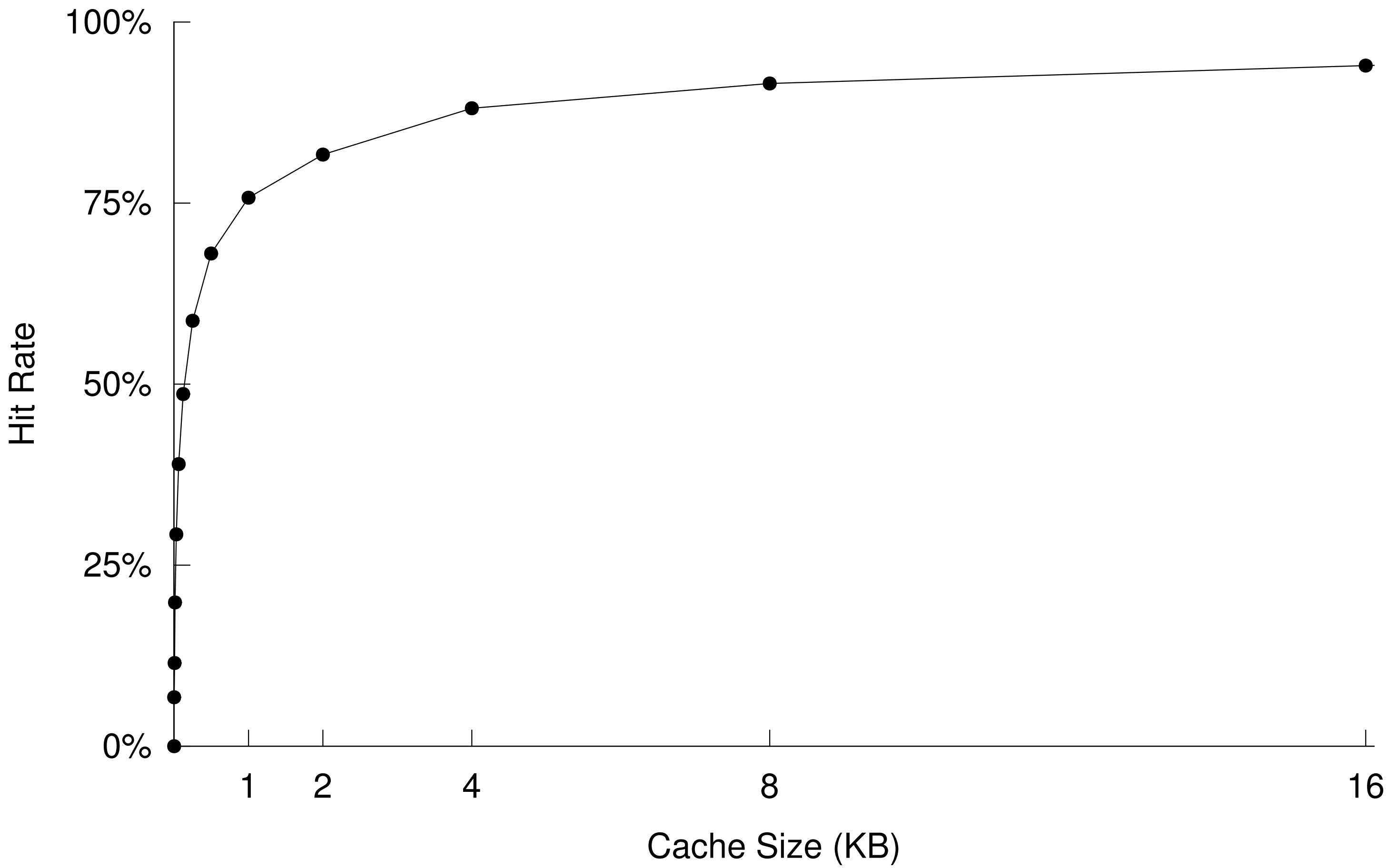
Memory Hierarchy

Cache	Hit Cost	Size
1st level cache / 1st level TLB	1 ns	64 KB
2nd level cache / 2nd level TLB	4 ns	256 KB
3rd level cache	12 ns	2 MB
Memory (DRAM)	100 ns	10 GB
Data center memory (DRAM)	100 μs	100 TB
Local non-volatile memory	100 μs	100 GB
Local disk	10 ms	1 TB
Data center disk	10 ms	100 PB
Remote data center disk	200 ms	1 XB

Every layer is a cache for the layer below it.

Working Set

1. Collection of a process' most recently used pages
   (The Working Set Model for Program Behavior, Denning,'68)
2. Pages referenced by process in last Δ time-units



Thrashing

Excessive rate of paging
Cache lines evicted before they can be reused

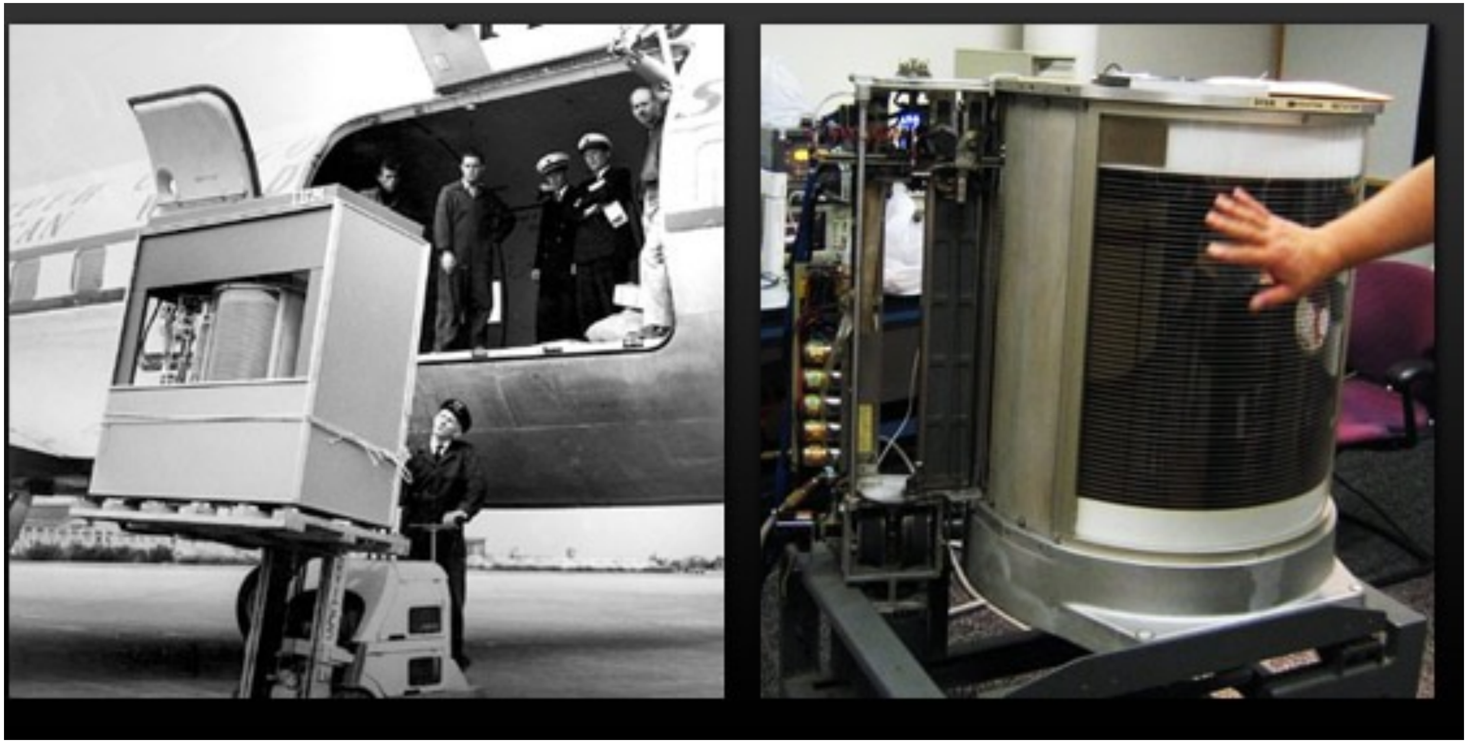
Causes:

- Too many processes in the system
- Cache not big enough to fit working set
- Bad luck (conflicts)
- Bad eviction policies (later)

Prevention:

- Restructure code to reduce working set
- Increase cache size
- Improve caching policies

Why “thrashing”?



The first hard disk drive—the IBM Model 350 Disk File (came w/IBM 305 RAMAC, 1956).

Total storage = 5 million characters (just under 5 MB).

http://royal.pingdom.com/2008/04/08/the-history-of-computer-data-storage-in-pictures/

“Thrash” dates from the 1960’s, when disk drives were as large as washing machines. If a program’s working set did not fit in memory, the system would need to shuffle memory pages back and forth to disk. This burst of activity would violently shake the disk drive.

Caching

- Assignment: where do you put the data?
- **Replacement: whom do you kick out?**

What do you do when memory is full?

Page Replacement Algorithms

- **Random:** Pick any page to eject at random
  - Used mainly for comparison
- **FIFO:** The page brought in earliest is evicted
  - Ignores usage
- **OPT:** Belady's algorithm
  - Select page not used for longest time
- **LRU:** Evict page that hasn't been used for the longest
  - Assumes past is a good predictor of the future
- **MRU:** Evict the most recently used page
- **LFU:** Evict least frequently used page
- And many approximation algorithms

Expectation

- more frames (i.e., larger cache) →
  not more misses

First-In-First-Out (FIFO) Algorithm

- *Reference string*: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- **3 frames** (3 pages in memory at a time per process):

frames

reference

1			**1**
2		**2**	1
3	**3**	2	1
4	3	2	**4**
1	3	**1**	4
2	**2**	1	4
5	2	1	**5**
1	2	1	5
2	2	1	5
3	2	**3**	5
4	**4**	3	5
5	4	3	5

← contents of frames after reference

page fault (miss)

hit

9 page faults

First-In-First-Out (FIFO) Algorithm

- *Reference string*: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- **4 frames** (4 pages in memory at a time per process):

frames

reference

1				**1**
2			**2**	1
3		**3**	2	1
4	**4**	3	2	1
1	4	3	2	1
2	4	3	2	1
5	4	3	2	**5**
1	4	3	**1**	5
2	4	**2**	1	5
3	**3**	2	1	5
4	3	2	3	**4**
5	3	2	**5**	4

← contents of frames after reference

page fault

hit

10 page faults

more frames → more page faults?

Belady's Anomaly

Optimal Algorithm (OPT)

- Replace frame that will not be used for the longest
- 4 frames example

1				**1**
2			**2**	1
3		**3**	2	1
4	**4**	3	2	1
1	4	3	2	1
2	4	3	2	1
5	**5**	3	2	1
1	5	3	2	1
2	5	3	2	1
3	5	3	2	1
4	5	3	2	**4**
5	5	3	2	4

6 page faults

Question: How do we tell the future?
Answer: We can't

OPT used as upper-bound in measuring how well your algorithm performs

OPT Approximation

In real life, we do not have access to the future page request stream of a program

→ Need to make a guess at which pages will not be used for the longest time

Least Recently Used (LRU) Algorithm

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

1				**1**
2			**2**	1
3		**3**	2	1
4	**4**	3	2	1
1	4	3	2	1
2	4	3	2	1
5	4	**5**	2	1
1	4	5	2	1
2	4	5	2	1
3	**3**	5	2	1
4	3	**4**	2	1
5	3	4	2	**5**

page fault

hit

8 page faults

Implementing LRU

- On reference: Timestamp each page
- On eviction: Scan for oldest page

Problems:

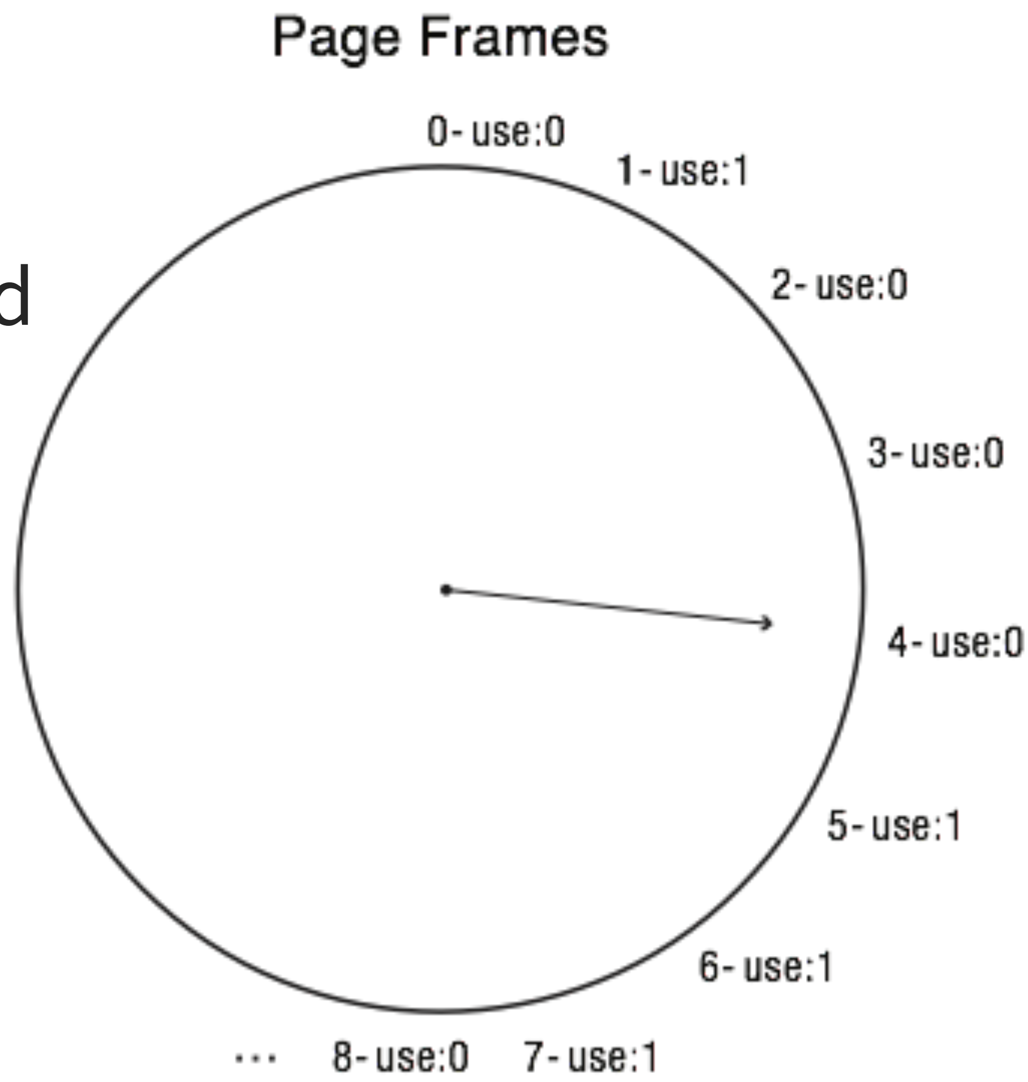
- Large page lists
- Timestamps are costly

Solution: **approximate LRU**

- Note: LRU is already an approximation
- Exploit *use* (REF) bit in PTE

Clock Algorithm

- To allocate a frame, inspect the *use* bit in the PTE at clock hand and advance clock hand
- Used? Clear *use* bit and repeat



Working Set Algorithm (WS)

- Maintain for each frame the approximate time the frame was last used
- At each clock tick
  - Update this time to the current time for all frames that were referenced since the last clock tick
    - i.e., the ones with *use* (REF) bits set
  - Clear all *use* bits
  - Put all frames that have not been used for some time Δ (working set parameter) on the free list
- When a frame is needed, use free list
  - If empty, pick any frame

Note: requires scan of all frames at each clock tick

Other Algorithms

MRU: Remove the most recently touched page

- Good for data accessed only once, *e.g.* a movie file

LFU: Remove page with lowest usage count

- Like CLOCK but use multiple bits. Shift right by 1 at regular intervals

MFU: remove the most frequently used page

Local versus Global Replacement

- So far, we have tacitly assumed that all frames are shared by all processes
  - This is called “global replacement”
- But is it fair?
  - Badly behaved processes can ruin the experience of processes with good locality
- Local replacement: divided the frames up evenly between the processes
  - Can lead to under-utilization