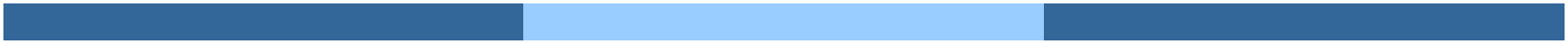
Virtual Memory and Demand Paging



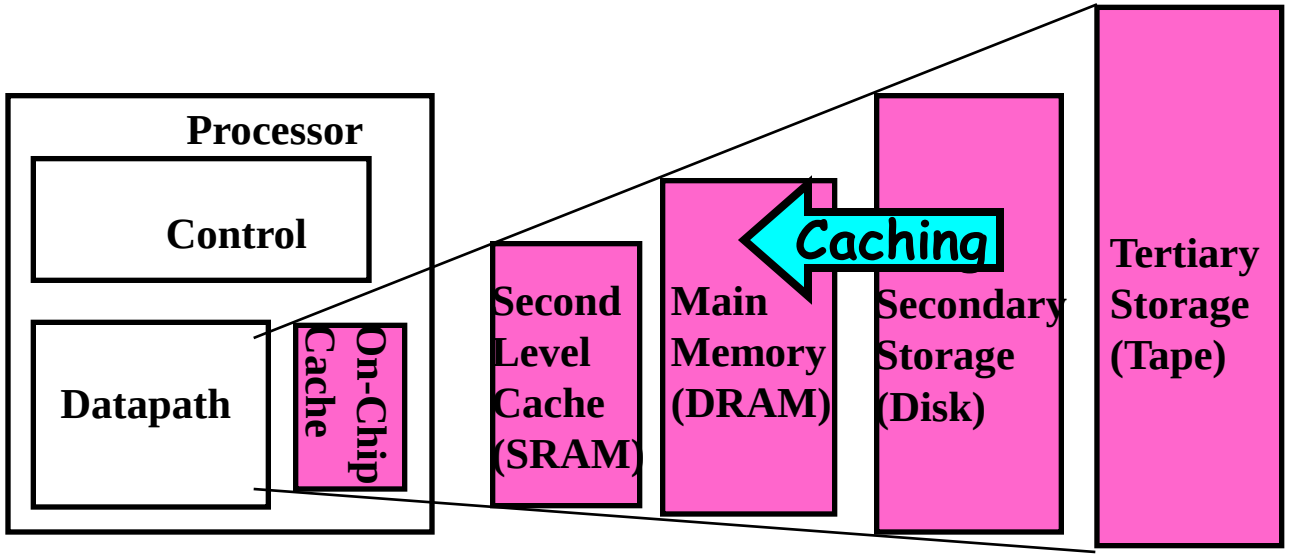
CS170 Fall 2015. T. Yang
Some slides from John Kubiatowicz's cs162 at UC Berkeley

What to Learn?

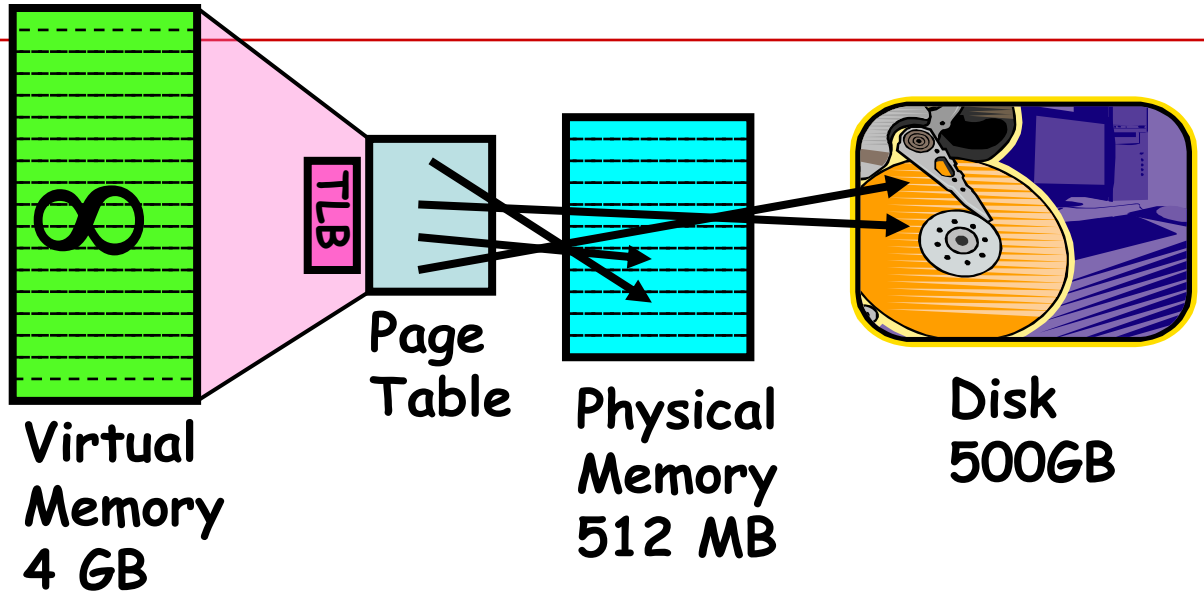
- **Chapter 9 in the text book**
- **The benefits of a virtual memory system**
- **The concepts of**
  - demand paging
  - page-replacement algorithms
  - and allocation of physical page frames
- **Other related techniques**
  - Memory mapped files

Demand Paging

- **Modern programs require a lot of physical memory**
  - Memory per system growing faster than 25%-30%/year
- **But they don't use all their memory all of the time**
  - 90-10 rule: programs spend 90% of their time in 10% of their code
  - Wasteful to require all of user's code to be in memory
- **Solution: use main memory as cache for disk**

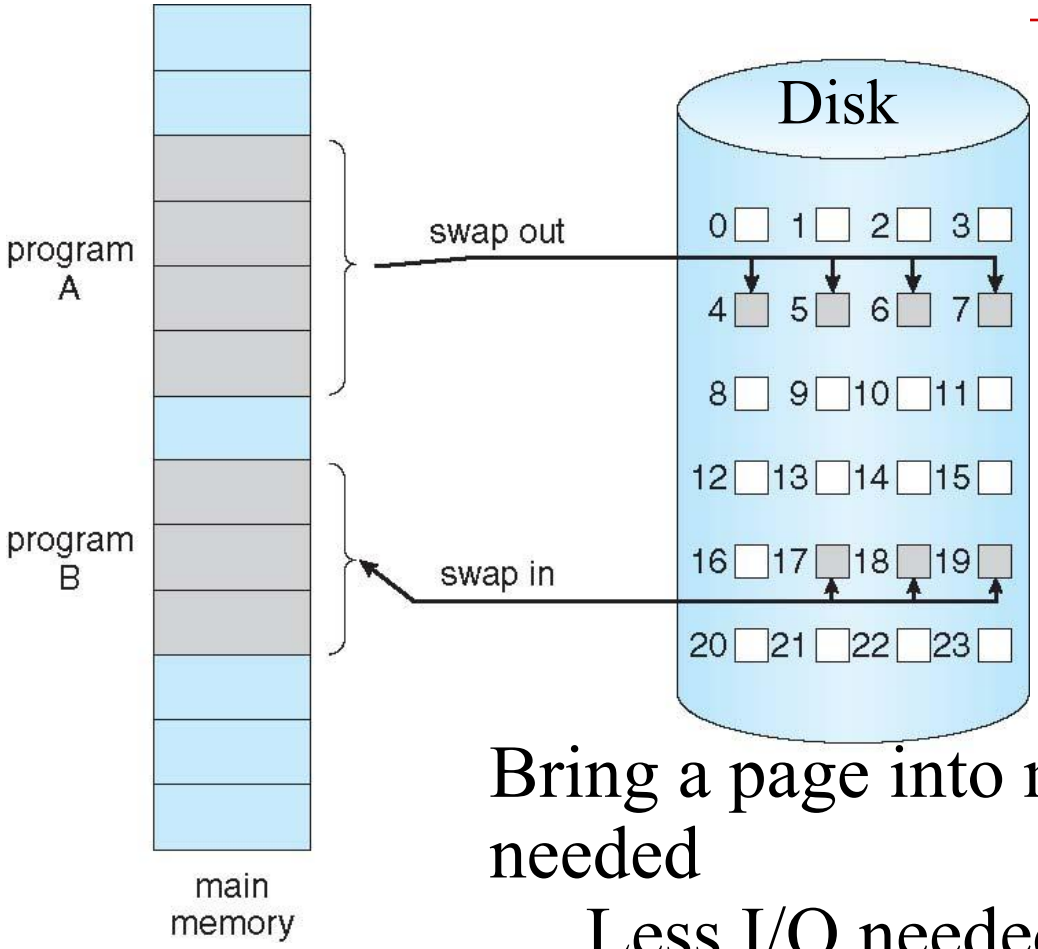


Illusion of Infinite Memory



- **Virtual memory can be much larger than physical memory ⇒**
  - Combined memory of running processes much larger than physical memory
    - More programs fit into memory, allowing more concurrency
- **Principle:**
  - Supports flexible placement of physical data
    - Data could be on disk or somewhere across network
  - Variable location of data transparent to user program
    - Performance issue, not correctness issue

Memory as a program cache



Bring a page into memory ONLY when it is needed

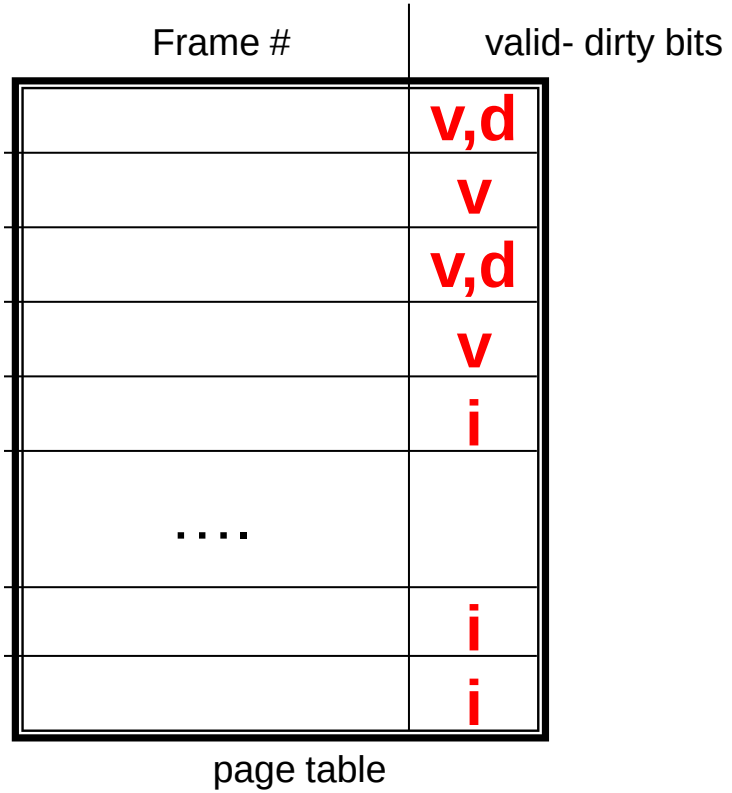
- Less I/O needed
- Less memory needed
- Faster response
- More users supported

Valid/dirty bits in a page table entry

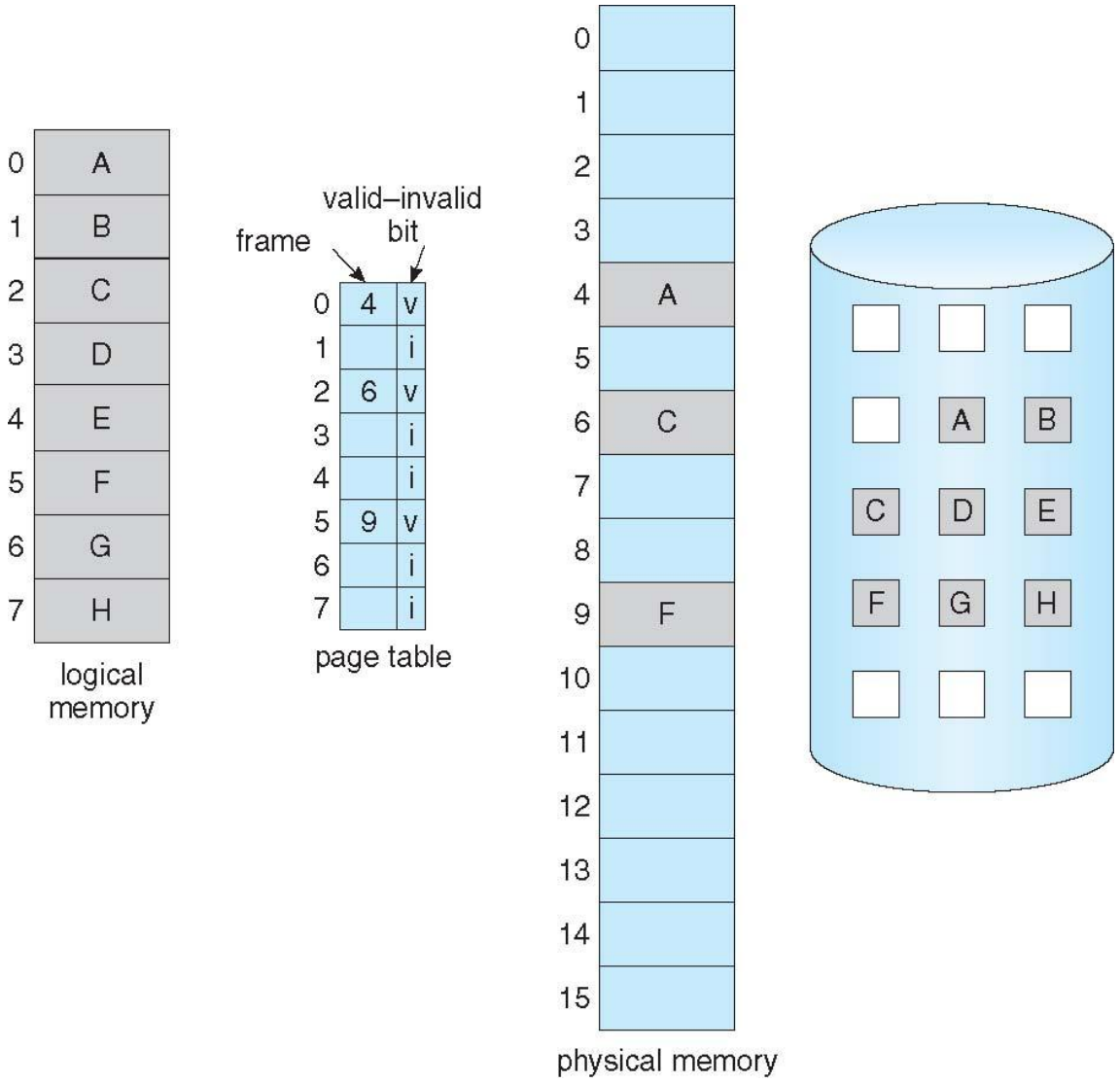
- **With each page table entry a valid–invalid bit is associated**
  (v ⇒ in-memory, i ⇒ not-in-memory)
- **Initially valid–invalid bit is set to i on all entries**
- **Not in memory ⇒ page fault**
- **Dirty bit**
  - Dirty means this page

  has been modified.

  It needs to be written back to disk



Example of Page Table Entries When Some Pages Are Not in Main Memory

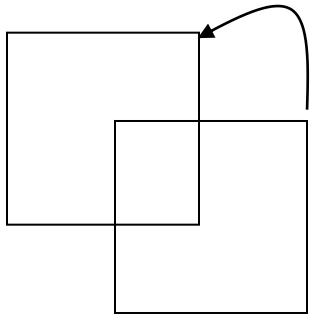


What does OS do on a Page Fault?

- Choose an old page to replace
- If old page modified (“Dirty=1”), write contents back to disk
- Change its PTE and any cached TLB to be invalid
- Get an empty physical page
- Load new page into memory from disk
- Update page table entry, invalidate TLB for new entry
- Continue thread from original faulting location
  - Restart the instruction that caused the page fault

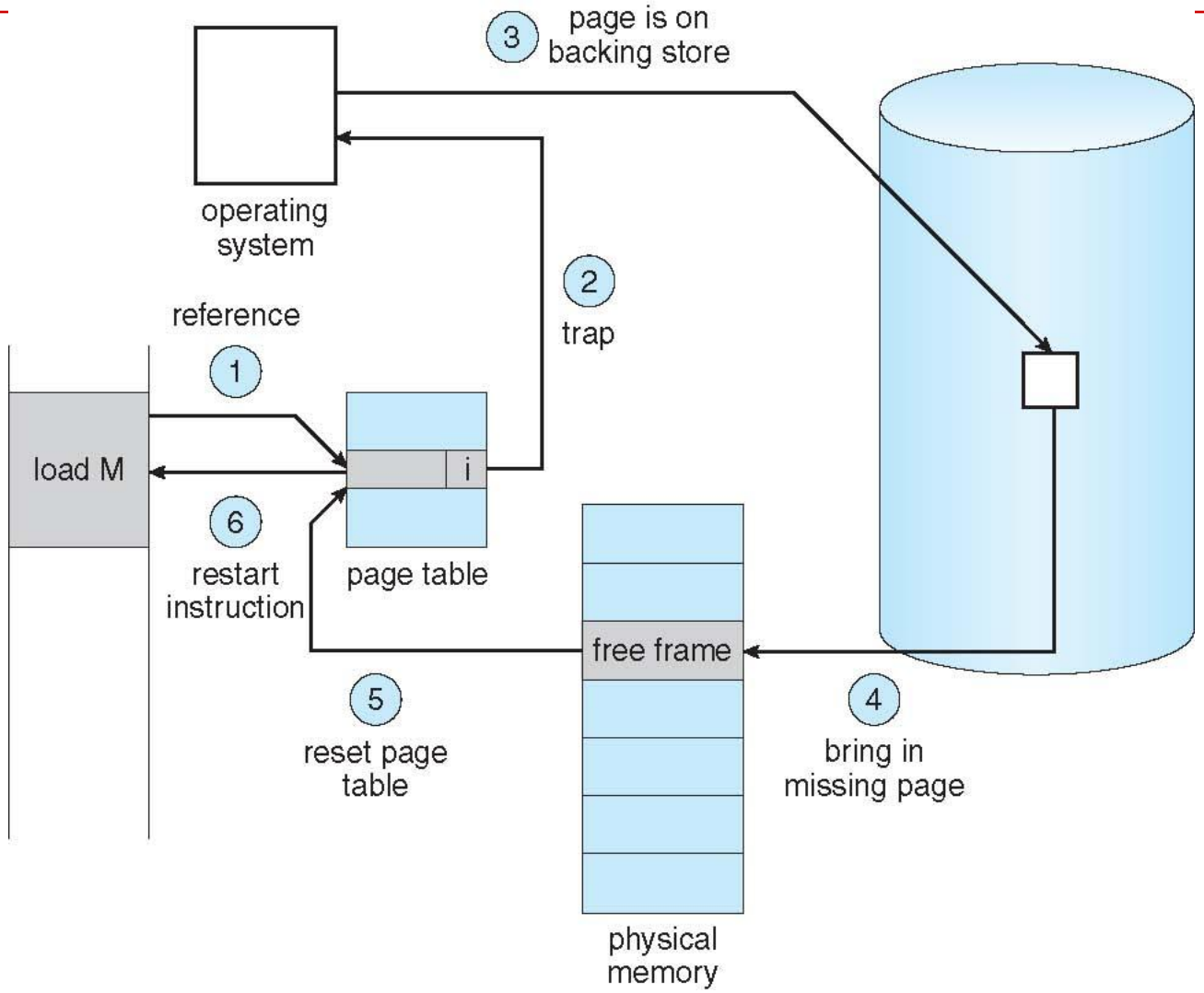
Restart the instruction that caused the page fault

- **Restart instruction if there was no side effect from last execution**
- **Special handling**
  - block move

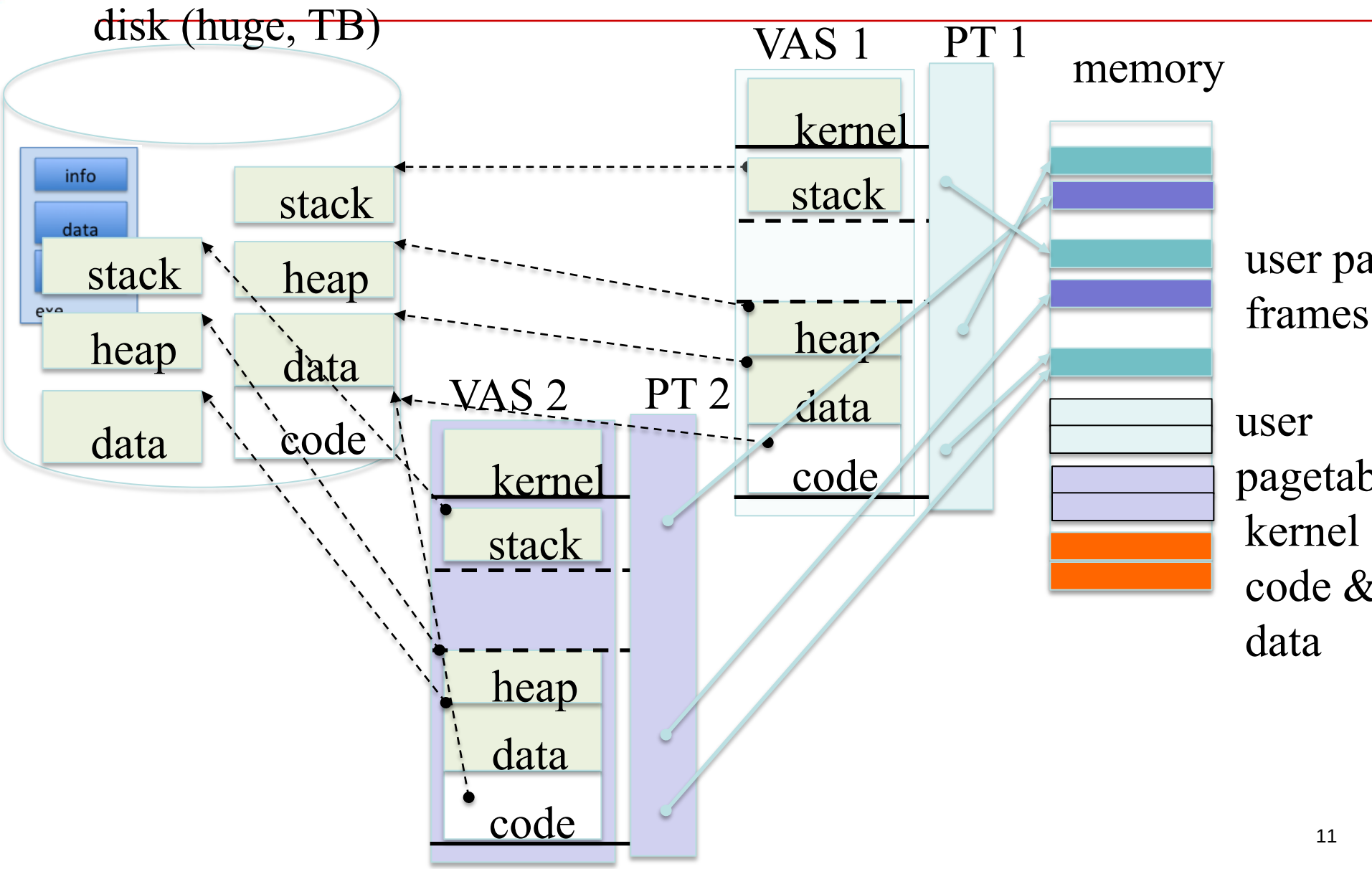


  - Auto increment/decrement

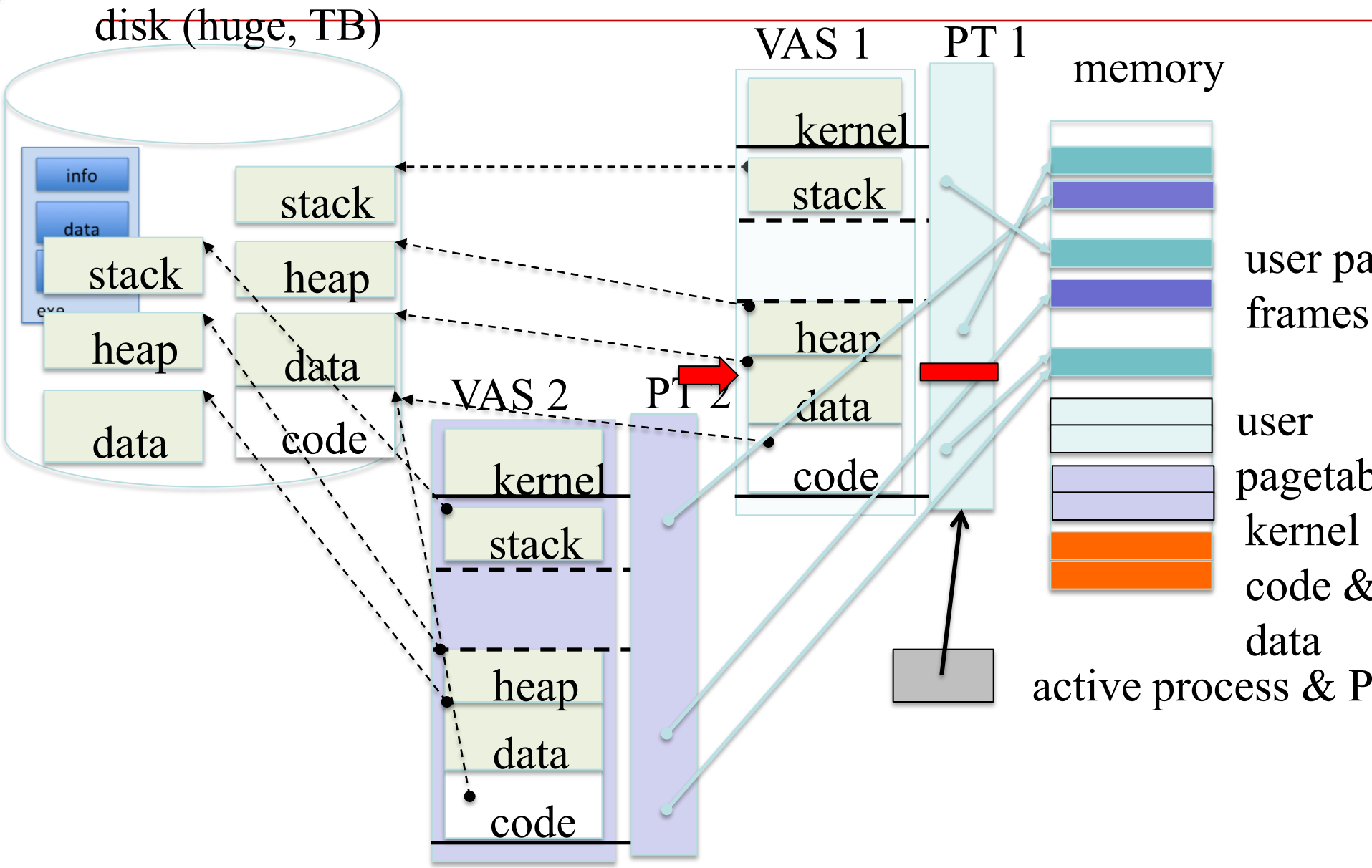
Steps in Handling a Page Fault



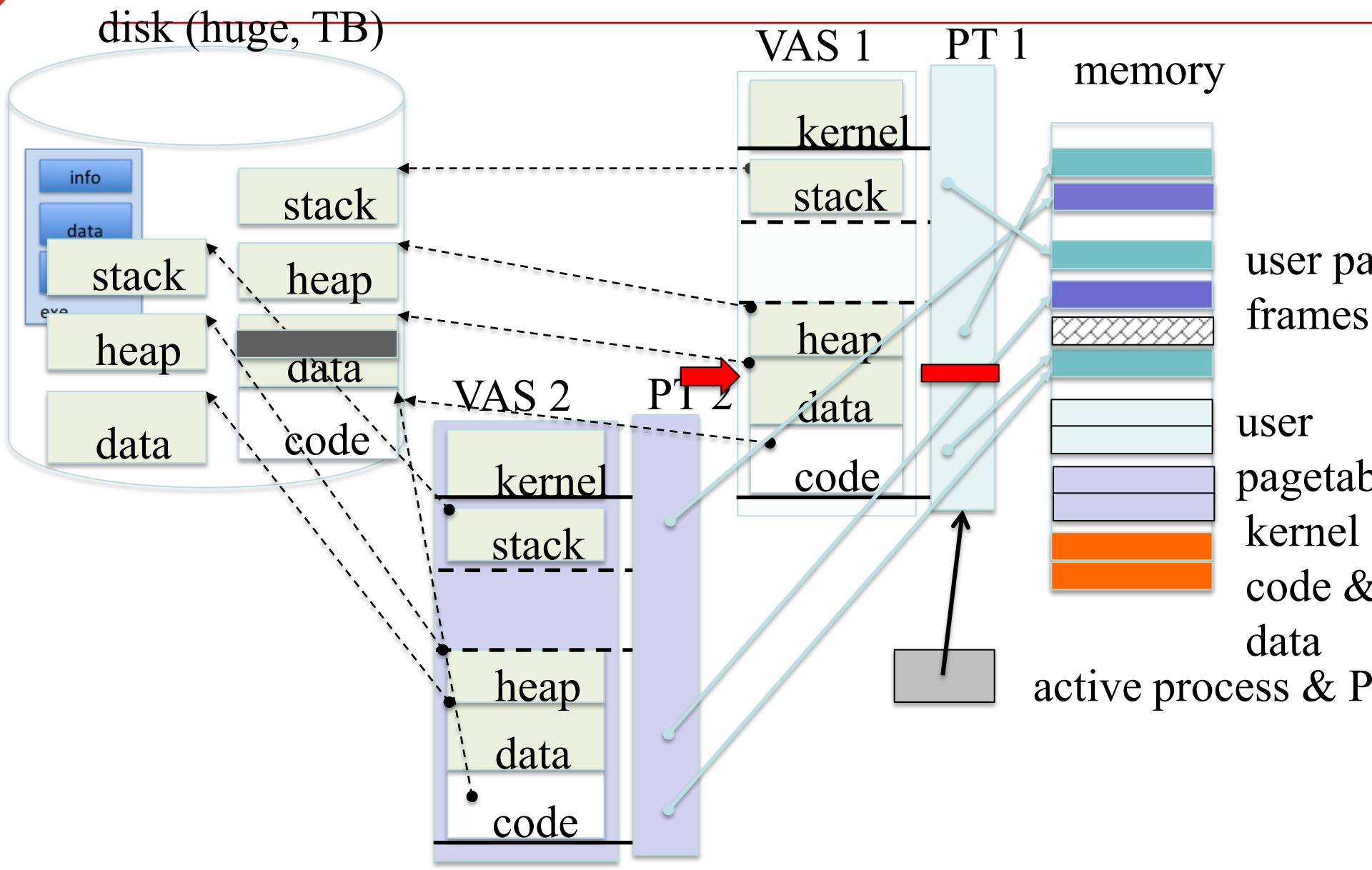
Provide Backing Store for VAS



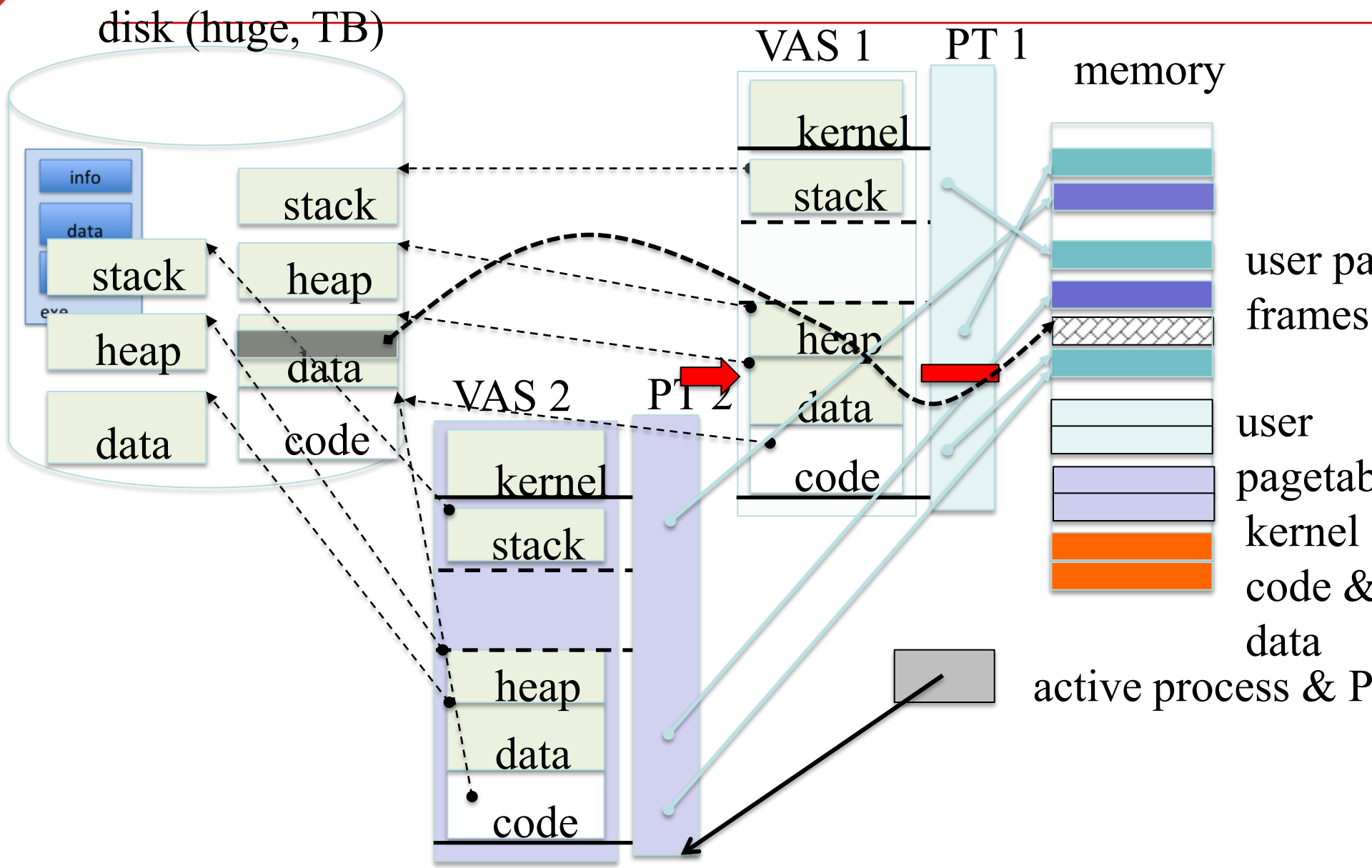
On page Fault …



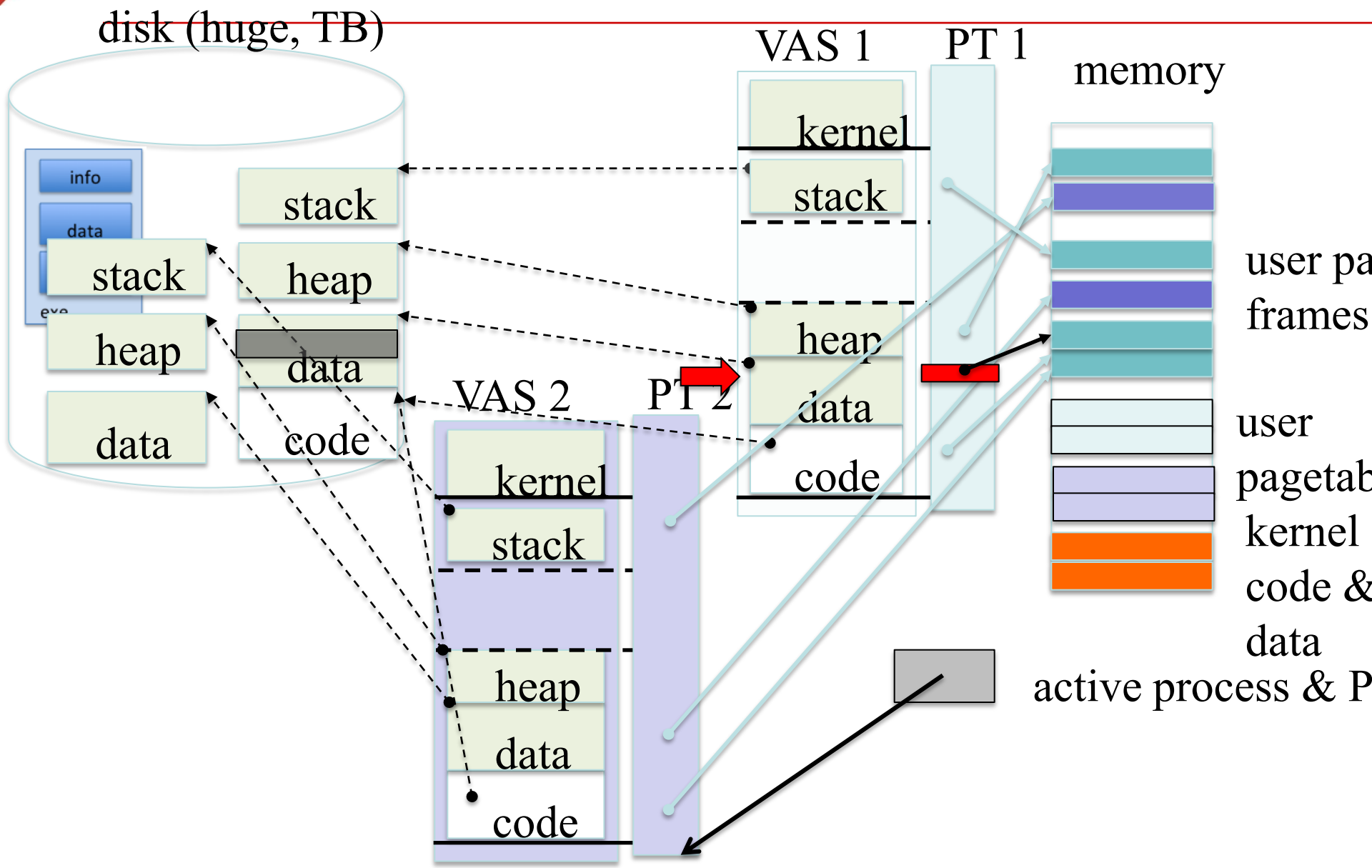
On page Fault … find & start load



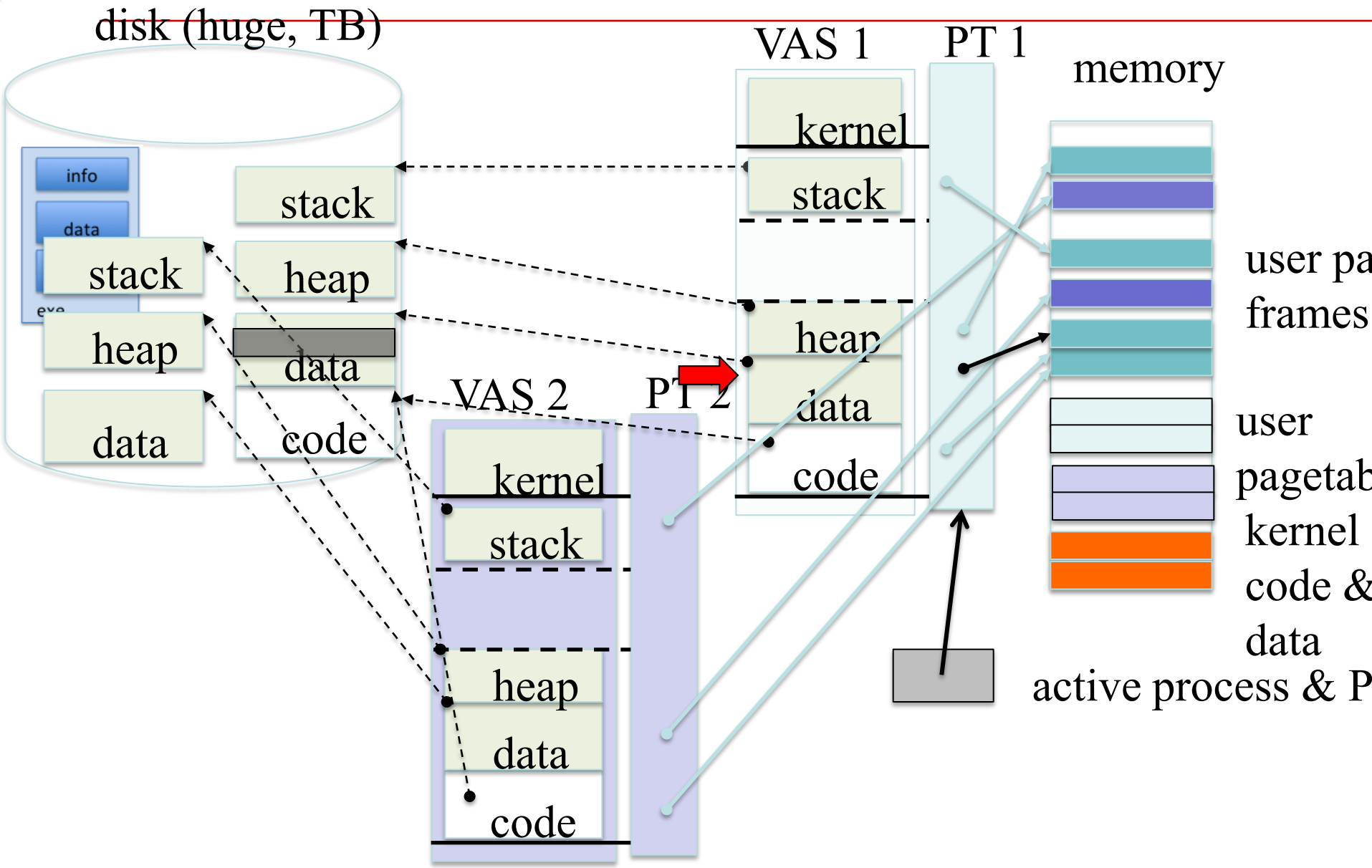
On page Fault … schedule other P or T



On page Fault … update PTE



Eventually reschedule faulting thread



Performance of Demand Paging

- ***p* : page fault rate**
- **$0 \leq p \leq 1.0$**
  - if $p = 0$, no page faults
  - if $p = 1$, every reference is a fault
- **Effective Access Time (EAT)**

$$\text{EAT} = (1 - p) \times \text{memory access}$$
$$+ p \text{ (page fault overhead}$$
$$+ \text{swap page out}$$
$$+ \text{swap page in}$$
$$+ \text{restart overhead)}$$

Demand Paging Performance Example

- **Memory access time = 200 nanoseconds**
- **Average page-fault service time = 8 milliseconds**

- EAT = (1 – p) x 200 + p (8 milliseconds)
  = (1 – p) x 200 + p x 8,000,000
  = 200 + p x 7,999,800
- If one access out of 1,000 causes a page fault, then
  EAT = 8.2 microseconds.
  This is a slowdown by a factor of 40!
- **What if want slowdown by less than 10%?**
  - EAT < 200ns x 1.1 $\Rightarrow$ p < 2.5 x 10^{-6}
  - This is about 1 page fault in 400000!

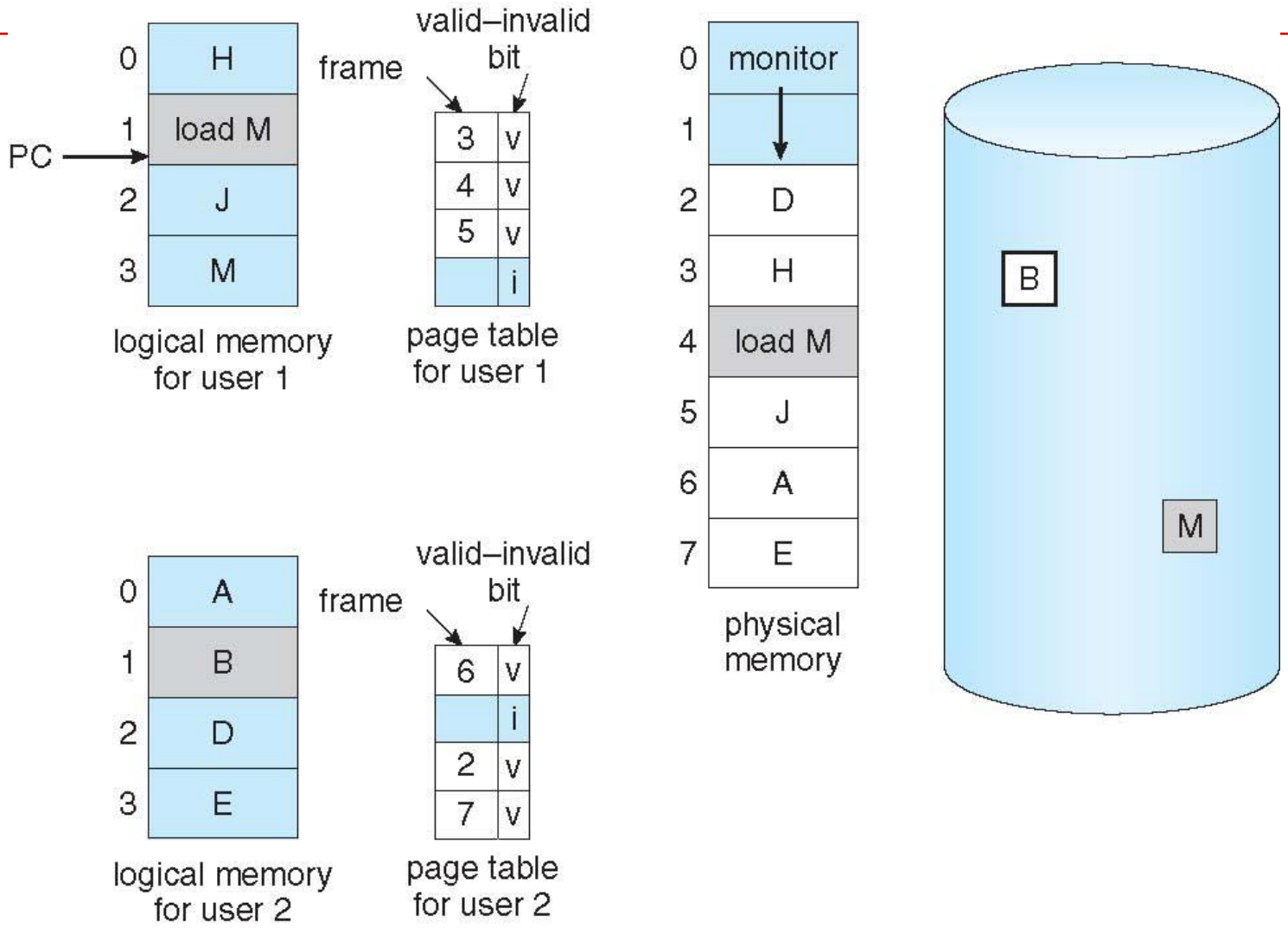
What Factors Lead to Misses?

- **Compulsory Misses:**
  - Pages that have never been paged into memory before
  - How might we remove these misses?
    - Prefetching: loading them into memory before needed
    - Need to predict future somehow! More later.
- **Capacity Misses:**
  - Not enough memory. Must somehow increase size.
  - Can we do this?
    - One option: Increase amount of DRAM (not quick fix!)
    - Another option: If multiple processes in memory: adjust percentage of memory allocated to each one!
- **Policy Misses:**
  - Caused when pages were in memory, but kicked out prematurely because of the replacement policy
  - How to fix? Better replacement policy

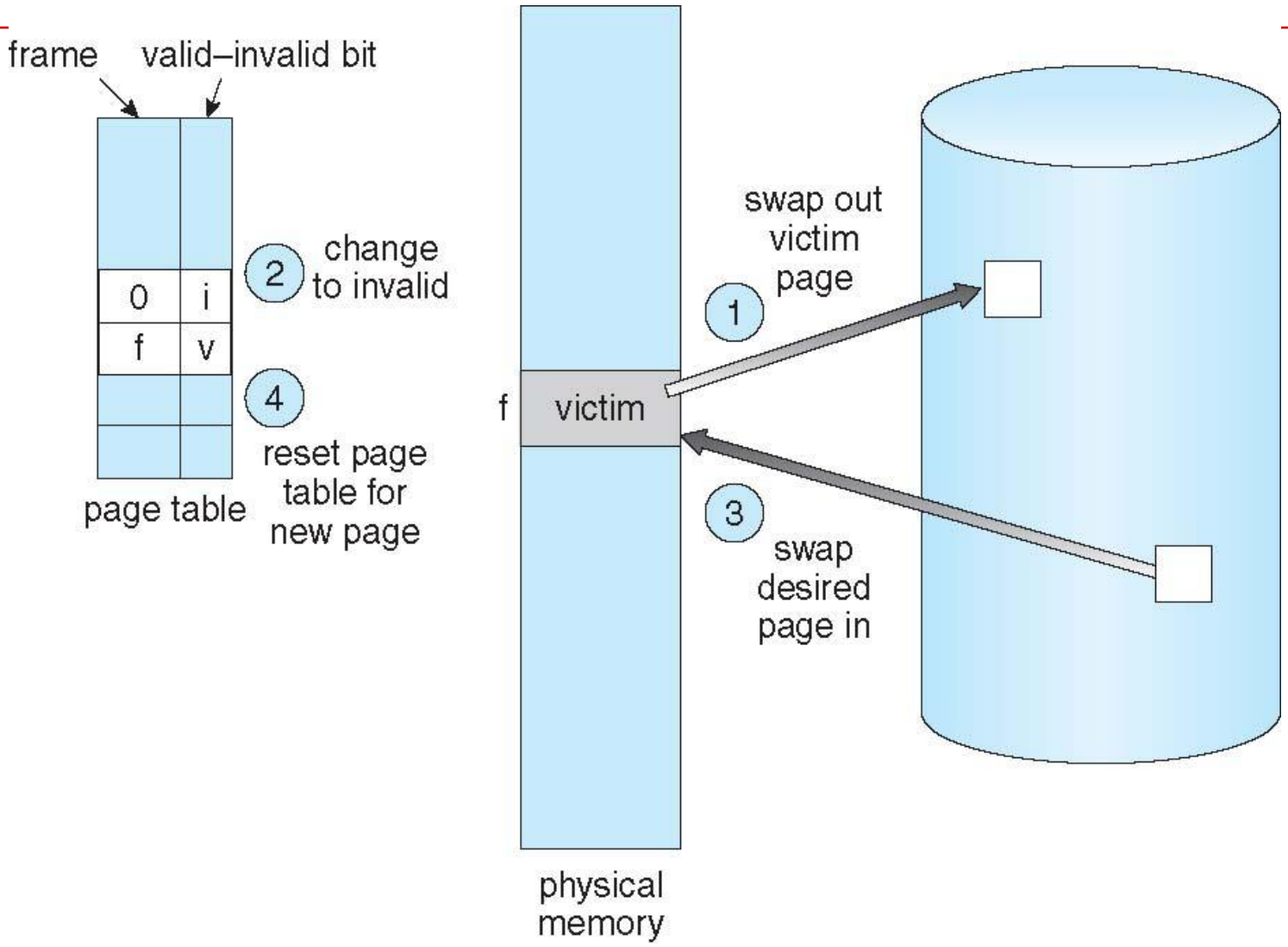
Demand paging when there is no free frame?

- **Page replacement – find some page in memory, but not really in use, swap it out**
  - Algorithm
  - Performance – want an algorithm which will result in minimum number of page faults
- **Same page may be brought into memory several times**

Need For Page Replacement



Page Replacement



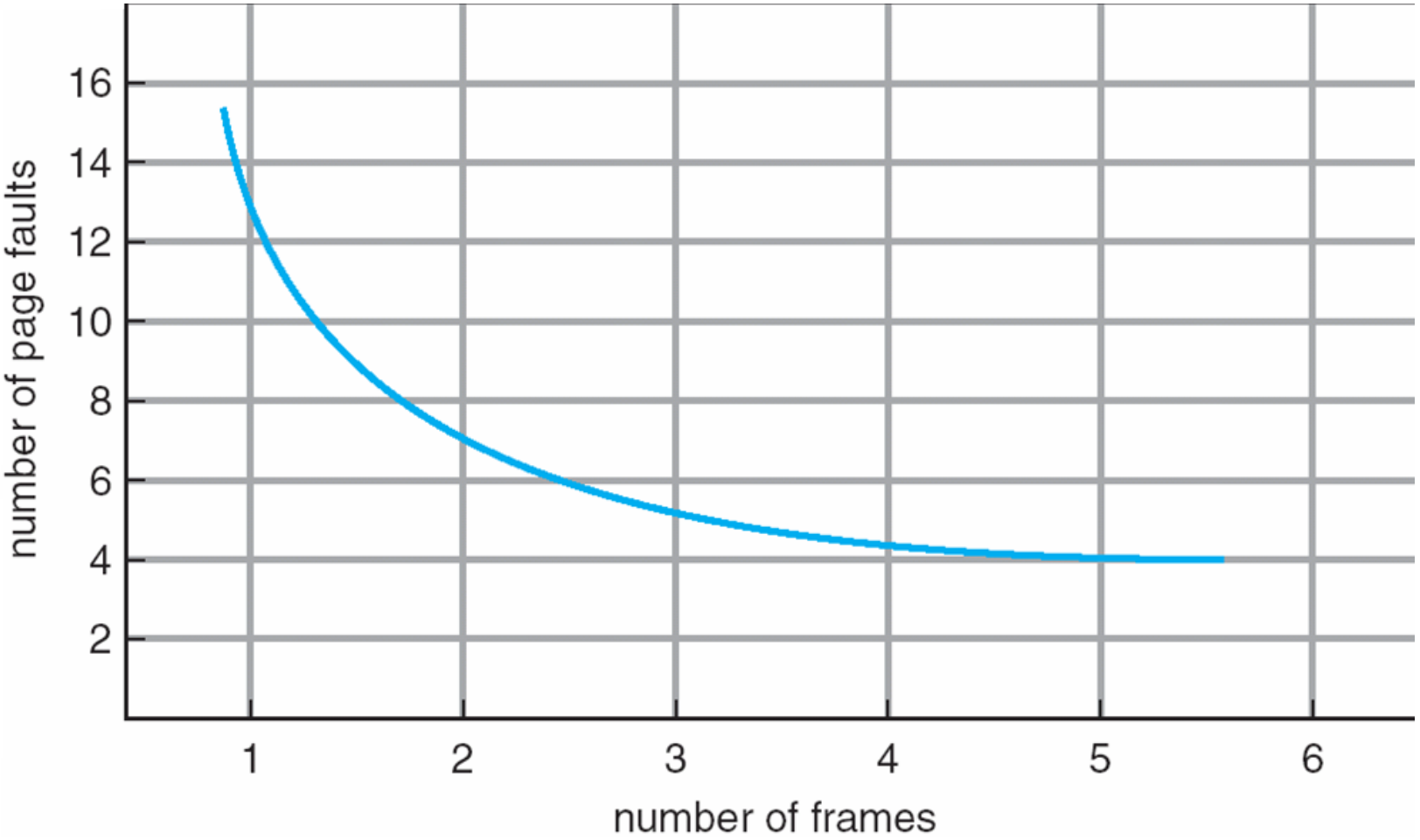
Basic Page Replacement

1. Find the location of the desired page on disk

2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a victim frame
3. Swap out: Use modify (dirty) bit to reduce overhead of page transfers – only modified pages are written to disk
4. Bring the desired page into the free frame.

   Update the page and frame tables

Expected behavior: # of Page Faults vs. # of Physical Frames

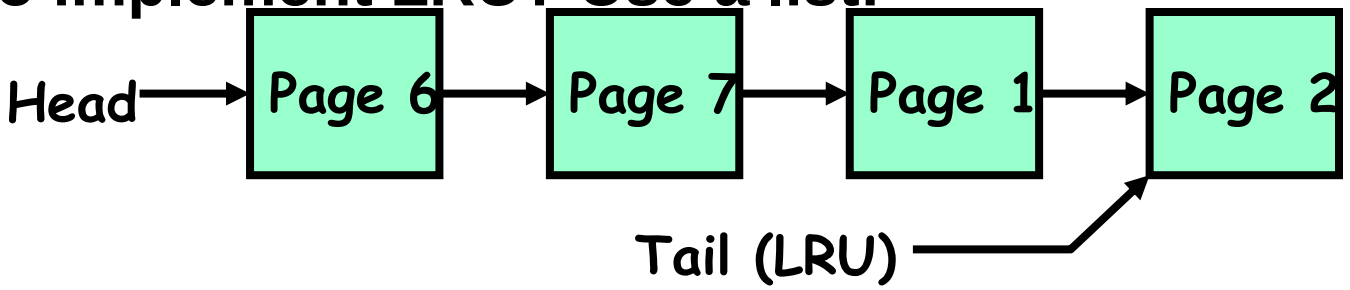


Page Replacement Policies

- **Why do we care about Replacement Policy?**
  - Replacement is an issue with any cache
  - Particularly important with pages
    - The cost of being wrong is high: must go to disk
    - Must keep important pages in memory, not toss them out
- **FIFO (First In, First Out)**
  - Throw out oldest page. Be fair – let every page live in memory for same amount of time.
  - Bad, because throws out heavily used pages instead of infrequently used pages
- **MIN (Minimum):**
  - Replace page that won't be used for the longest time
  - Great, but can't really know future…
  - Makes good comparison case, however
- **RANDOM:**
  - Pick random page for every replacement
  - Typical solution for TLB's. Simple hardware
  - Pretty unpredictable – makes it hard to make real-time guarantees

Replacement Policies (Con't)

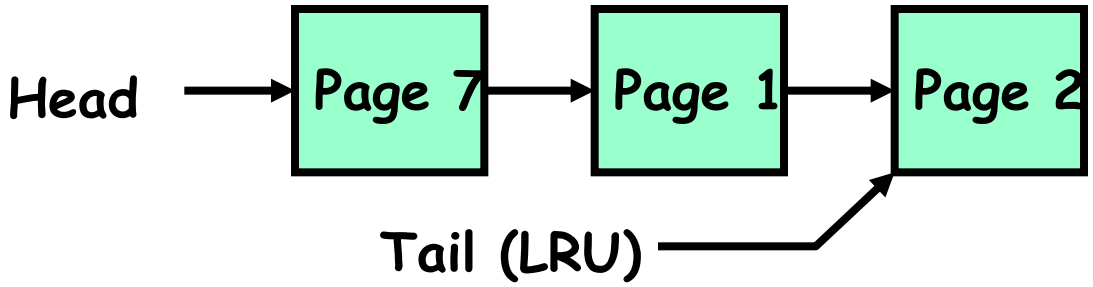
- **LRU (Least Recently Used):**
  - Replace page that hasn't been used for the longest time
  - Programs have locality, so if something not used for a while, unlikely to be used in the near future.
  - Seems like LRU should be a good approximation to MIN.
- **How to implement LRU? Use a list!**



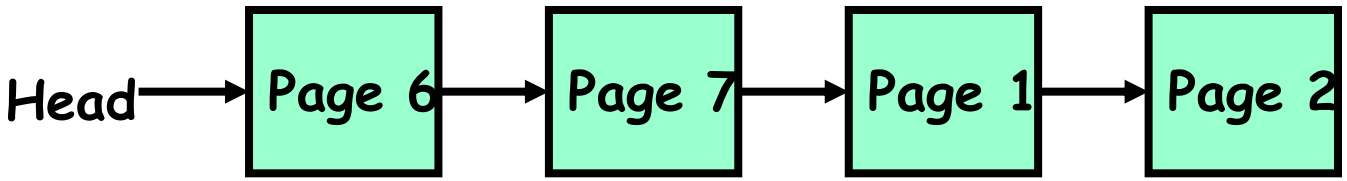
  - On each use, remove page from list and place at head
  - LRU page is at tail
- **Problems with this scheme for paging?**
  - Need to know immediately when each page used so that can change position in list…
  - Many instructions for each hardware access
- **In practice, people approximate LRU (more later)**

LRU Example

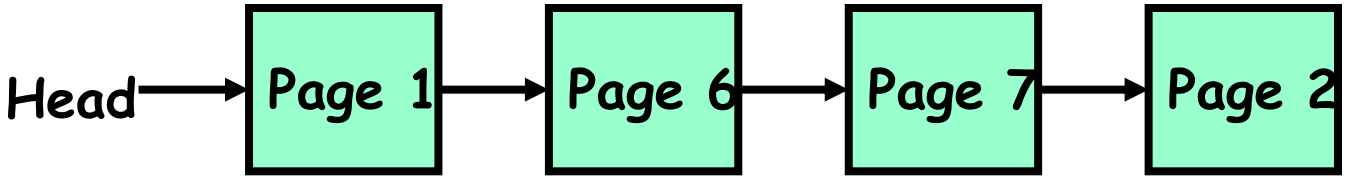
- Initially



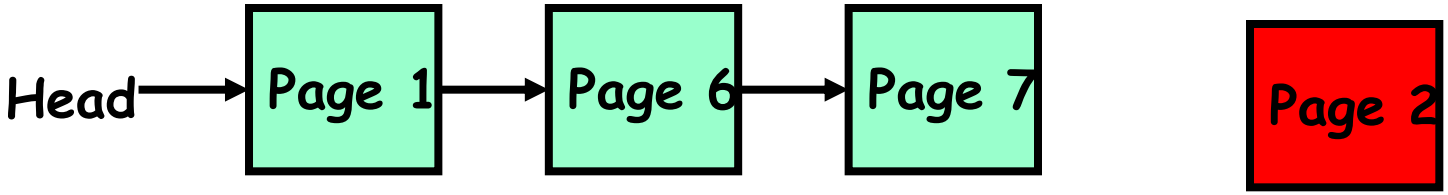
- Access Page 6



- Access Page 1

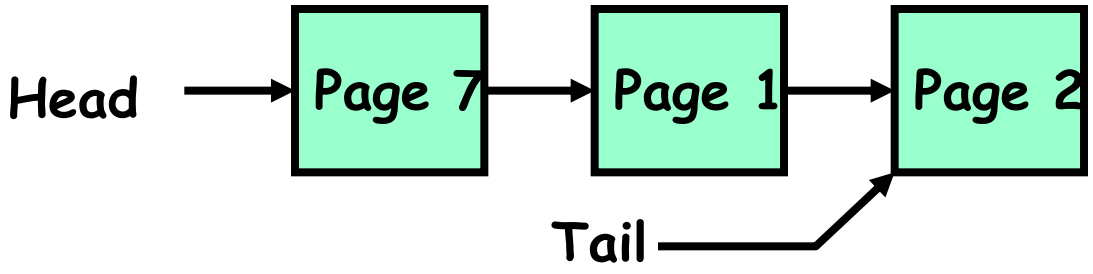


- Find a victim to remove

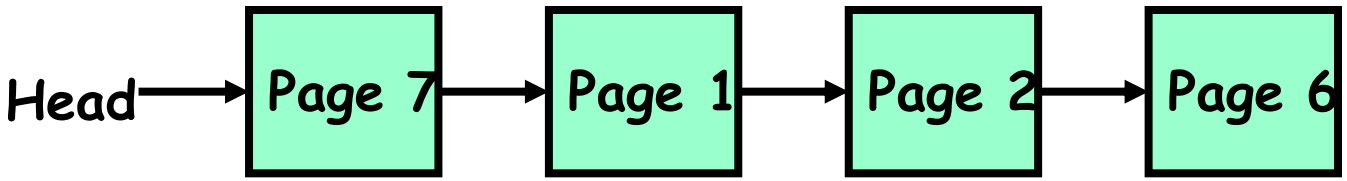


FIFO Example

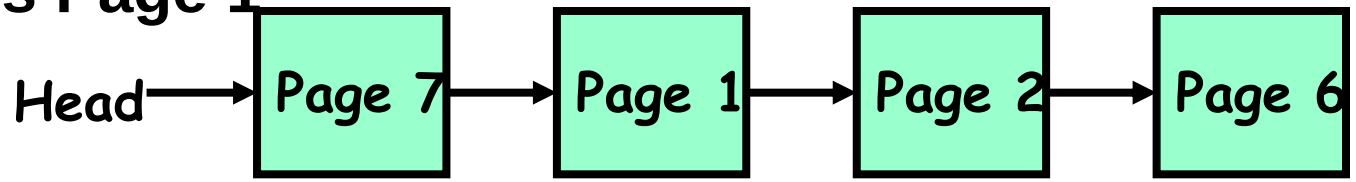
- Initially



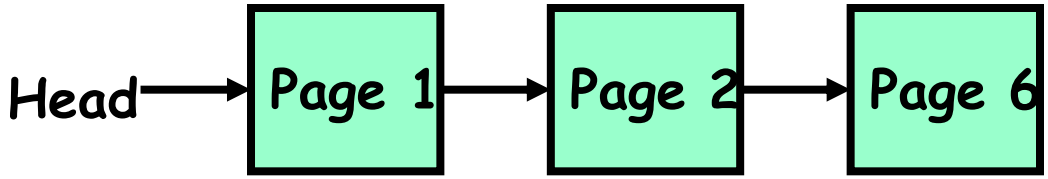
- Access Page 6



- Access Page 1



- Find a victim to remove



Example: FIFO

- **Suppose we have 3 page frames, 4 virtual pages, and following reference stream:**
  - A B C A B D A D B C B
- **Consider FIFO Page replacement:**

Ref: / Page:	A	B	C	A	B	D	A	D	B	C	B
1	A					D				C	
2		B					A				
3			C						B		

  - FIFO: 7 faults.
  - When referencing D, replacing A is bad choice, since need A again right away

Example: MIN

- **Suppose we have the same reference stream:**
  - A B C A B D A D B C B
- **Consider MIN Page replacement:**

Ref: / Page:	A	B	C	A	B	D	A	D	B	C	B
1	A									C	
2		B									
3			C			D					

  - MIN: 5 faults
  - Where will D be brought in? Look for page not referenced farthest in future.
- **What will LRU do?**
  - Same decisions as MIN here, but won't always be true!

When will LRU perform badly?

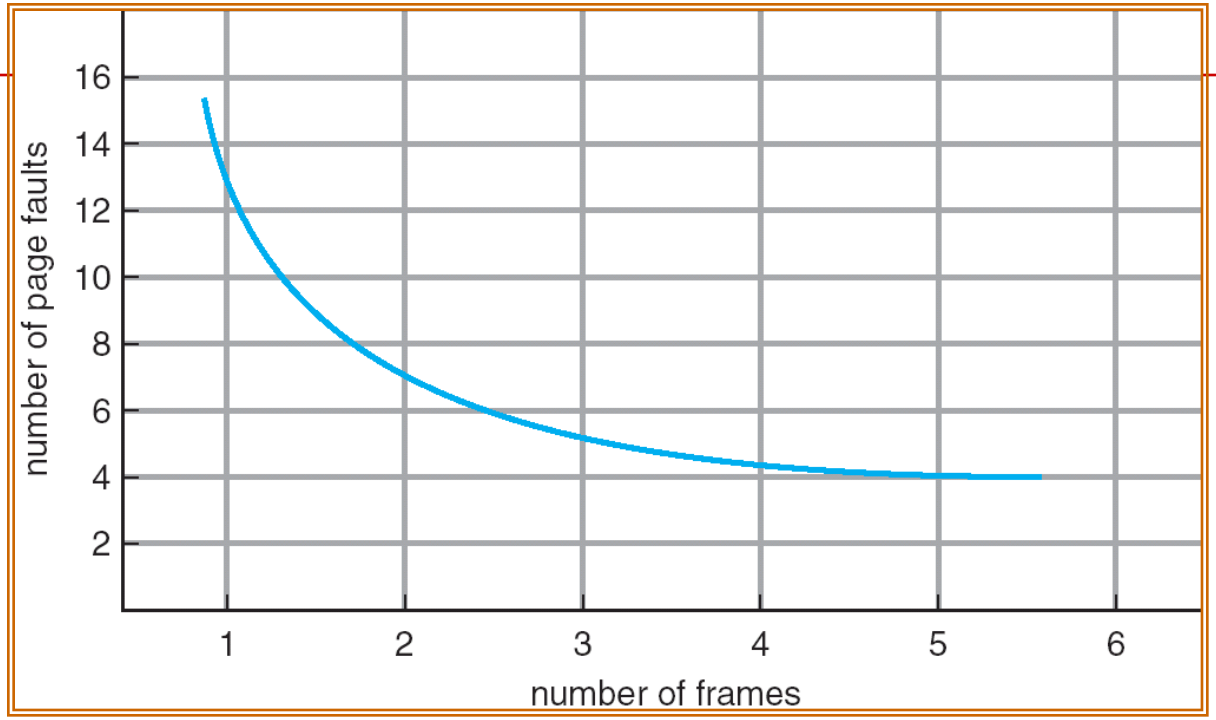
- **Consider the following: A B C D A B C D A B C D**
- **LRU Performs as follows (same as FIFO here):**

Ref: / Page:	A	B	C	D	A	B	C	D	A	B	C	D
1	A			D			C			B		
2		B			A			D			C	
3			C			B			A			D

  - Every reference is a page fault!
- **MIN Does much better:**

Ref: / Page:	A	B	C	D	A	B	C	D	A	B	C	D
1	A									B		
2		B					C					
3			C	D								

Graph of Page Faults Versus The Number of Frames



- **One desirable property: When you add memory the miss rate goes down**
  - Does this always happen?
  - Seems like it should, right?
- **No: BeLady's anomaly**
  - Certain replacement algorithms (FIFO) don't have this obvious property!

Adding Memory Doesn't Always Help Fault Rate

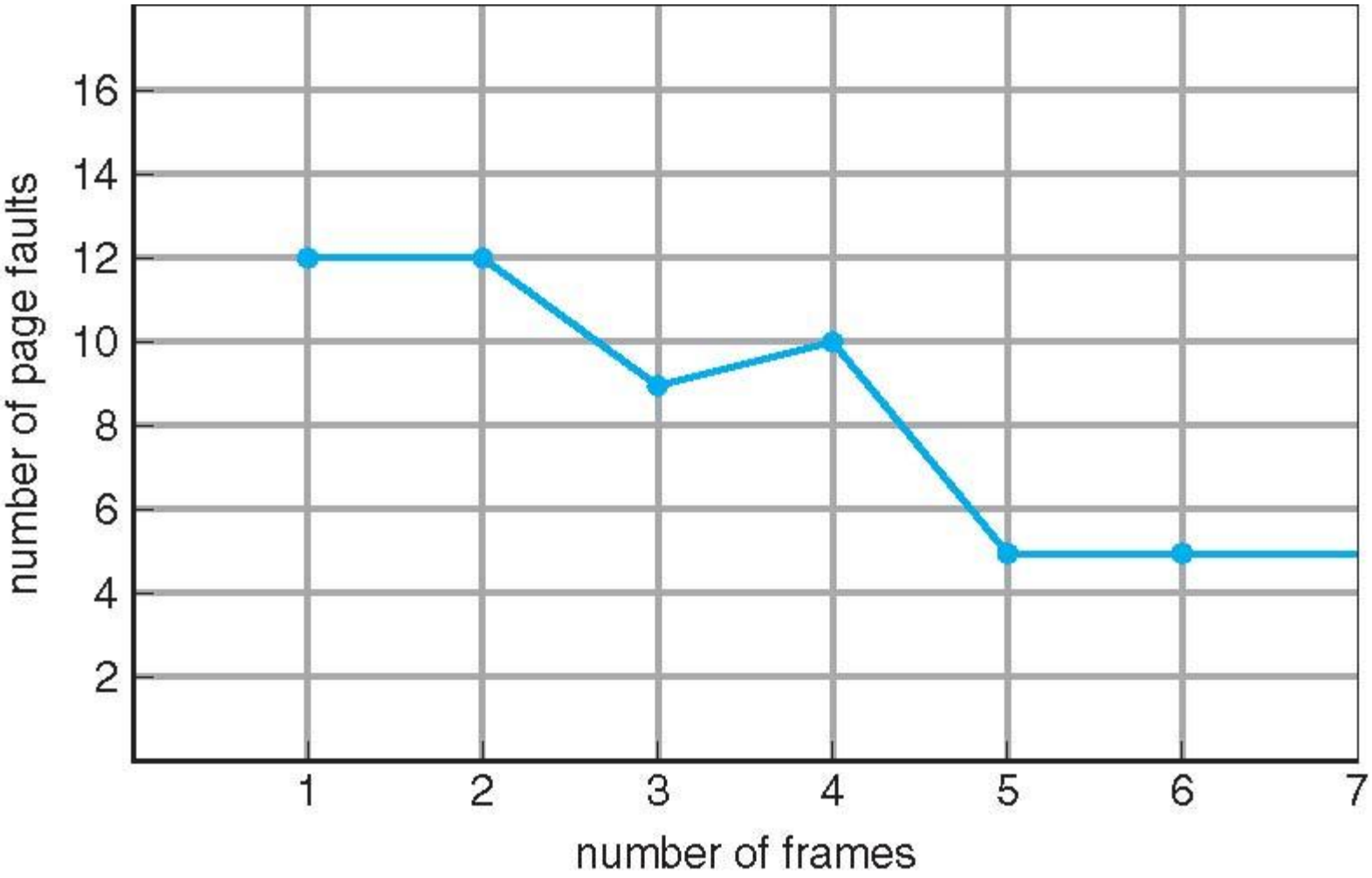
- **Does adding memory reduce number of page faults?**
  - Yes for LRU and MIN
  - Not necessarily for FIFO! (Called Belady's anomaly)

Ref: Page:	A	B	C	D	A	B	E	A	B	C	D	E
1	A			D			E					
2		B			A					C		
3			C			B					D	

Ref: Page:	A	B	C	D	A	B	E	A	B	C	D	E
1	A						E				D	
2		B						A				E
3			C						B			
4				D						C		

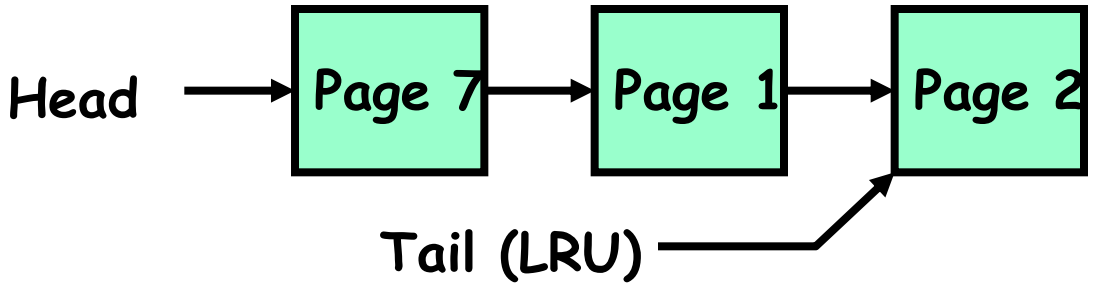
- **After adding memory:**
  - With FIFO, contents can be completely different
  - In contrast, with LRU or MIN, contents of memory with X pages are a subset of contents with X+1 Page

FIFO Illustrating Belady's Anomaly

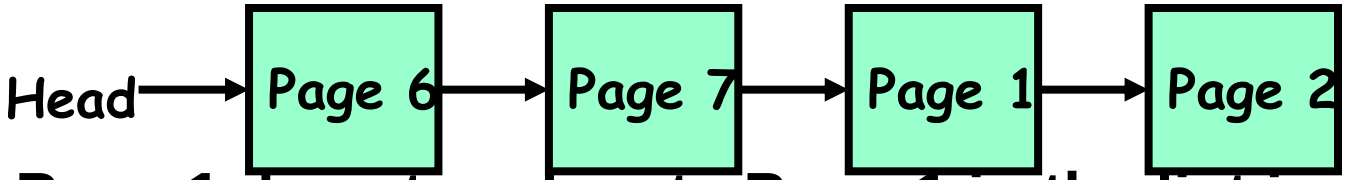


Implement LRU

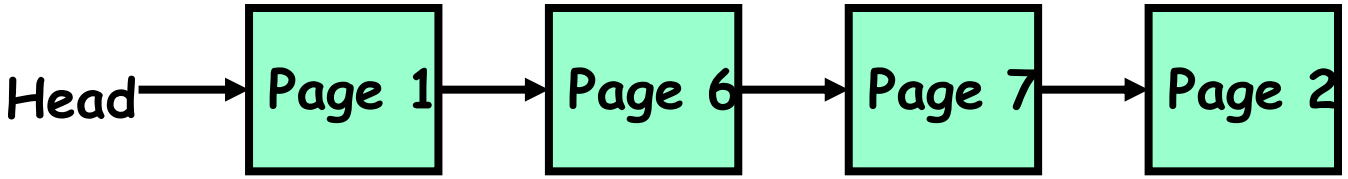
- Initially



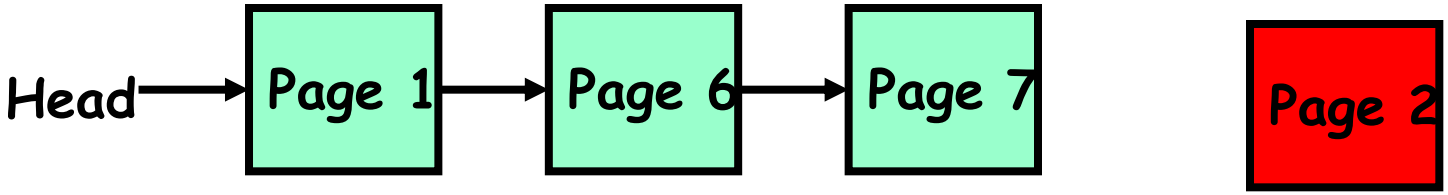
- Access Page 6



- Access Page 1, how to relocate Page 1 in the list in O(1)?



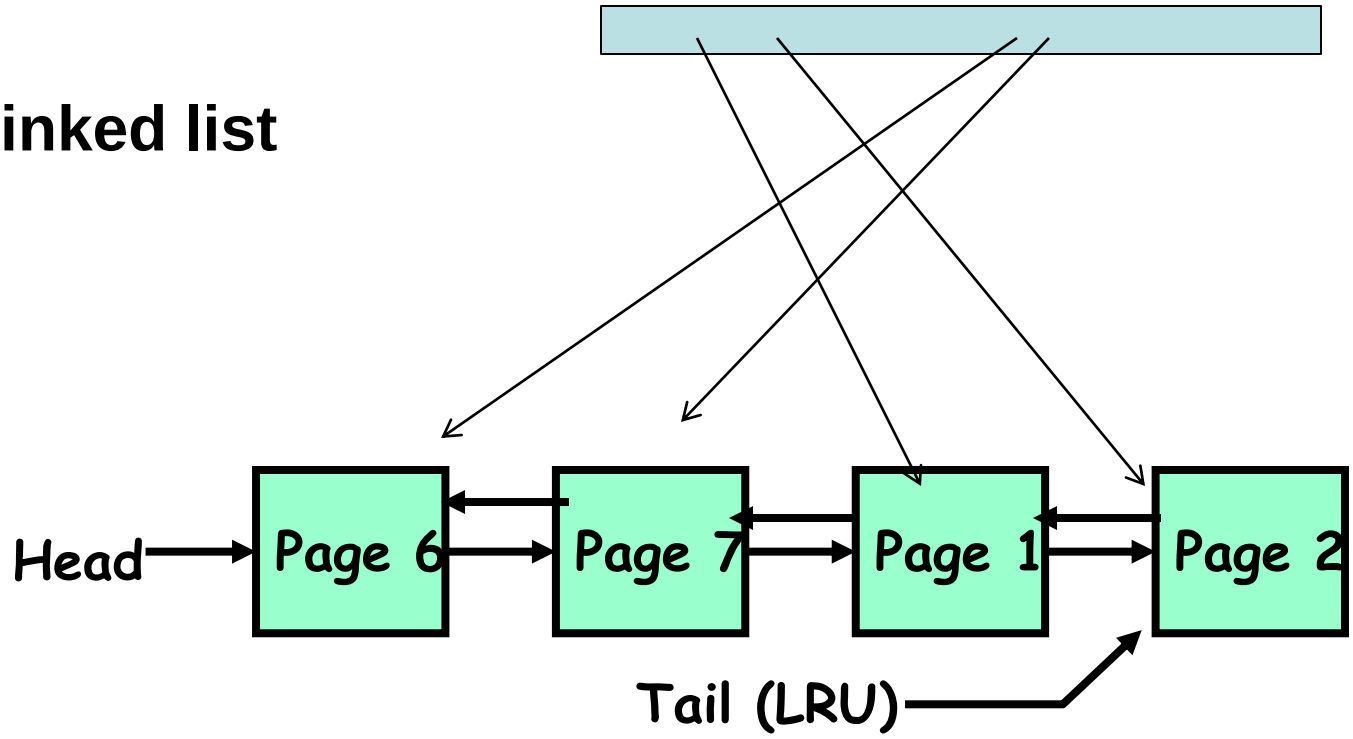
- Find a victim: how to remove the tail in O(1)?



Implement LRU

How many memory accesses per page access?

- **Key-value map or bitmap**
- **Double linked list**



Essentially

Keep list of pages ordered by time of reference

Too expensive to implement in reality for many reasons.

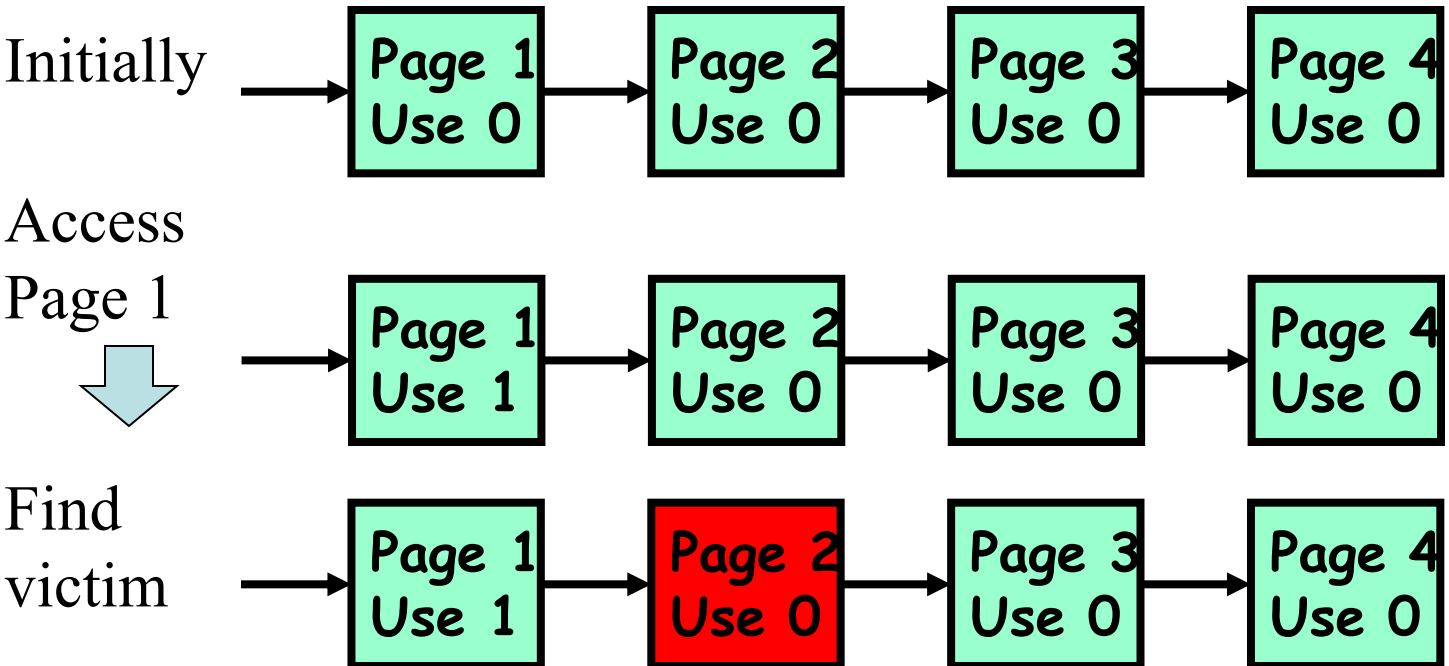
(6 pointer updates when accessing a page)

Implementing LRU with Approximation

- **Clock Algorithm: Arrange physical pages in circle with single clock hand**
  - Approximate LRU
  - Replace an old page, may not the oldest page
- **Details:**
  - Hardware "use" bit per physical page:
    - Hardware sets use bit on each reference
    - If use bit isn't set, means not referenced in a long time
    - Nachos hardware sets use bit in the TLB; you have to copy this back to page table when TLB entry gets replaced
  - On page fault:
    - Advance clock hand (not real time)
    - Check use bit: 1→used recently; clear and leave alone
      0→selected candidate for replacement
  - Will always find a page or loop forever?
    - Even if all use bits set, will eventually loop around⇒FIFO

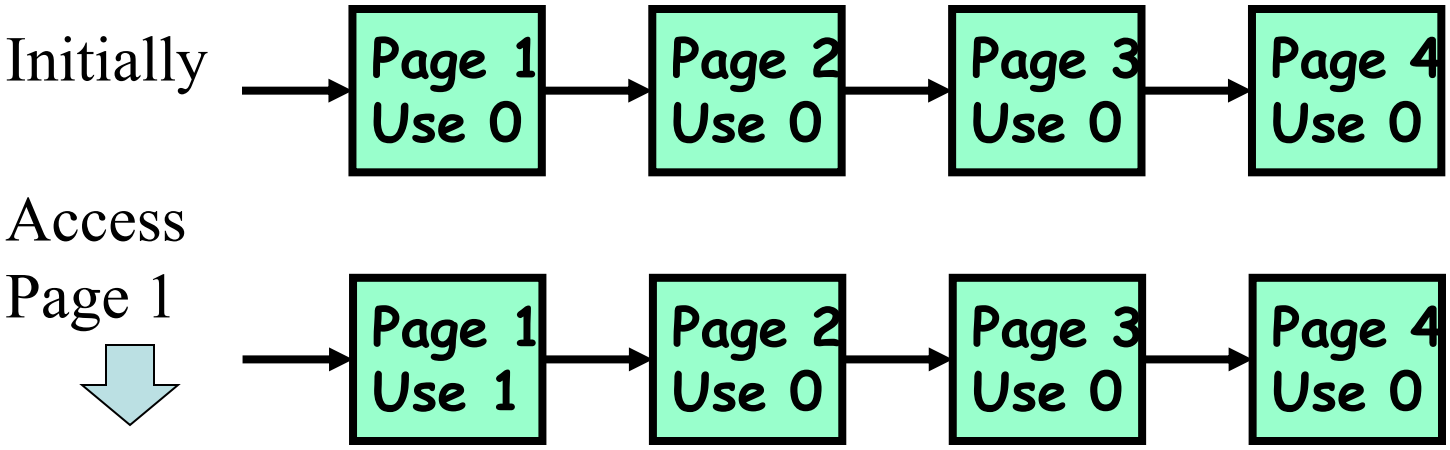
LRU Approximation: Use bit

- **Use bit (or called reference bit)**
  - With each page associate a bit, initially = 0
  - When page is referenced, bit set to 1
  - Replace the one which is 0 (if one exists)
    - We do not know the order, however

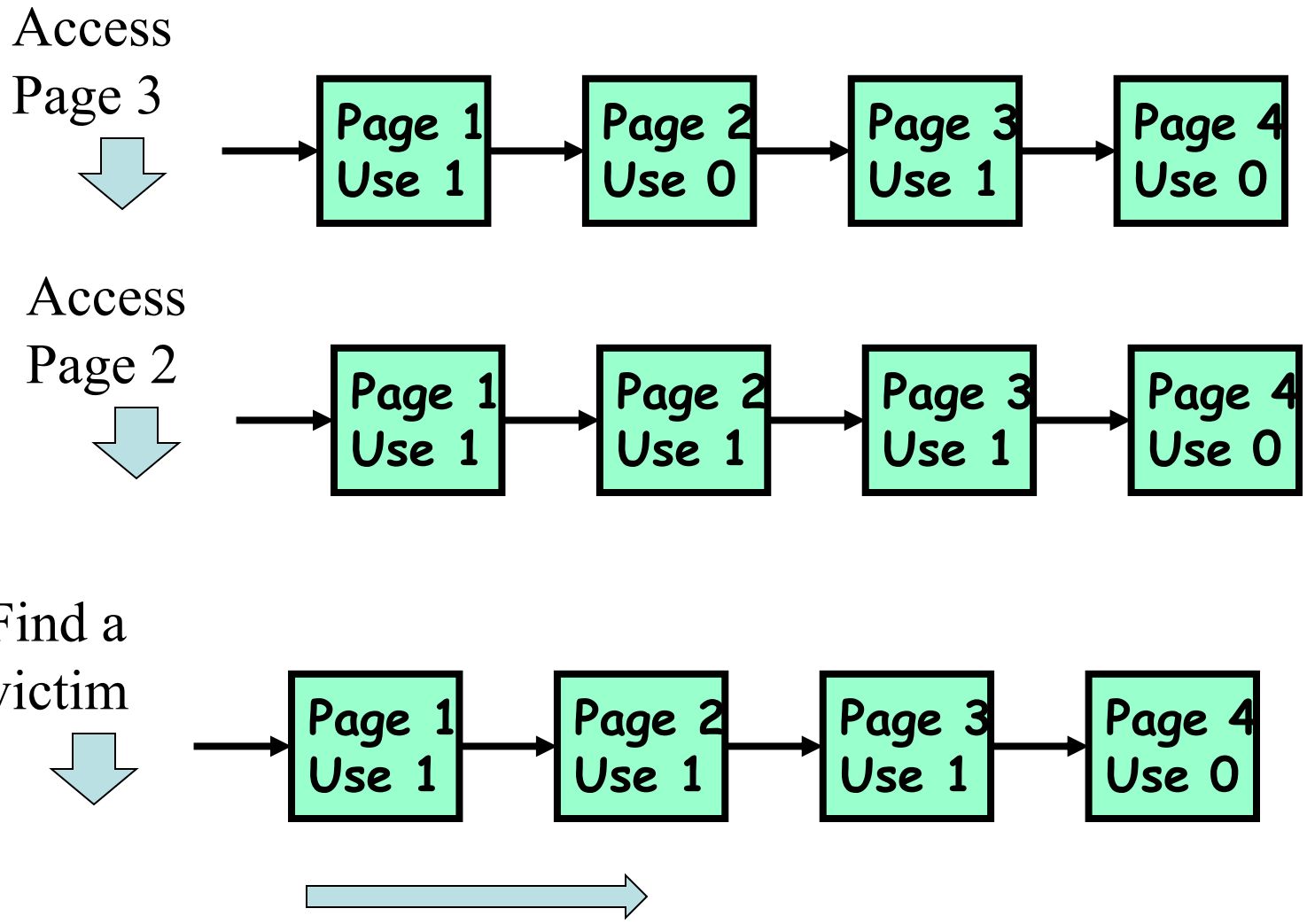


LRU Approximation: Second chance

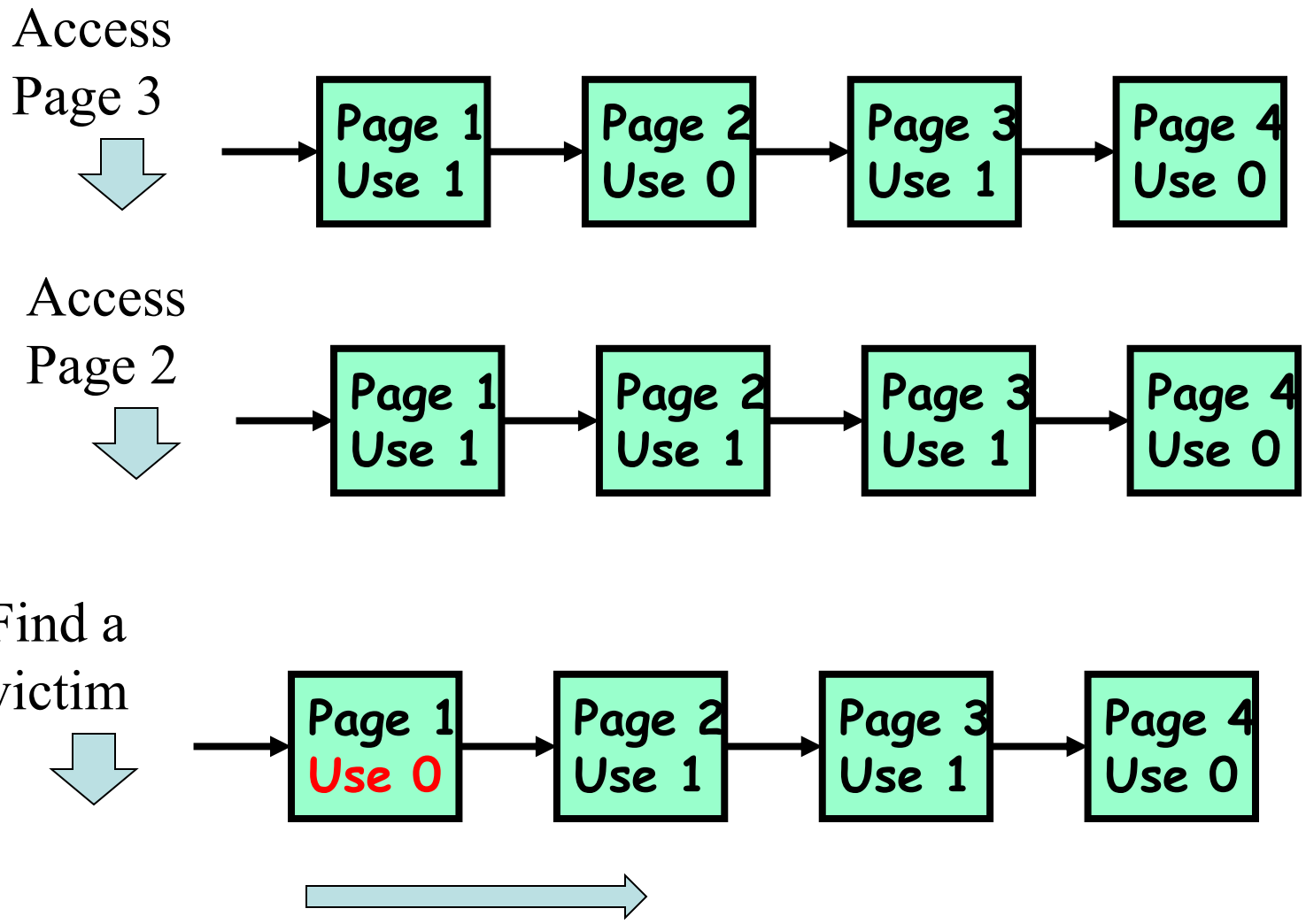
- **Second chance in replacement**
  - If page has reference bit = 1 then:
    - Leave this page in memory, but set reference bit 0.
    - Visit next page (in clock order)
  - If reference bit is 0, replace it.



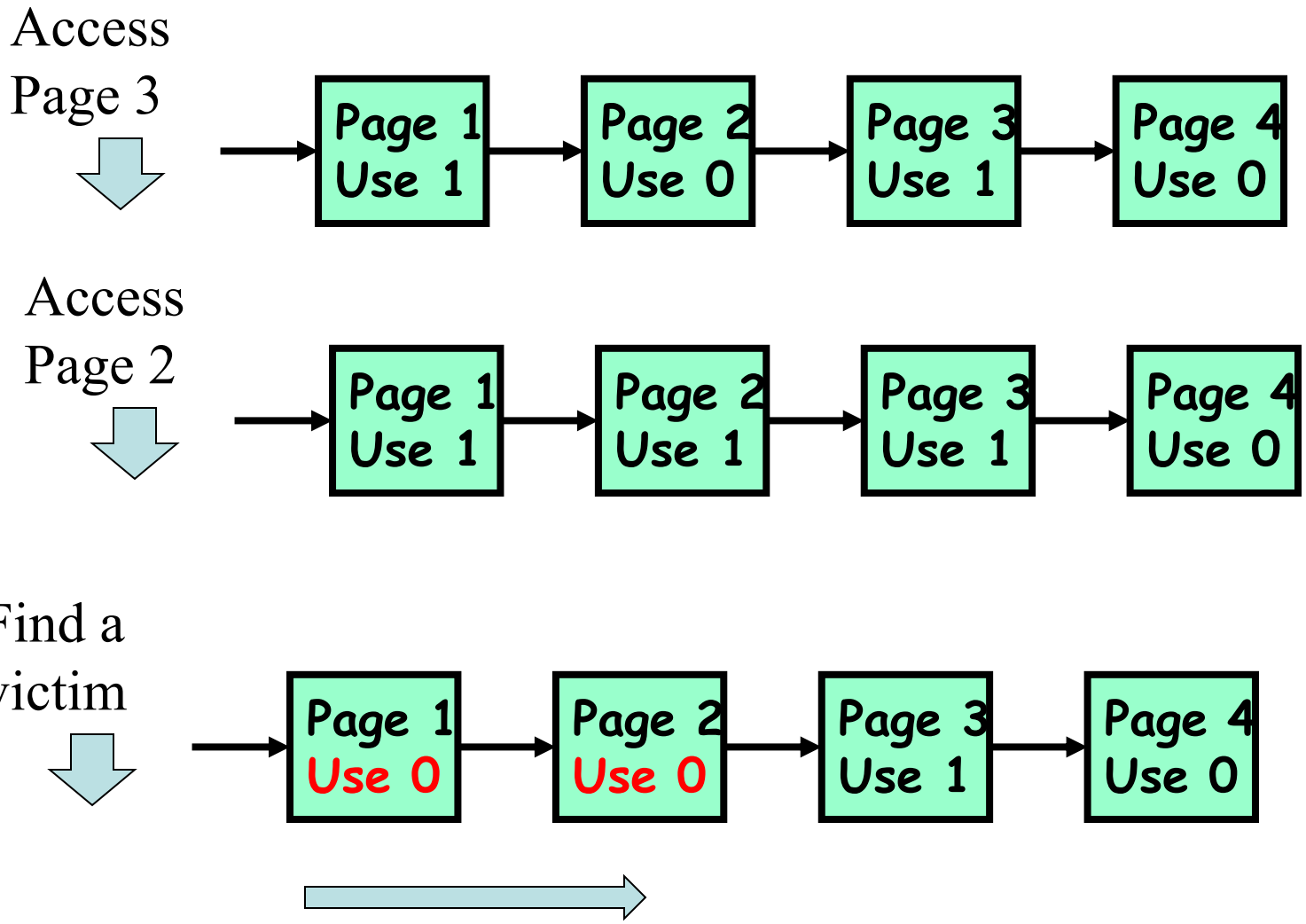
LRU Approximation: Second chance



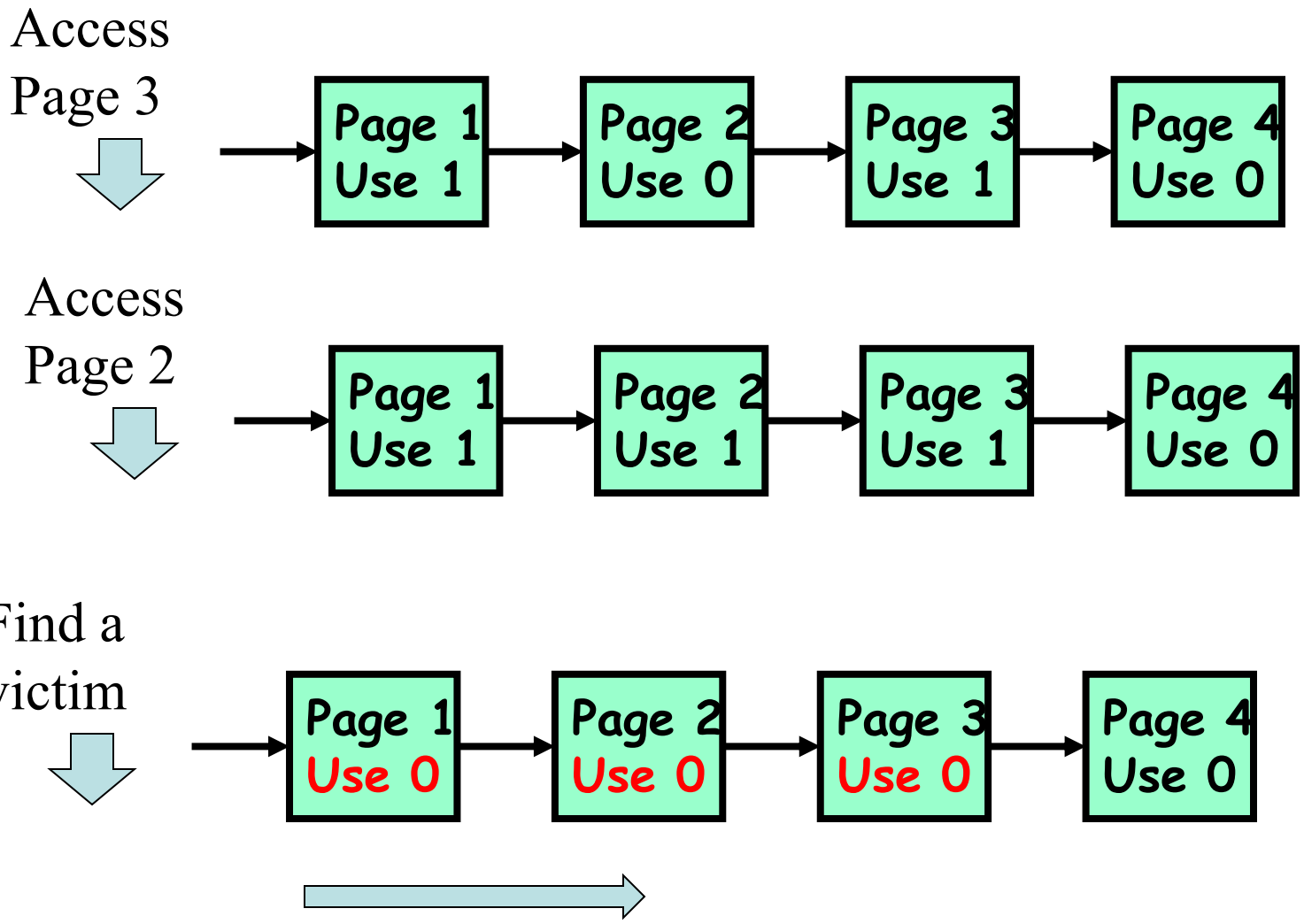
LRU Approximation: Second chance



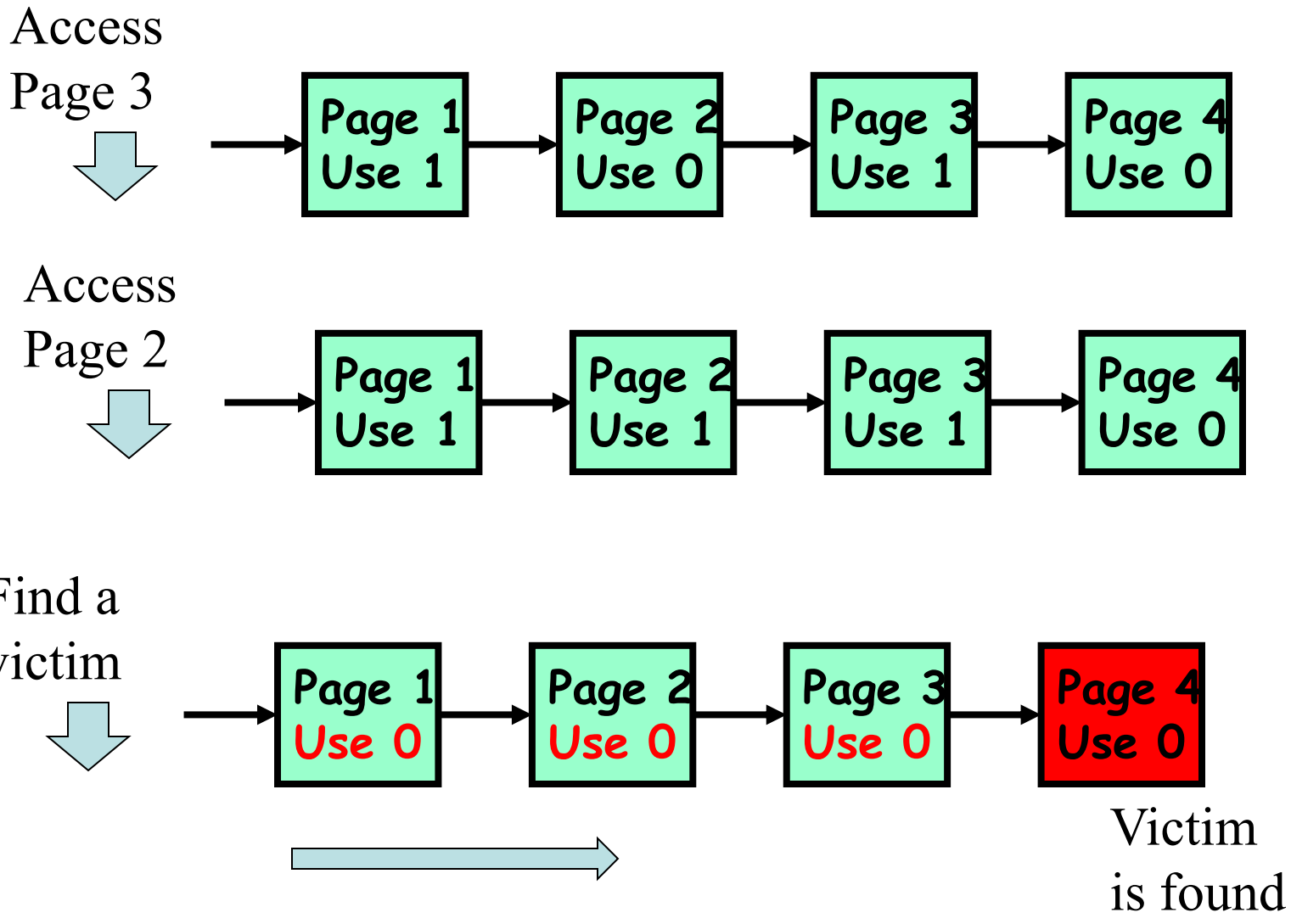
LRU Approximation: Second chance



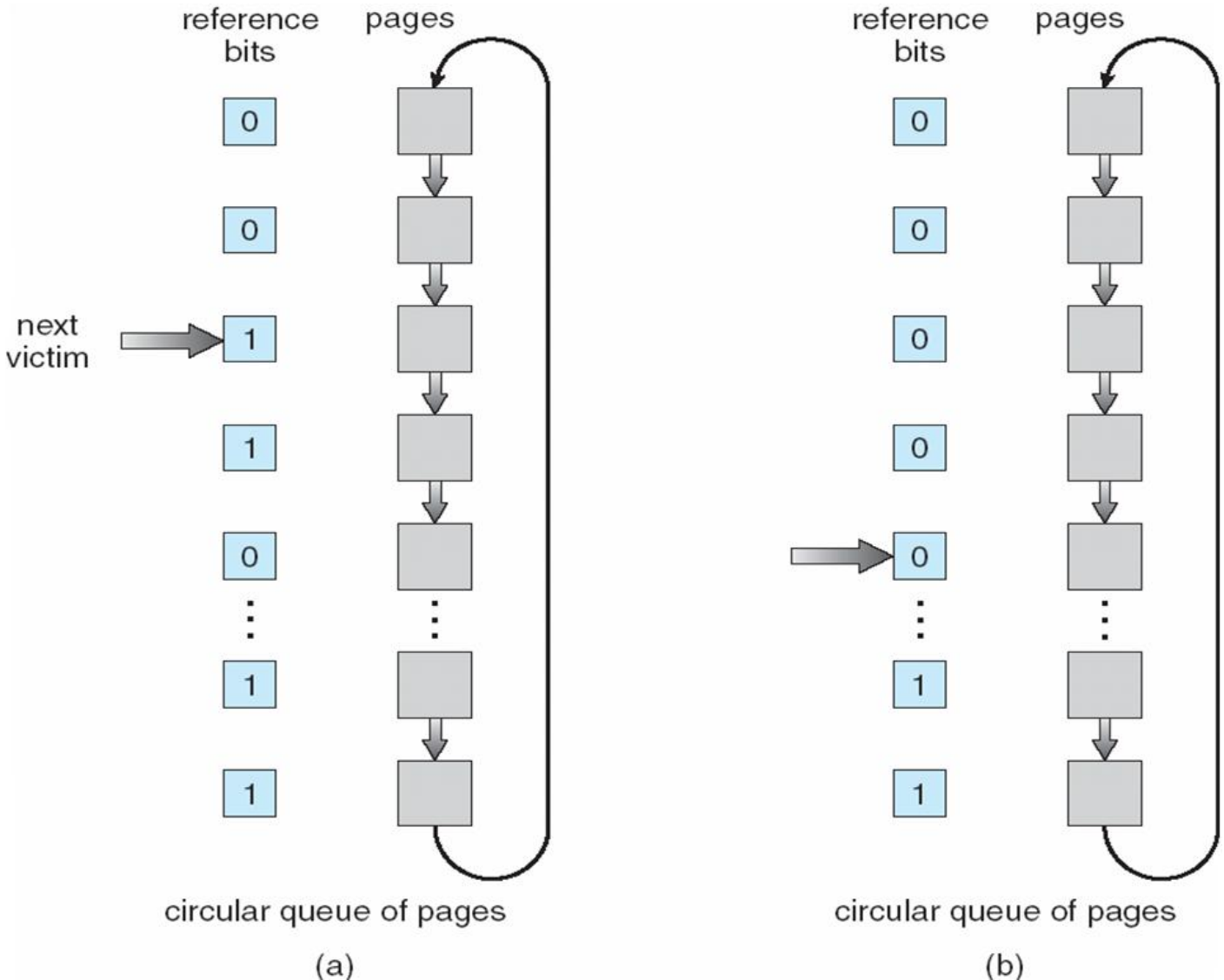
LRU Approximation: Second chance



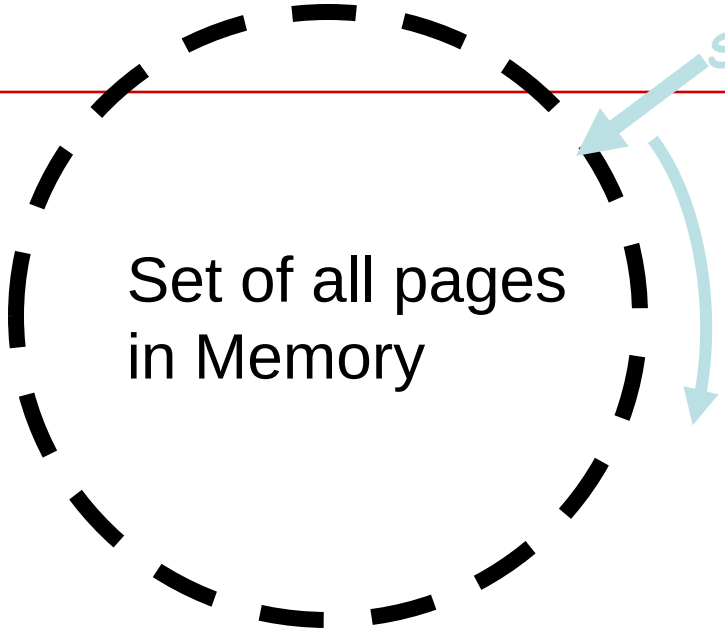
LRU Approximation: Second chance



Search in second-chance page-replacement is circular



Clock Algorithm: Not Recently Used



Single Clock Hand:

Advances only on page fault!
Check for pages not used recently
Mark pages as not used recently

- **What if hand moving slowly?**
  - Good sign or bad sign?
    - Not many page faults and/or find page quickly
- **What if hand is moving quickly?**
  - Lots of page faults and/or lots of reference bits set
- **One way to view clock algorithm:**
  - Crude partitioning of pages into two groups: young and old
  - Why not partition into more than 2 groups?

N^{th} Chance version of Clock Algorithm

- **N^{th} chance algorithm: Give page N chances**
  - OS keeps counter per page: # sweeps
  - On page fault, OS checks use bit:
    - 1⇒clear use and also clear counter (used in last sweep)
    - 0⇒increment counter; if count=N, replace page
  - Means that clock hand has to sweep by N times without page being used before page is replaced
- **How do we pick N?**
  - Why pick large N? Better approx to LRU
    - If N ~ 1K, really good approximation
  - Why pick small N? More efficient
    - Otherwise might have to look a long way to find free page
- **What about dirty pages?**
  - Takes extra overhead to replace a dirty page, so give dirty pages an extra chance before replacing?
  - Common approach:
    - Clean pages, use N=1
    - Dirty pages, use N=2 (and write back to disk when N=1)

Issues with Page Allocation and Replacement

- **Initial Allocation. Each process needs *minimum* number of pages**
  - Two schemes: Fixed allocation vs. priority allocation
- **Where to find frames**
- **Global replacement – find a frame from all processes.**
- **Local replacement – find only from its own allocated frames**

Fixed Allocation

- **0 allocation**
- **Equal allocation –** For example, if there are 100 frames and 5 processes, give each process 20 frames.

$s_i = \text{size of process } p_i$

$S = \sum s_i$

$m = \text{total number of frames}$

$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$

- **Proportional allocation – Allocate according to the size of process**

$m = 64$

$s_i = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \times 64 \approx 5$

$a_2 = \frac{127}{137} \times 64 \approx 59$

Priority Allocation

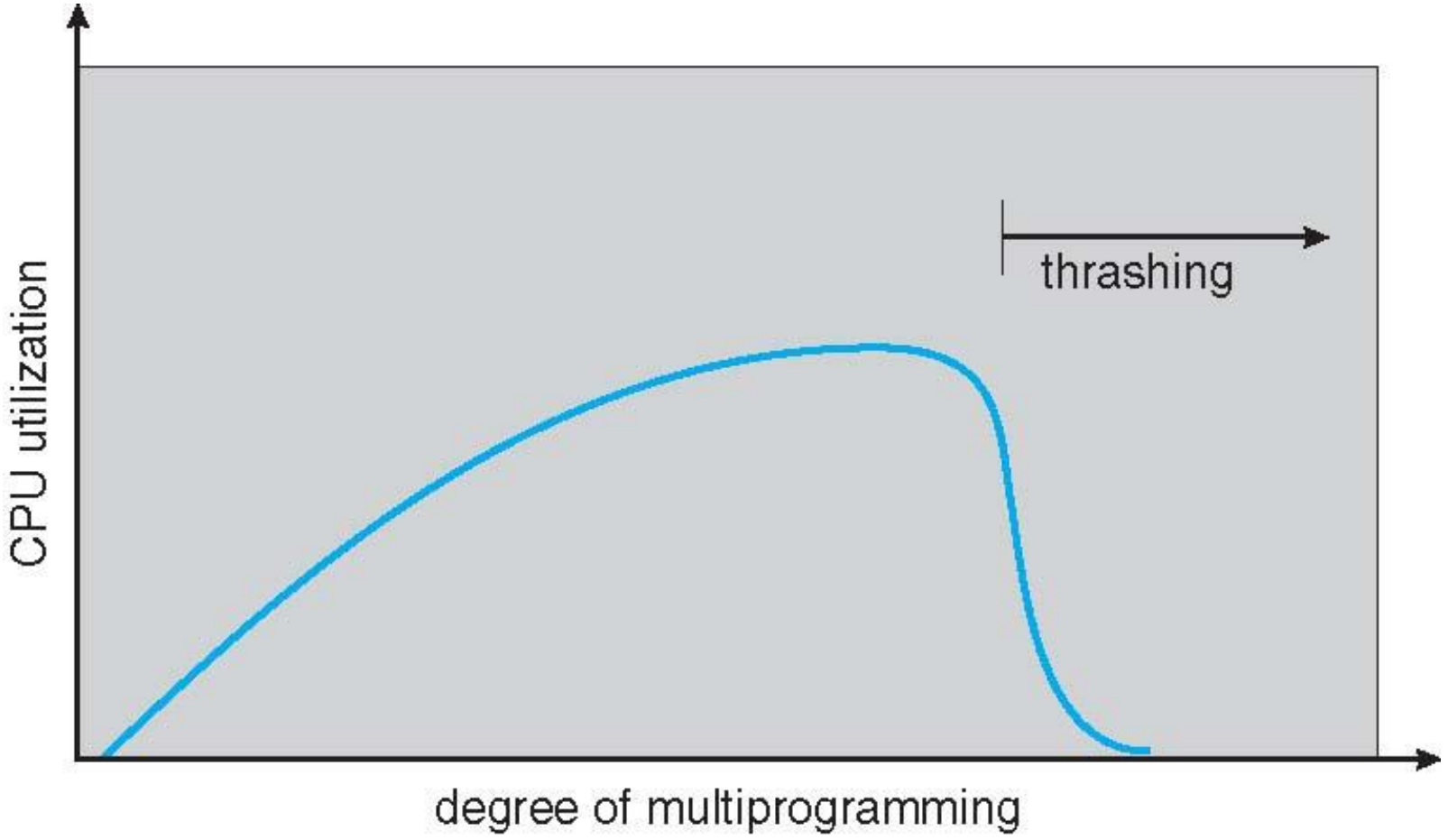
- **Use a proportional allocation scheme using priorities rather than size**

- **If process P_i generates a page fault,**
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number

Thrashing

- **If a process does not have “enough” pages, the page-fault rate is very high. This leads to:**
  - low CPU utilization
  - operating system thinks that it needs to increase the degree of multiprogramming
  - another process added to the system
- **Thrashing ≡ a process is busy swapping pages in and out**

Thrashing (Cont.)

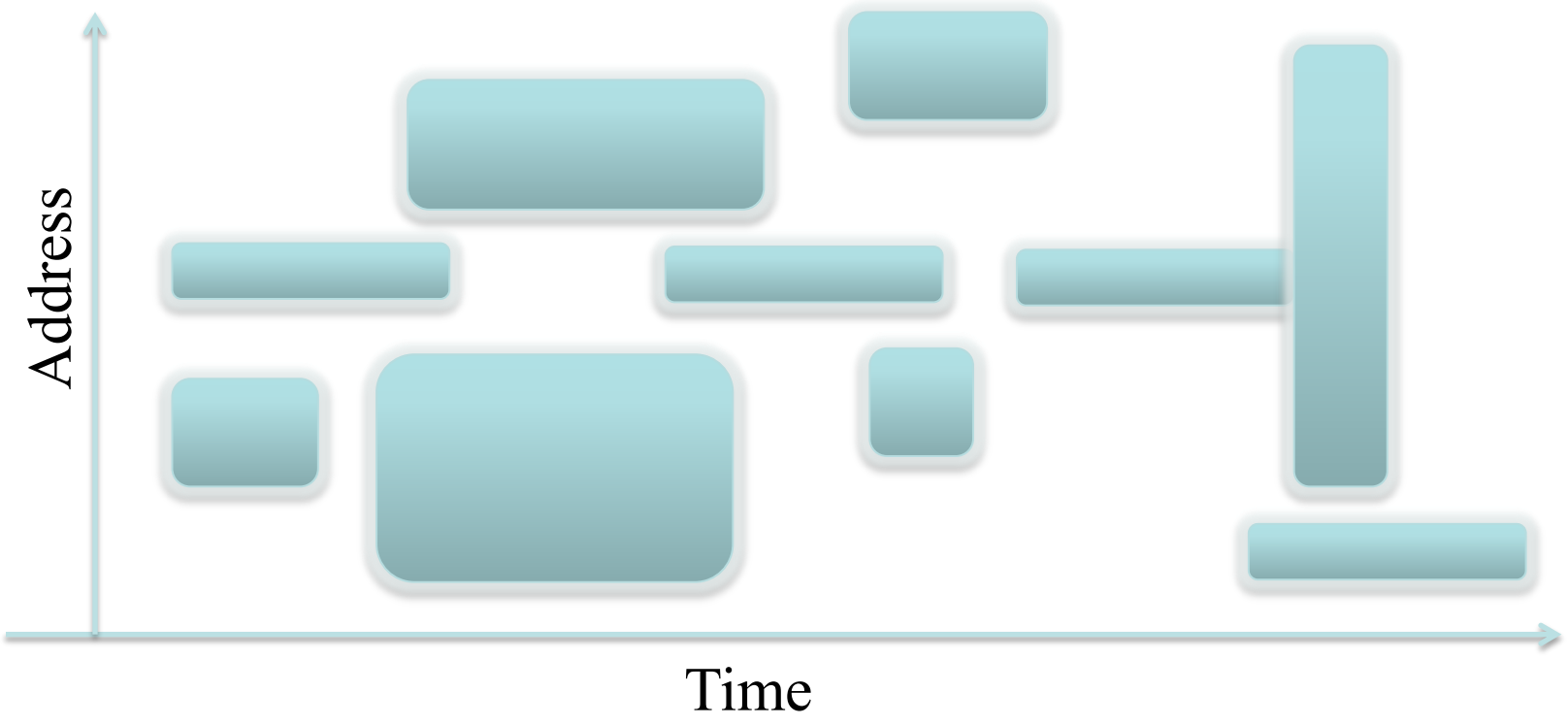


Relationship of Demand Paging and Thrashing

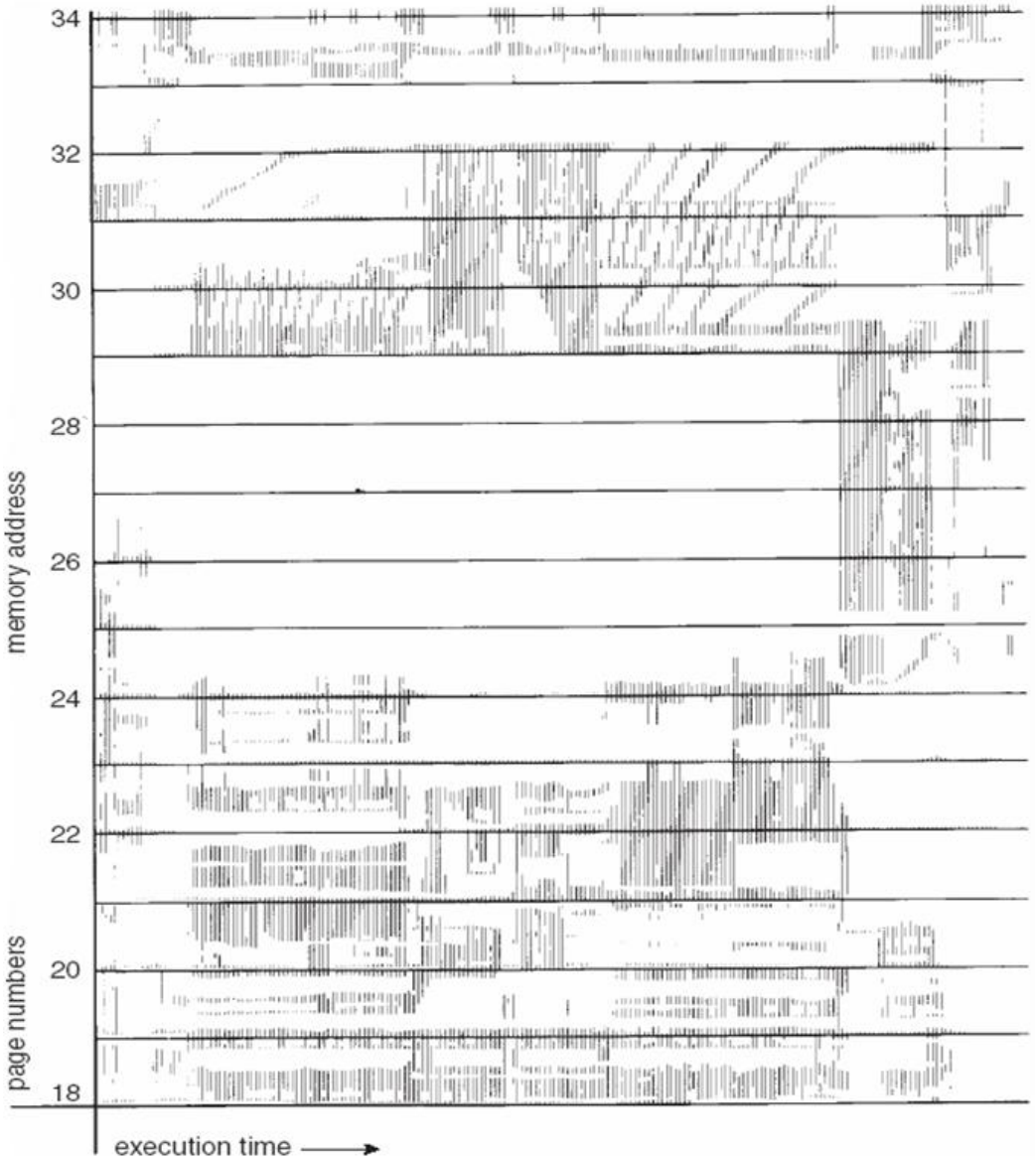
- **Why does demand paging work?**
  Locality model
  - Process migrates from one locality to another
  - Localities may overlap
  - Need a minimum working set to be effective
- **Why does thrashing occur?**
  Σ working-sets > total memory size

Working Set Model

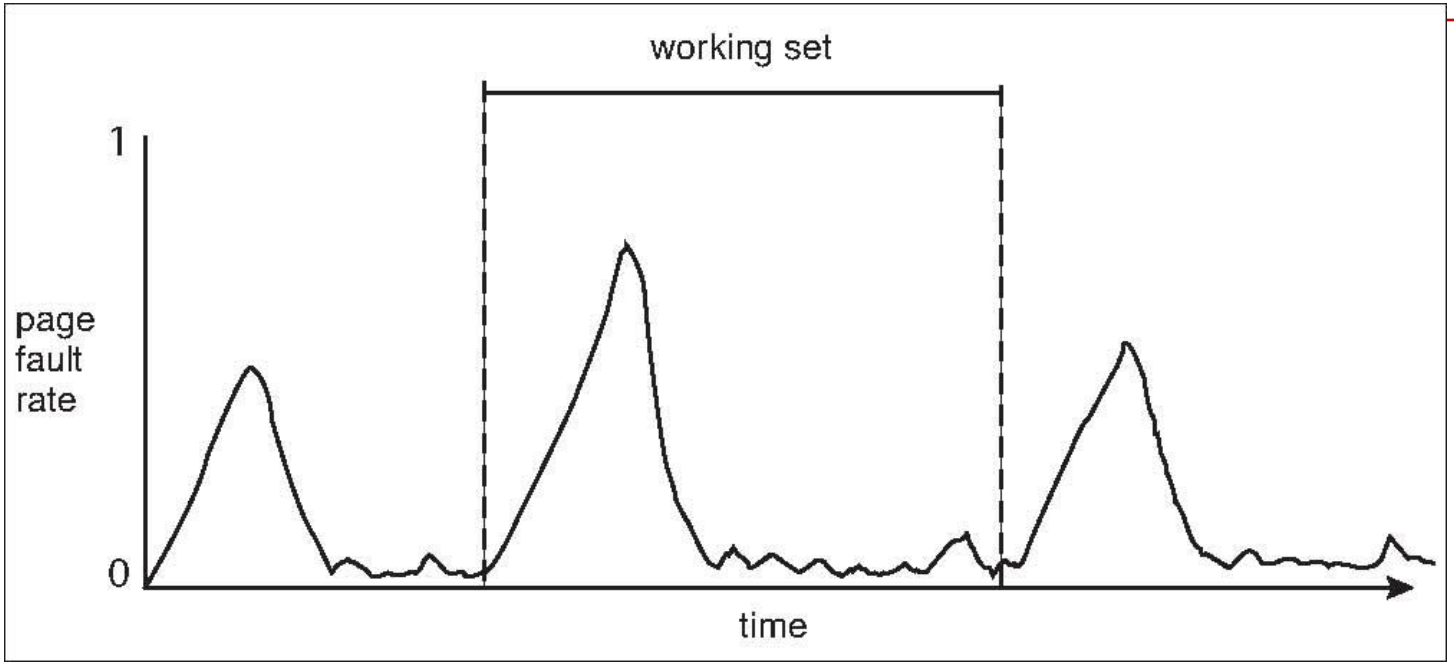
- **As a program executes it transitions through a sequence of "working sets" consisting of varying sized subsets of the address space**



Locality In A Memory-Reference Pattern

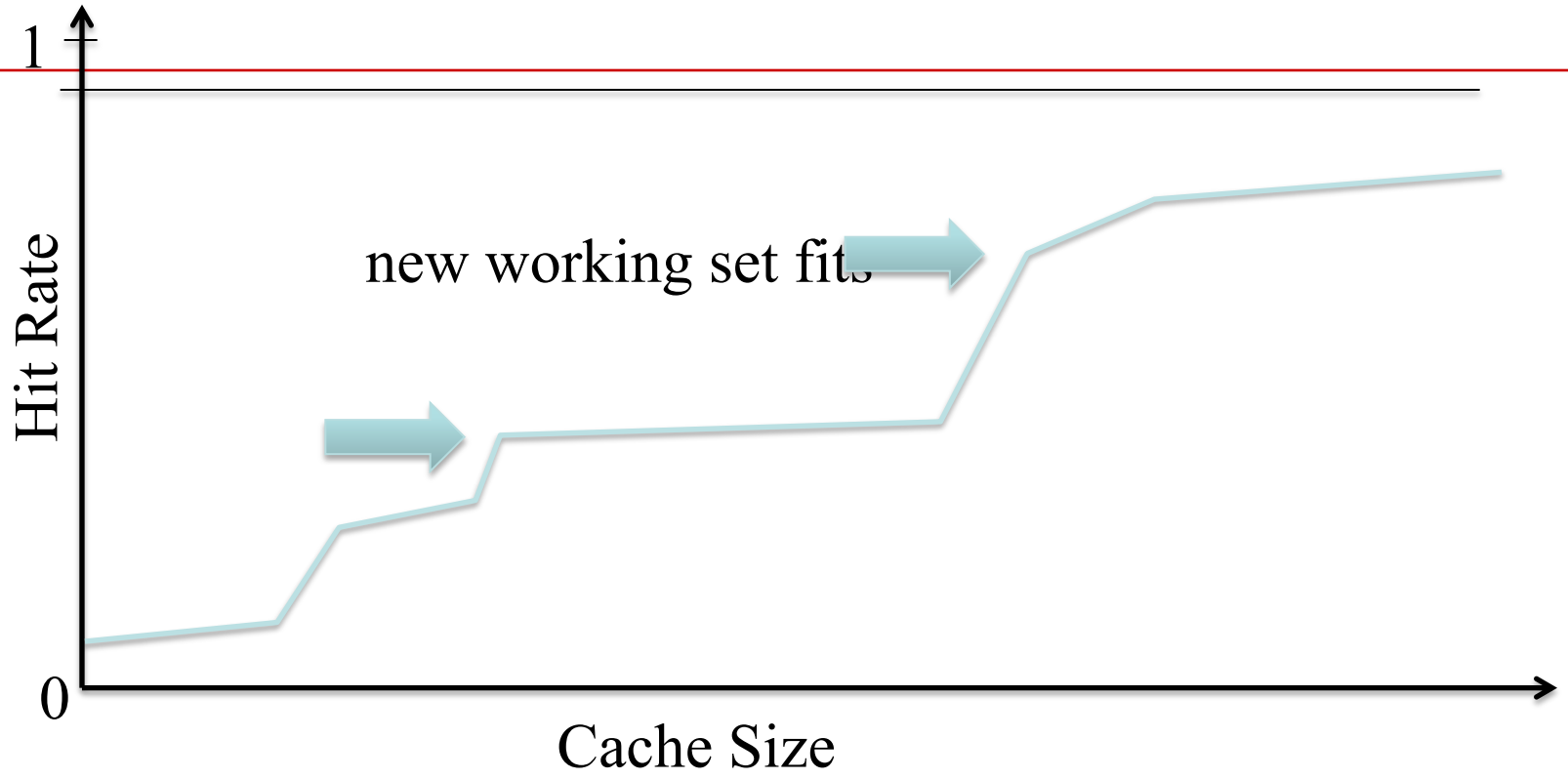


Working Sets and Page Fault Rates



- **Working set**
  - The set of memory locations that a program has referenced in the recent past.
  - Data access pattern of program
- **Great performance if working-set fits into memory**

Cache Behavior under WS model



- **Amortized by fraction of time the WS is active**
- **Transitions from one WS to the next**
- **Capacity, Conflict, Compulsory misses**
- **Applicable to memory caches and pages. Others ?**

Example of program structure, data locality and page fault

- **Less than 128 physical memory pages for each process. Each page is of 128x4 bytes**
- **Program structure (data access pattern)**
  - `int data[128,128];`
  - Each row is stored in one page
  - Page faults of Program 1?

```
for (j = 0; j <128; j++)
    for (i = 0; i < 128; i++)
        data[i,j] = 0;
```

  - Page faults of Program 2 ?

```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i,j] = 0;
```

Example of program structure, data locality and page fault

- **Less than 128 physical pages for each process. Each page is of 128x4 bytes**
- **Program structure (data access pattern)**
  - `int[128,128] D;`
  - Each row is stored in one page
  - Program 1

```
for (j = 0; j <128; j++)
    for (i = 0; i < 128; i++)
        D[i,j] = 0;
```

128 x 128 = 16,384 page faults

  - Program 2

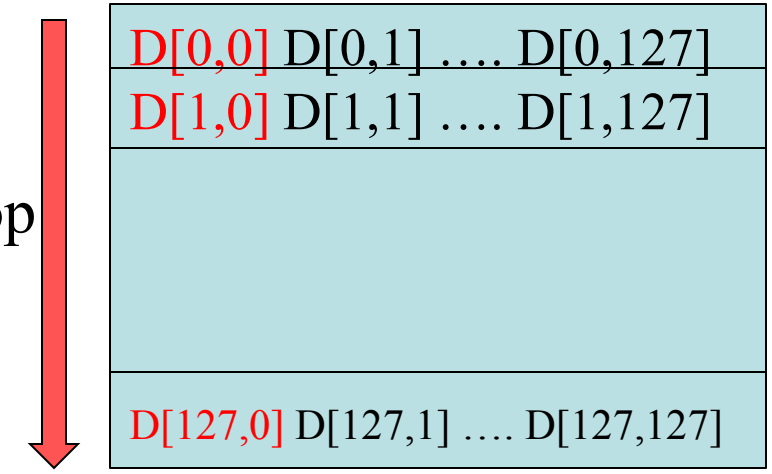
```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        D[i,j] = 0;
```

128 page faults

Impact of Data Access Pattern in a Program on Data Locality and Page Fault

```
for (j = 0; j <128; j++)
    for (i = 0; i < 128; i++)
        D[i,j] = 0;
```

128 page faults in one inner loop iteration

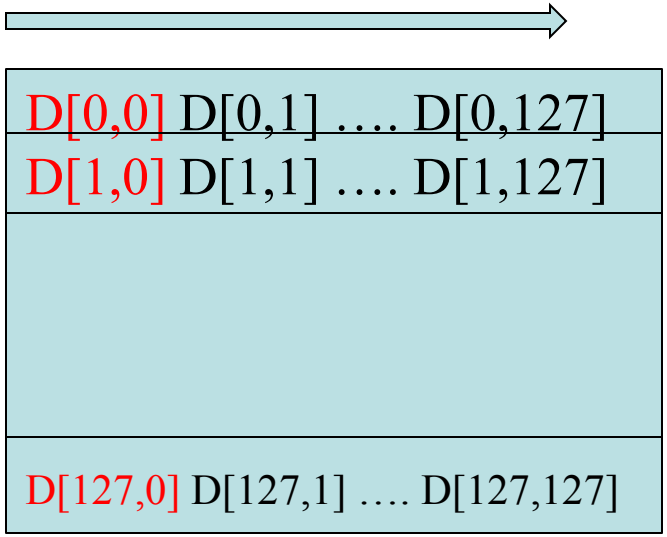


128 x 128 = 16,384 page faults

There is no data locality. Fetched page is only used once before swapping out.

Impact of Data Access Pattern in a Program on Data Locality and Page Fault

```
for (i = 0; j <128; j++)
    for (j = 0; i < 128; i++)
        D[i,j] = 0;
```



1 page fault
in one inner loop
iteration

128 page faults

There is a spatial data locality. Fetched page is used 128 times before swapping out (consecutive data access within the same page)

Miss hit hit hit …hit

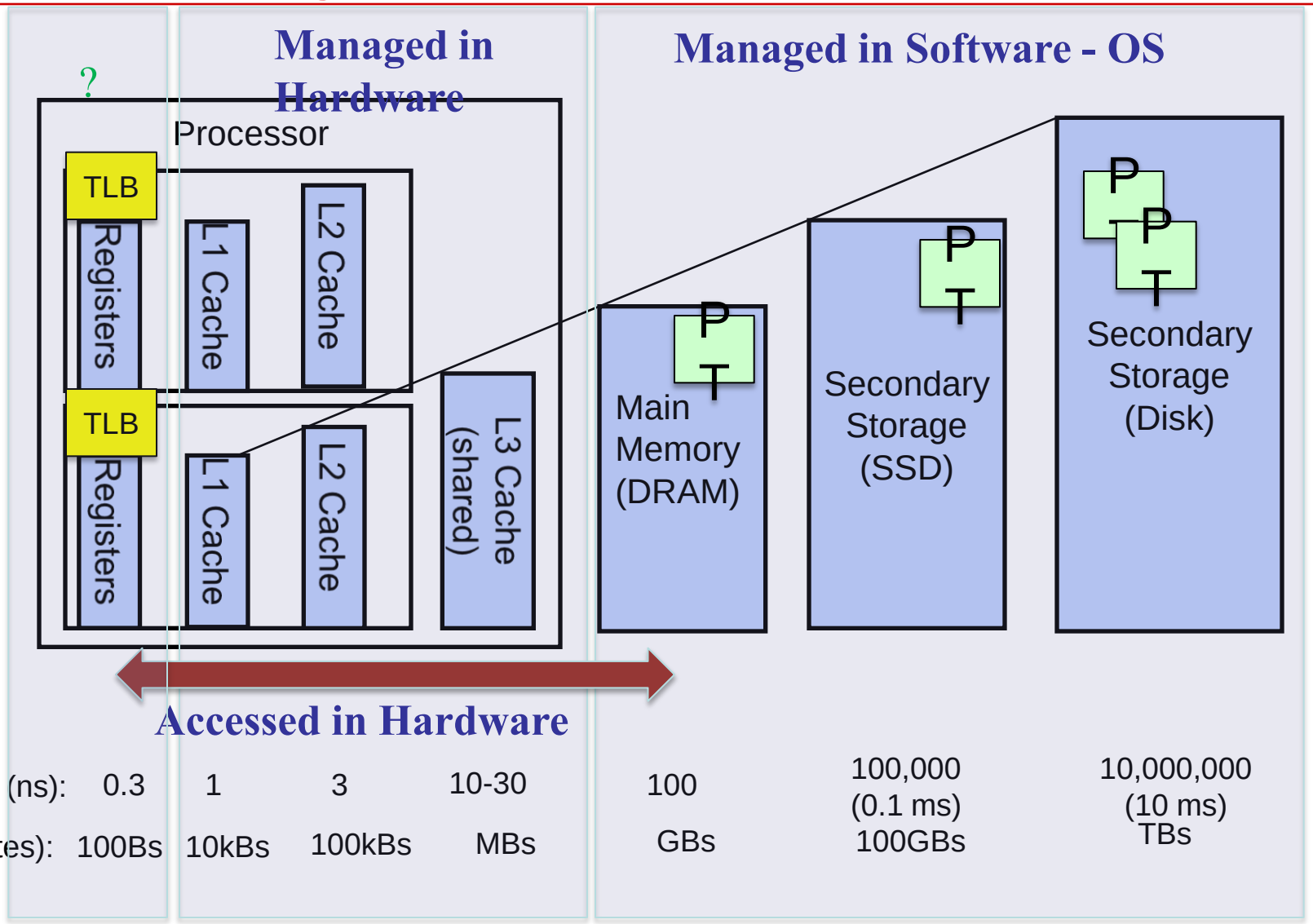
Tradeoffs of Page Size on Performance

- **Impact of page size selection:**
  - TLB hit rate: increase or decrease?
  - Internal fragmentation: increase or decrease?
  - Page table size
    - Increase or decrease?
  - I/O overhead: more useless I/O?
    - Does demand-paging from disk carry more or less overhead?

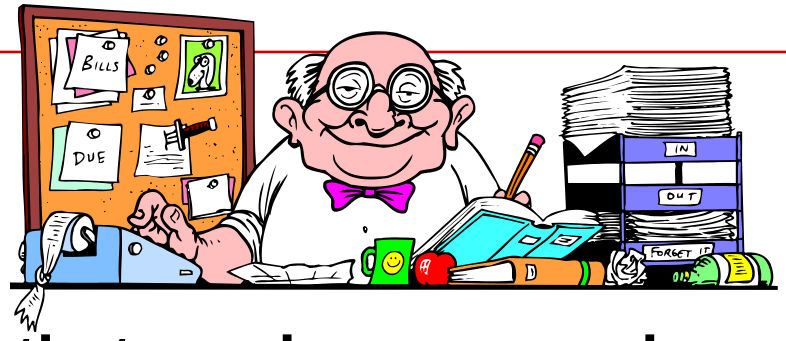
Tradeoffs of Page Size on Performance

- **Impact of page size selection:**
  - TLB hit rate: increase
  - Internal fragmentation: decrease
  - Page table size
    - decrease
  - I/O overhead (useless I/O)
    - Depend on data access pattern of a program

Management & Access to the Memory Hierarchy



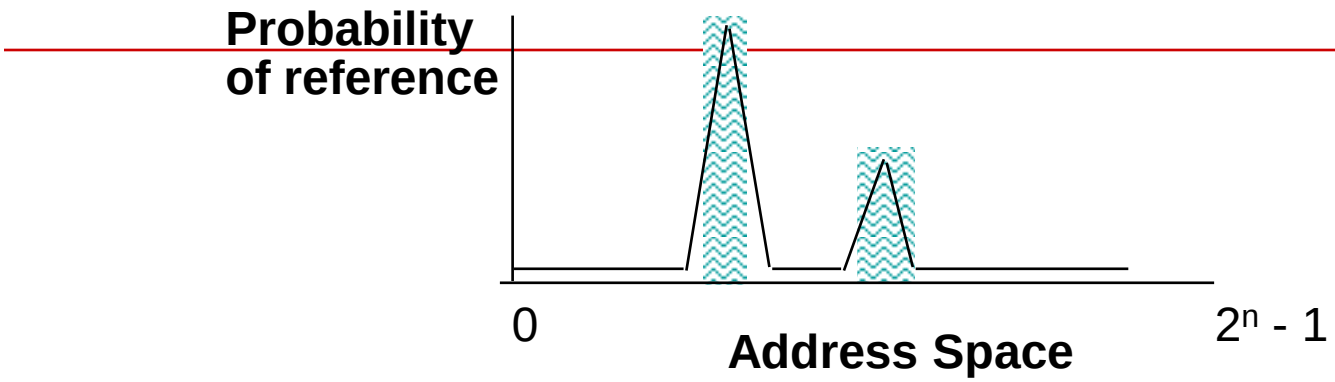
Review of Caching Concept



- **Cache: a repository for copies that can be accessed more quickly than the original**
  - Make frequent case fast and infrequent case less dominant
- **Caching underlies many of the techniques that are used today to make computers fast**
  - Can cache: memory locations, address translations, pages, file blocks, file names, network routes, etc…
- **Only good if:**
  - Frequent case frequent enough and
  - Infrequent case not too expensive
- **Important measure: Average Access time =**

  (Hit Rate x Hit Time) + (Miss Rate x Miss Time)

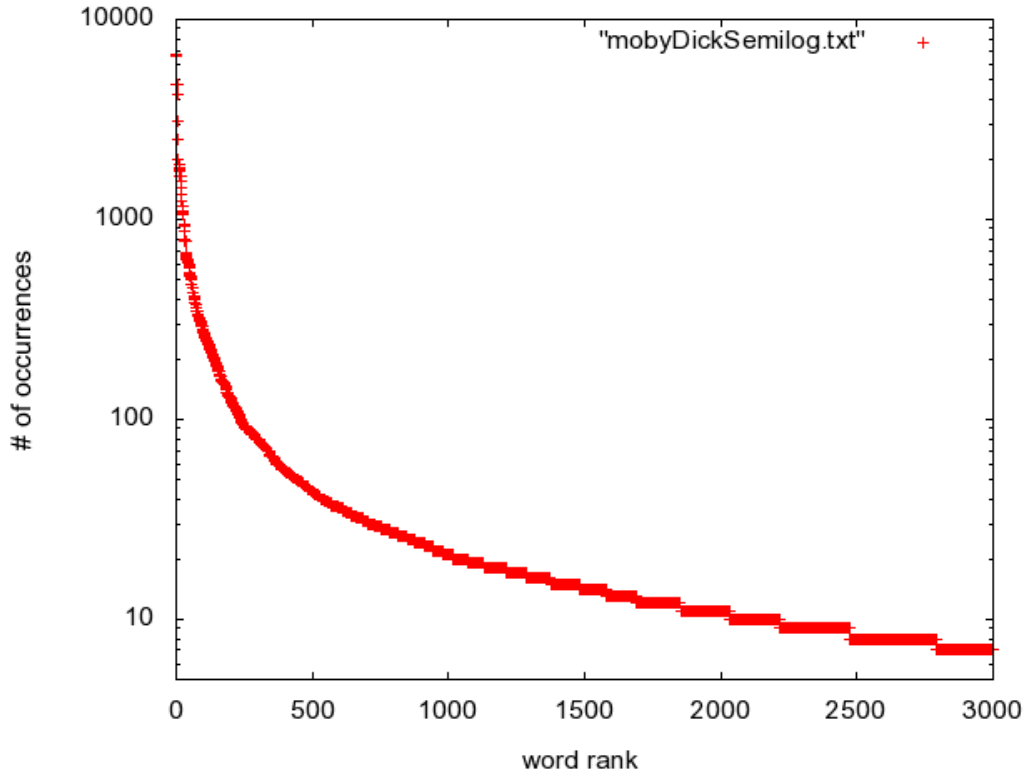
Why Does Caching Help? Locality!



- **Temporal Locality (Locality in Time):**
  - Keep recently accessed data items closer to processor
  - For i = 1 to 100
    - sum =sum*2; // sum is used again and again
- **Spatial Locality (Locality in Space):**
  - Data access is contiguous following its layout.
  - For i = 1 to 100
    - sum = sum + x[i]; // x[1], x[2], x[3] … accessed consecutively

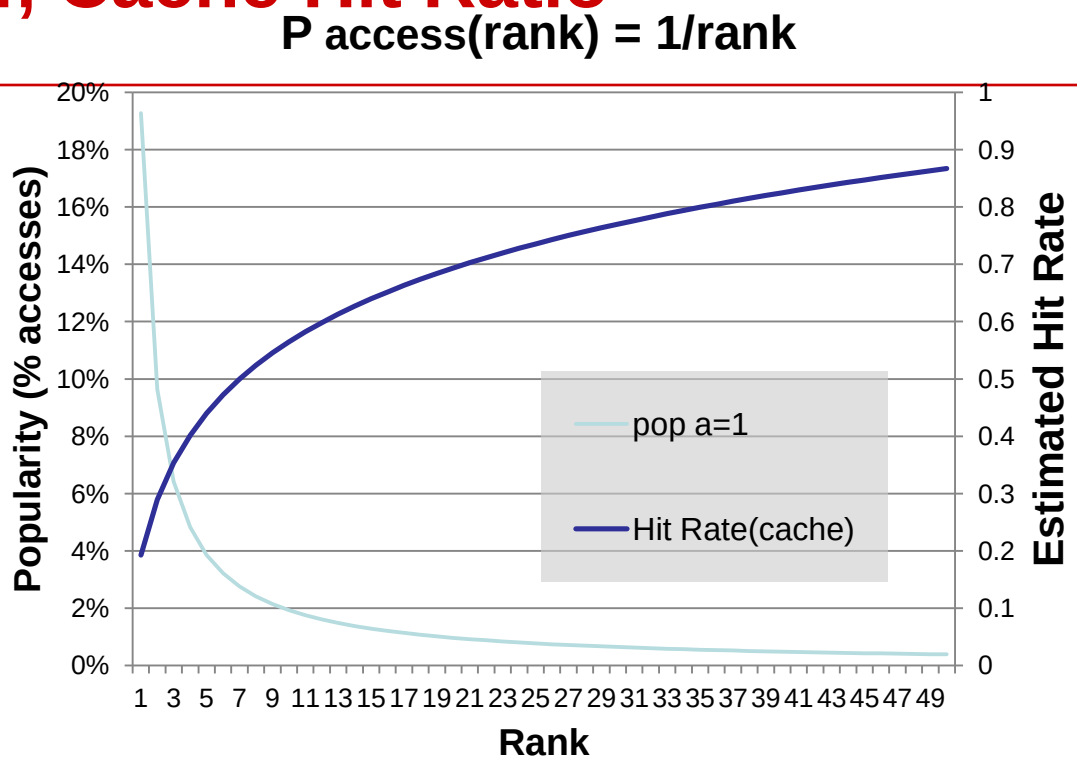
Zipf Distribution on Caching Behavior

- **Caching behavior of many systems are not well characterized by the working set model**
- **An alternative is the Zipf distribution**
  - Popularity ~ $1/k^c$, for k-th most popular item



Popularity of #1 is x
Popularity of #2 is x/2
Popularity of #3 is x/3

Zipf, Cache Hit Ratio



- **Likelihood of accessing item of rank k is ~ $1/k^c$**
- **Many rare items → "heavy tailed" distribution.**
- **Caching popular items can yield a very high cache hit ratio**
  - LRU is a good replacement policy that assumes recent accessed items may be accessed again.

Application Cache that exploits Zipf

- **Cache frequently accessed data in OS**
  - TLB cache. VM. File cache for disk sectors.
- **Cache popular web pages in web servers or in ISP**
  - Popular URLs are accessed again and again
  - Akamai.com caches popular content in all ISP sites.
- **Cache popular queries in Google.com**
  - Popular queries are entered by many users
  - Caching results greatly improve search response time
- **Cache popular nodes in a big social graph**
  - # of followers in Twitter
- **Cache information of popular items sold on Amazon.com**
  - Popular items have more chances to be browsed/purchased

Other benefits of VM: Memory-Mapped Files

- Allow file I/O to be treated as routine memory access by mapping a disk block to a page in memory
- Simplifies file access through direct memory access rather than `read()` `write()` system calls
- File access is managed with demand paging.
- Several processes may map the same file into memory as shared data

Memory Mapped Files

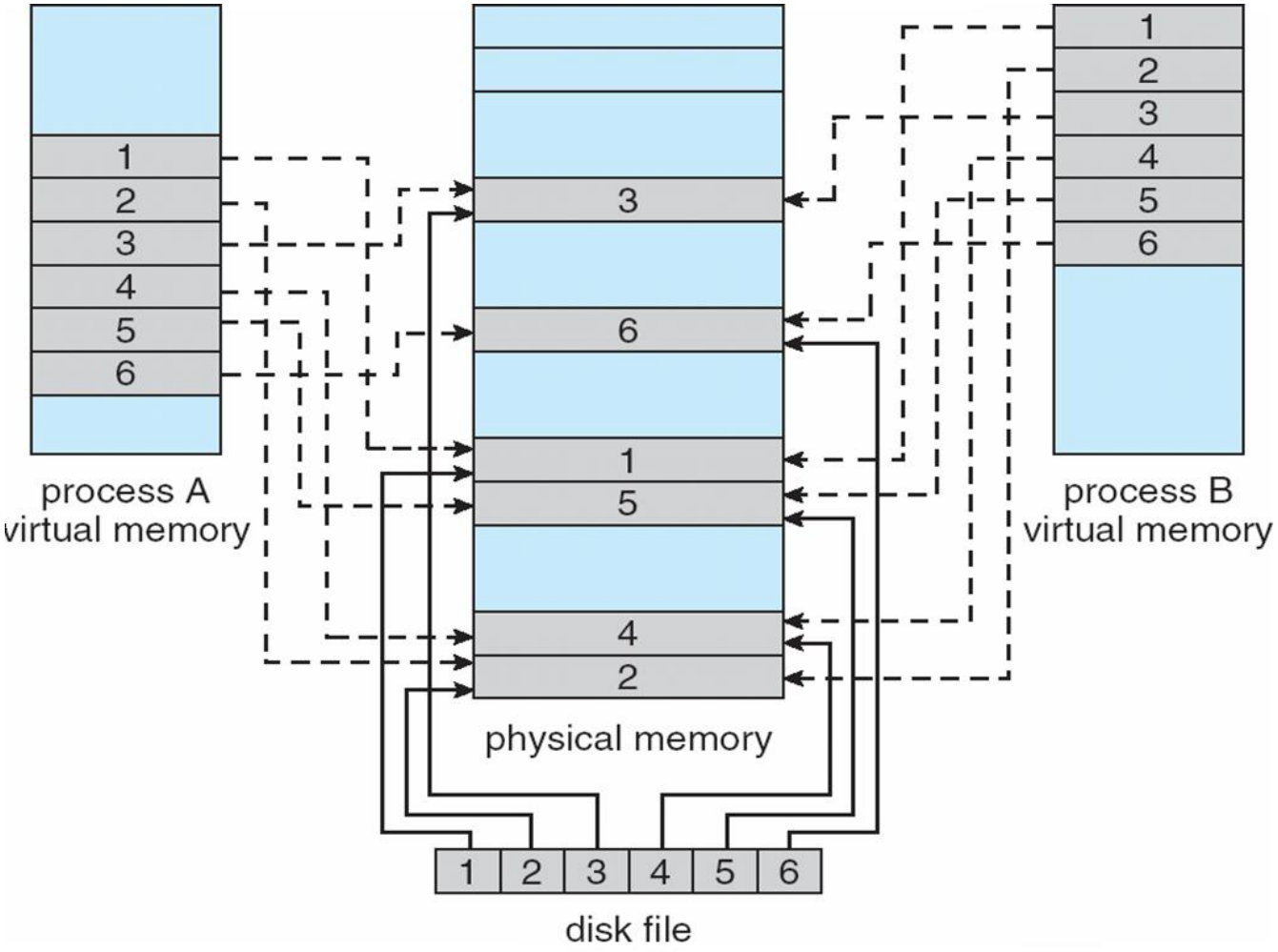
```
MMAP(2)                    BSD System Calls Manual                    MMAP(2)

NAME
     mmap -- allocate memory, or map files or devices into memory

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <sys/mman.h>

     void *
     mmap(void *addr, size_t len, int prot, int flags, int fd,
         off_t offset);

DESCRIPTION
     The mmap() system call causes the pages starting at addr and continuing
     for at most len bytes to be mapped from the object described by fd,
     starting at byte offset offset.  If offset or len is not a multiple of
```

Example of Linux mmap

```
#define NUMINTS  1000
 #define FILESIZE    NUMINTS * sizeof(int)
fd = open(FILEPATH, O_RDWR | O_CREAT |
    O_TRUNC,  0600);
result = lseek(fd, FILESIZE-1, SEEK_SET);
result = write(fd, "", 1);
map = mmap(0, FILESIZE, PROT_READ |
    PROT_WRITE, MAP_SHARED, fd, 0);
for (i = 1; i <=NUMINTS; ++i)  map[i] = 2 * i;
munmap(map, FILESIZE);
close(fd);
```

Direct memory access
→file I/O

Memory Mapped Files



Summary

- **The benefits of a virtual memory system**
  - Virtual memory can be much larger than physical memory
  - Application example: Memory mapped files
- **Demand paging and page-fault handling**
- **Page-replacement algorithms**
  - FIFO. MIN (optimum). LRU.
  - LRU approximation: second-chance
- **Allocation of physical page frames**
- **Performance impact**
  - Relationship of Demand Paging and Thrashing
  - Working set: program access pattern, data locality, page fault.
  - Zipf distribution and caching performance