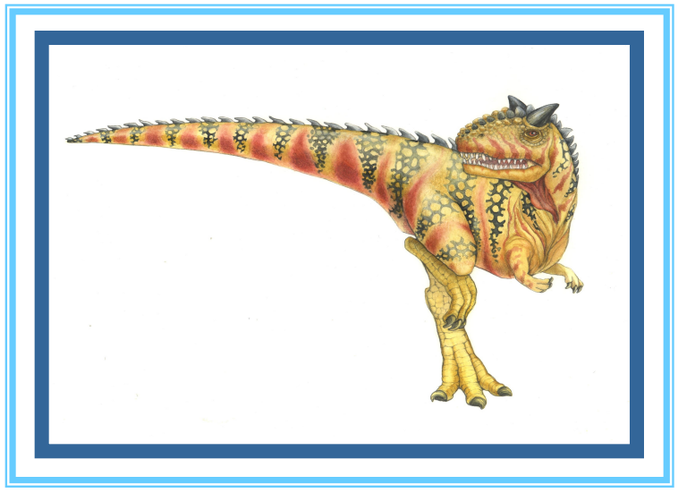
Chapter 9: Virtual Memory



Chapter 9: Virtual Memory

- Background
- Demand Paging
- Page Replacement
- Allocation of Frames
- Thrashing
- Allocating Kernel Memory
- Other Considerations in Virtual Memory
- Operating-System Examples

Objectives

- To describe the benefits of a virtual memory system.
- Illustrate how pages are loaded into memory using demand paging.
- Apply various page replacement algorithms: FIFO, optimal, LRU.
- Describe the working set of a process, and explain how it is related to program locality.
- Discuss how Linux, Windows 10, and Solaris manage virtual memory.

Background

- In chapter 8, we have discussed how the OS manages memory. Memory management has one major goal. What is it?
  - To keep many processes in memory so as to allow for multiprogramming.
- Processes needs to be brought in memory before execution.
- **Virtual memory** is a technique that allows the execution of processes that are not completely in memory.
  - Since physical memory size is a constraint, we may not enough space to bring the **whole** process into the memory.
  - However, we can bring **a part of the process** into memory.
- **Advantage of virtual memory?**
  - Size of physical memory no longer a limiting factor in writing programs.
  - Can write programs that are larger than the physical memory.
  - Hence, main memory is abstracted into an extremely large array of storage.
- As usual, nothing comes for free.
- Virtual memory comes with a cost.
- We will evaluate the cost and complexity of virtual memory throughout this chapter.

Introduction

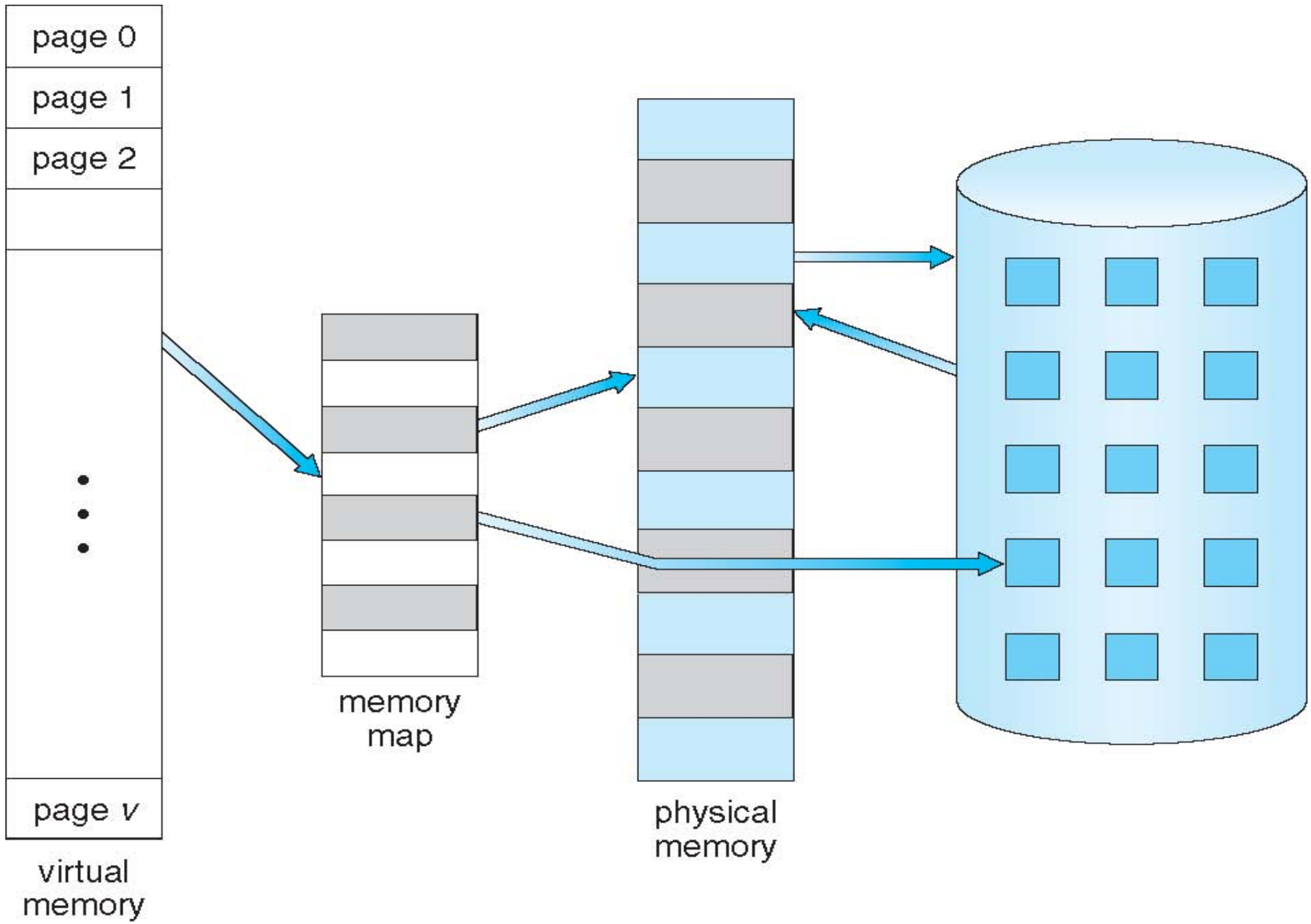
- Need to place process in memory before it can be executed.
- However, examining real programs shows us that:
  - Programs often have code to handle unusual error conditions. Since these errors seldom occur, if ever, this code is almost never executed.
  - Arrays are often allocated more memory than they actually need.
  - Certain options (code) of a program may be used rarely.
- Even if the entire program is needed, it may not all be needed at the same time.
- We need the ability to execute a program that is only partially in memory. Benefits of doing this?
  - Program no longer constrained by the amount of physical memory.
  - Each program takes up lesser physical memory, so more programs can be run at the same time, degree of multiprogramming, throughput, CPU utilization go up (how?).
  - Less I/O needed to load or swap programs into memory, so each user program would run faster.

Recap.

- **Virtual memory** – separation of user logical memory from physical memory.
  - Only part of the program needs to be in memory for execution.
  - Logical address space can therefore be much larger than physical address space.
  - Allows physical address space to be shared by more processes.
  - More programs running concurrently.
  - Less I/O needed to load or swap processes.

- Virtual memory can be implemented via:
  - Demand paging.

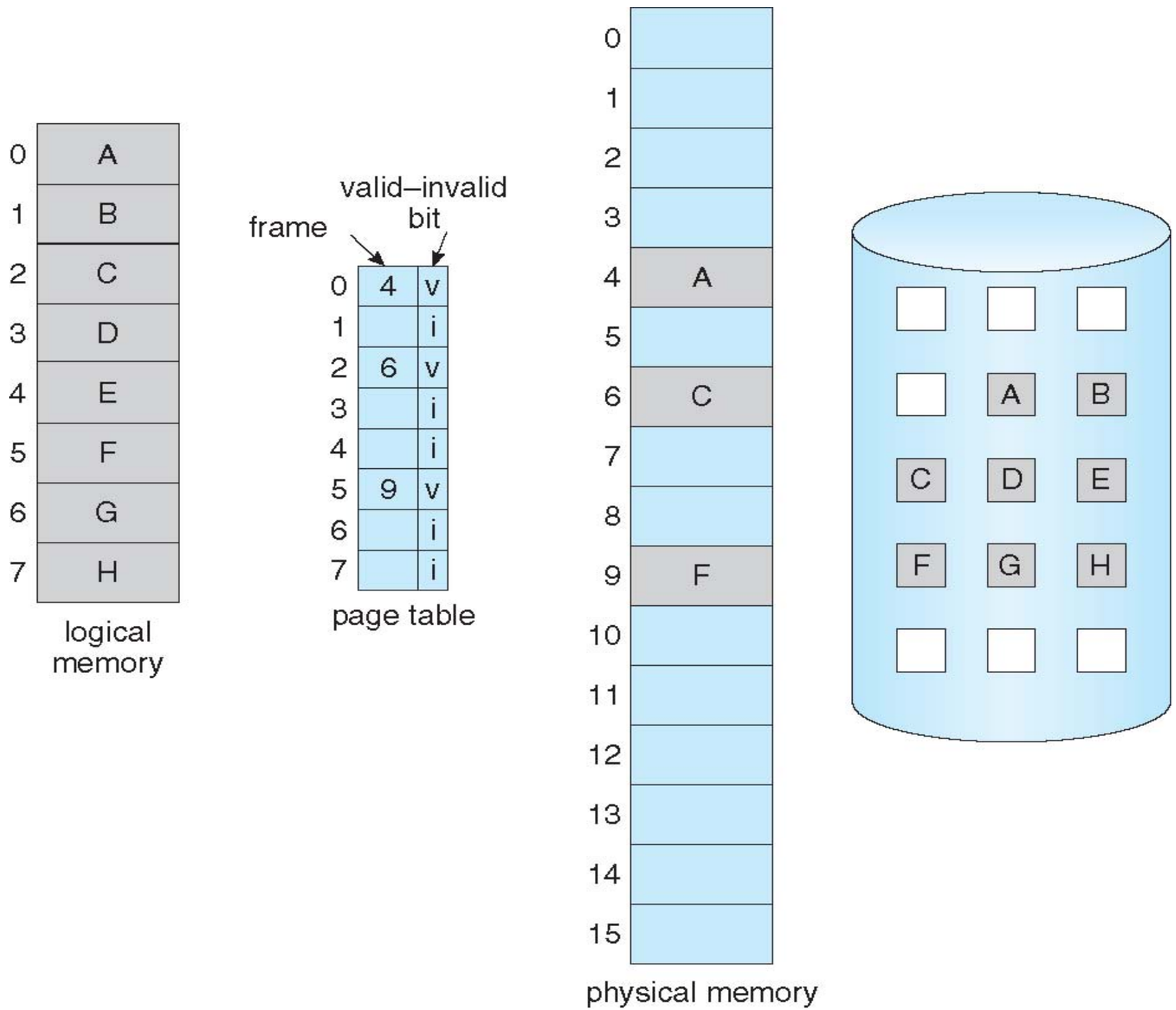
Virtual Memory >>>> Physical Memory



Demand Paging

- As stated earlier, it may not be necessary to load entire process into memory before execution.
  - May not initially need the entire process in memory.
  - For example, a program that accepts user option and accordingly executes certain code.
  - Not wise to load all the code (for all the options) as initially, only one option is selected.
- A strategy is to load pages only as they are needed during program execution.
  - Pages that are not needed are never loaded (swapped) into memory.
- This is called **demand paging** and is used in virtual memory systems.
- The picture in the next slide illustrates the technique.
- Need some form of hardware support for this.
  - Need to distinguish which pages are in memory and which pages are in the disk.
- Can associate a valid-invalid bit with each page.
  - Valid → page is in memory
  - Invalid → page is in disk.

Page Table, some pages not in main memory



Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated
  (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

Frame #	valid-invalid bit
	v
	v
	v
	i
….	
	i
	i

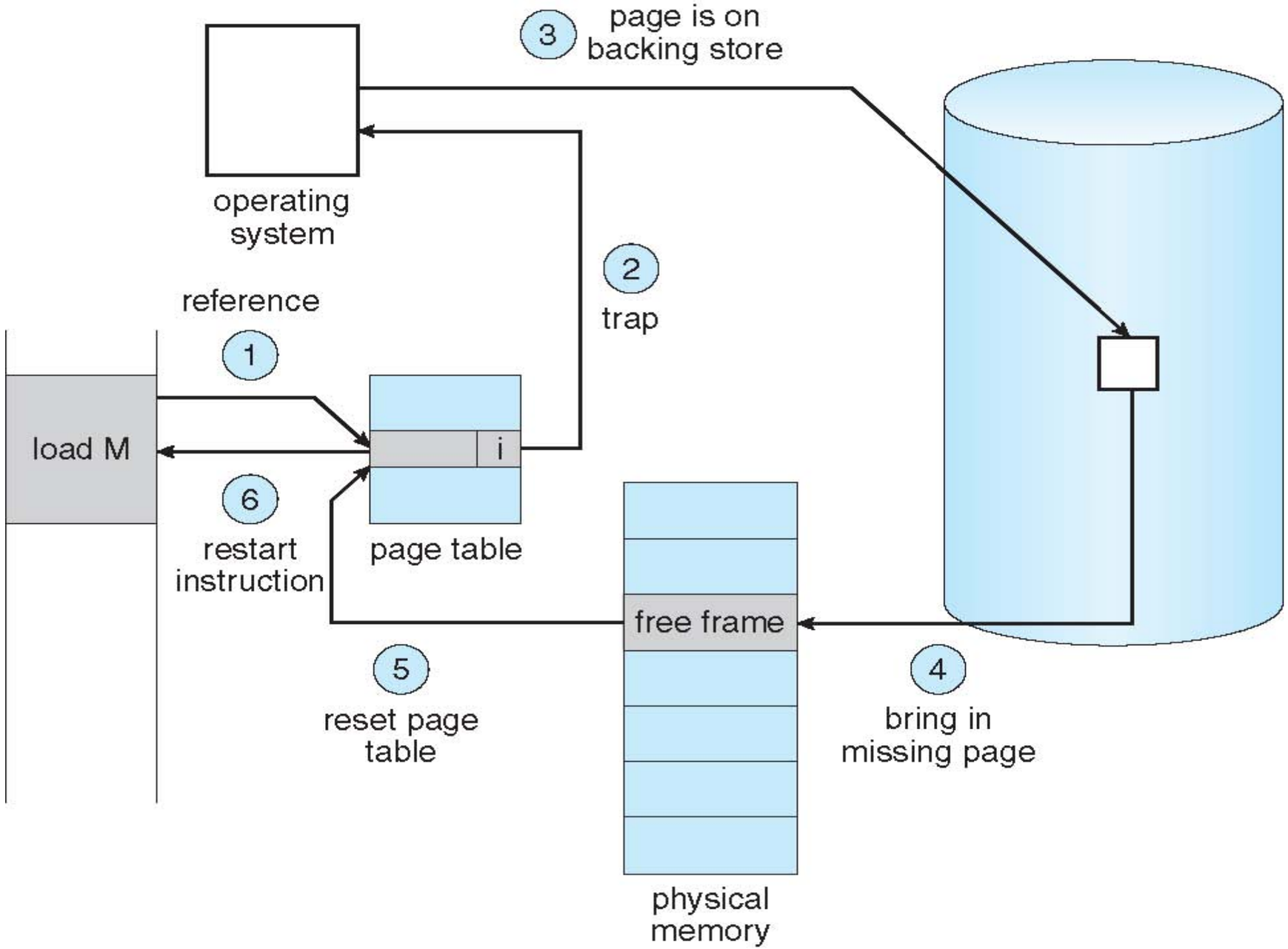
page table

- During address translation, if valid–invalid bit in page table entry is **I** ⇒ **page fault.**

Page Fault

- What happens if the process tries to access a page that was not brought into memory?
  - Access to page marked invalid causes a page fault (a trap to the OS).
  - Paging hardware, in translating the address, notices that the page is not in memory and traps to the OS.
  - Now, the page needs to be brought into memory.
- How do we handle a page fault?
  1. First check whether the page reference was valid (i.e. in the process address space). How?
  2. If reference is valid, need to bring the page in from disk → memory.
  3. Need to find a free frame (take one from the free frame list).
  4. Schedule disk operation to read the desired page into the newly allocated frame.
  5. When disk operation is complete, modify the page table to indicate that the page is now in memory (set the valid bit).
  6. Restart the instruction that caused the page fault. The process can now access the page as if it was always in memory.
- Initially, we start process execution with *no* pages in memory.
- Pages are brought in as desired.

Steps in Handling a Page Fault



Performance of Demand Paging

- Demand paging can affect the computing system performance.
  - As disk operations are involved.
  - These are on the slower side.
- Let's define a quantity called – effective mean access time. It is given by the following equation:
- Effective mean access time (*emat*) = (1 – *p*) x *ma* + (*p* x page fault time).
  - *p* denotes the probability of a page fault ($0 \leq p \leq 1$).
  - What would p = 0 mean?
    - No page faults (all pages required in memory).
  - What would p = 1 mean?
    - Every page reference leads to a page fault (no page in memory).
  - *ma* denotes the memory access time.
- The performance bottleneck in the above equation?
- The page fault time.
- As we will see in the next slide, the page fault time comprises of the time taken for a number of operations.

Performance of Demand Paging

- **A page fault launches the following sequence of events:**

1. Trap to the operating system.
2. Save the user registers and process state.
3. Determine that the interrupt was a page fault.
4. Check that the page reference was legal and determine the location of the page on the disk.
5. Issue a read from the disk to a free frame:
   a) Wait in a queue for this device until the read request is serviced.
   b) Wait for the device seek and/or latency time.
   c) Begin the transfer of the page to a free frame.
6. While waiting, allocate the CPU to some other user.
7. Receive an interrupt from the disk I/O subsystem (I/O completed).
8. Save the registers and process state for the other user.
9. Determine that the interrupt was from the disk.
10. Correct the page table and other tables to show page is now in memory.
11. Wait for the CPU to be allocated to this process again.
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction.

Demand Paging Example

- Lets say the memory access time (*ma*) = 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 = 200 + (7999800/1000) = 200 + 7999.8 = 8199.8 nanoseconds = 8.2 microseconds. How much of a slowdown is this?

  - This is a slowdown by a factor of 40!
- If we want performance degradation < 10 %, then, we have
  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 memory accesses!

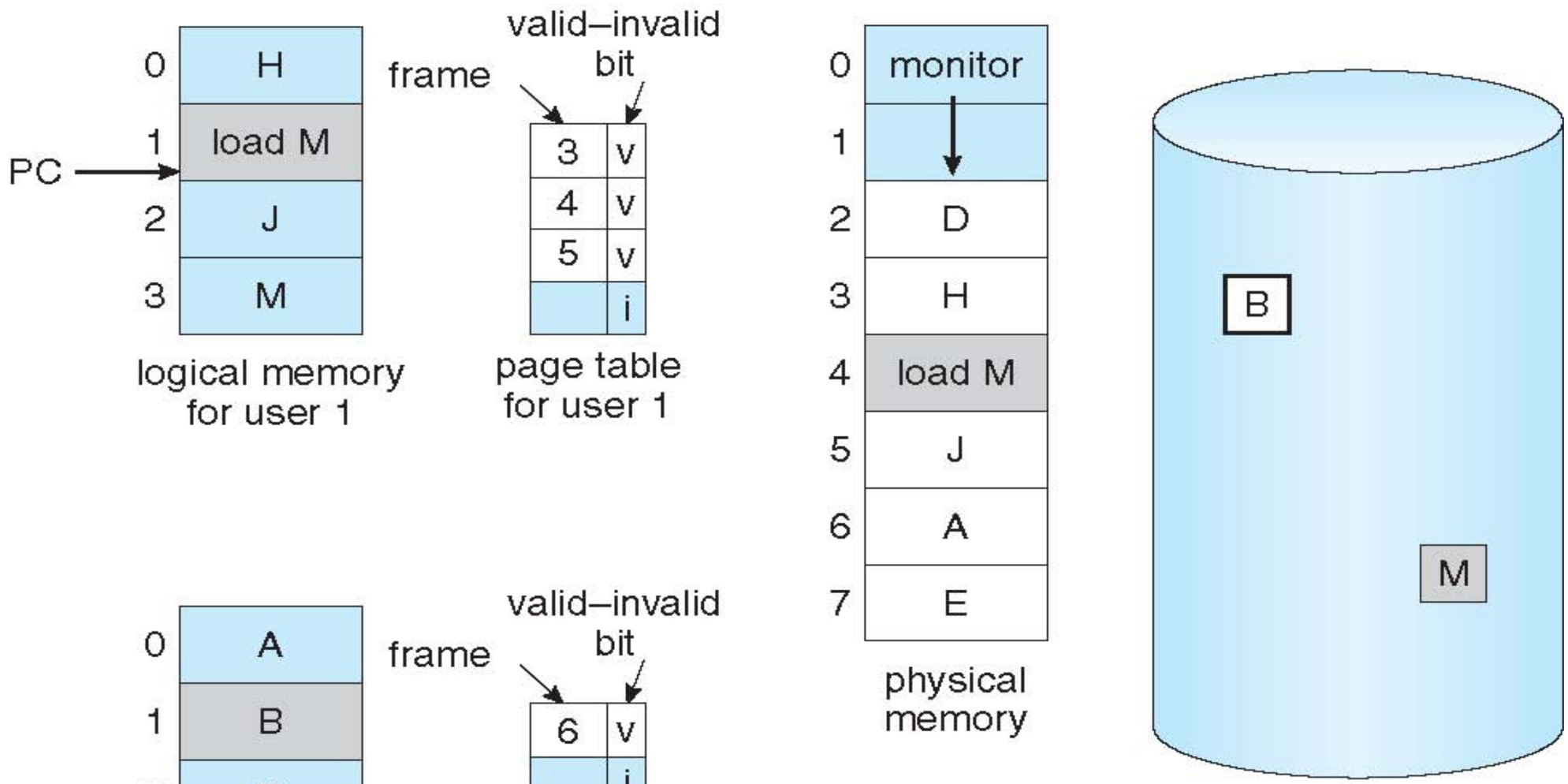
Demand Paging Examples.

- Assume we have a demand-paged memory. The page table is held in registers. It takes 8 milliseconds to service a page fault if an empty page is available or the replaced page is not modified, and 20 milliseconds if the replaced page is modified. Memory access time is 100 nanoseconds. Assume that the page to be replaced is modified 70 percent of the time. What is the maximum acceptable page-fault rate for an effective access time of no more than 200 nanoseconds?
- It has been observed that the number of instructions executed between page faults is directly proportional to the number of page frames allocated to a program. If the available memory is doubled, the mean interval between page faults is also doubled. Suppose that a normal instruction takes 1 microsecond, but if a page fault occurs, it takes 2001 microseconds (i.e. 2 milliseconds to handle the page fault). If a program runs for 60 seconds, during which time it gets 15,000 page faults, how long would it run for if twice as memory was available?

What Happens if There is no Free Frame?

- Page fault, but no free frame available.
- All available frames used up by process pages.
- Frames also in demand from the kernel, I/O buffers, etc
- **Page replacement** – find some page in memory, but not really in use, page it out.
  - Algorithm – **terminate process in question**? **swap the process out**? **replace a suitable page**?
  - Performance – want an algorithm which will result in minimum number of page faults.
- **Replace heavily used page or lightly used page?**

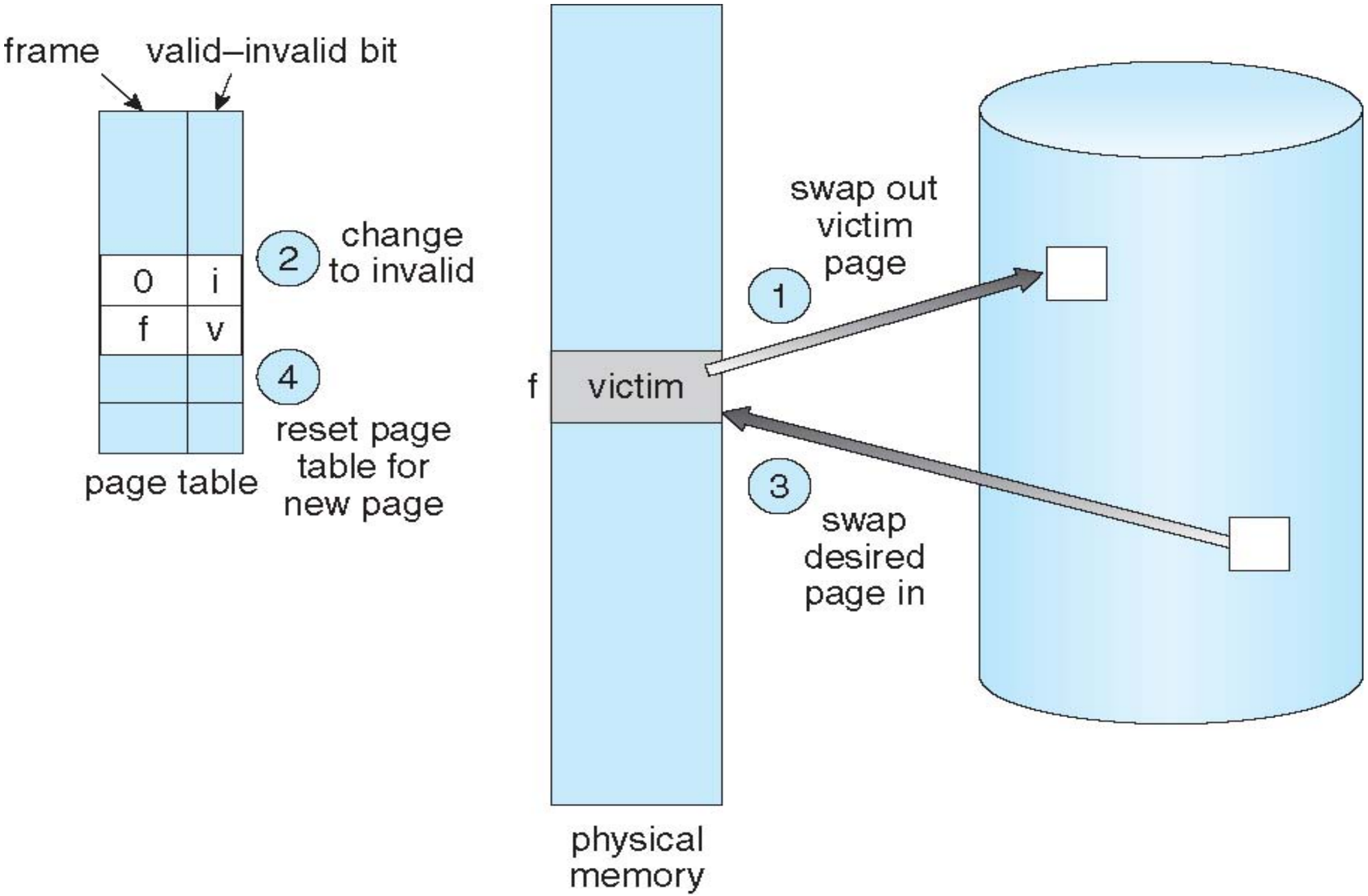
Need For Page Replacement



Page Replacement

- Now, the page fault routine needs to include the page replacement as well.
  - Find location of desired page on disk.
  - Find a free frame:
    - If there is a free frame, use it.
    - If not, use a page-replacement algorithm to select a victim frame.
    - Write victim frame to disk; change page table.
  - Read the desired page into newly freed frame; change page table.
  - Restart user process.
- Note that if no frames are free, two page transfers (one out and one in) are required.
- This is a major overhead.
- We can perform an optimization to reduce this overhead.

Page Replacement



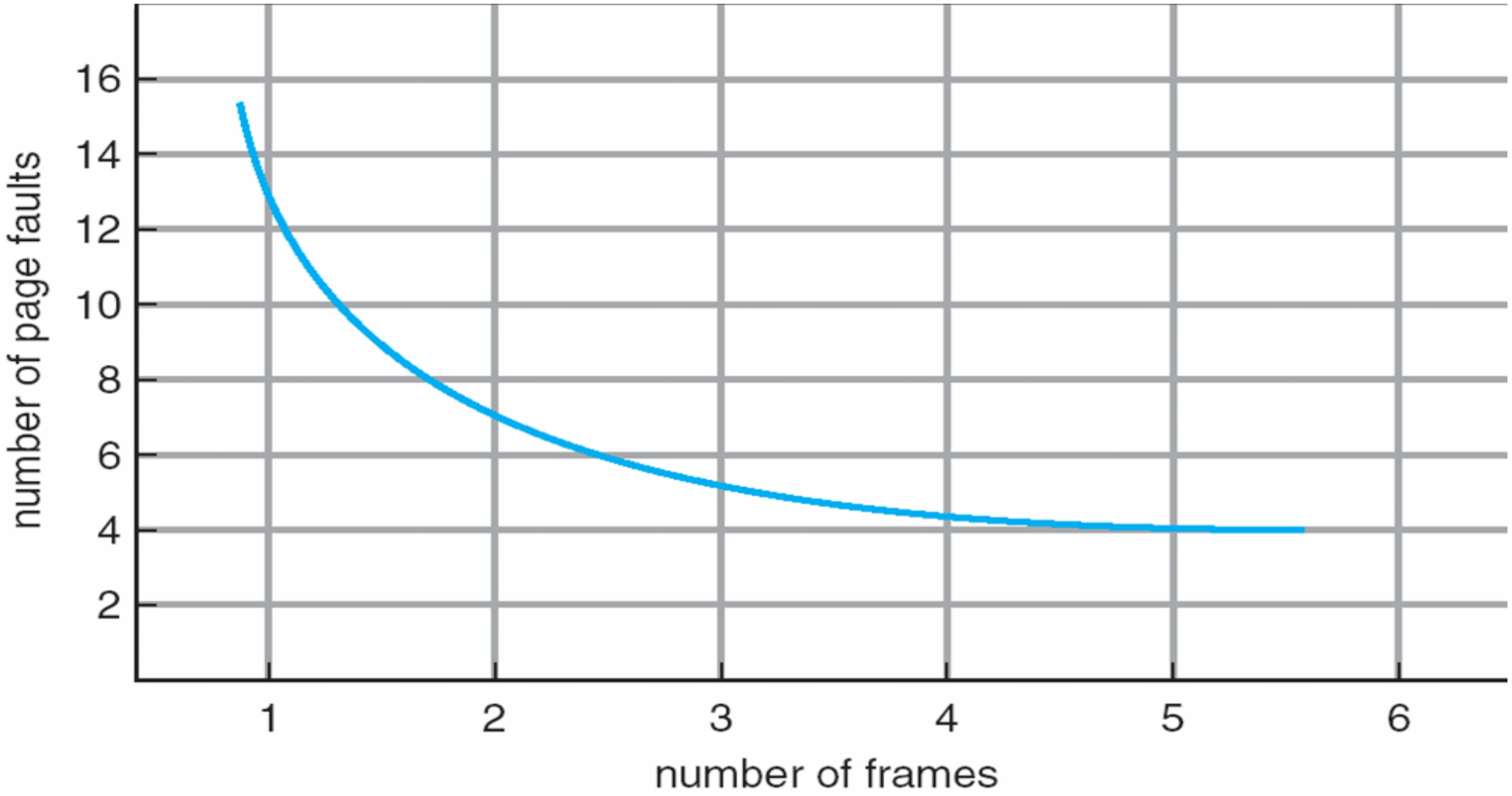
Page Replacement

- Optimization is based on this observation:
  - We read in a page from the disk and then later write it back to the disk.
  - If the page has not been modified while it was in the memory, do we need to write it back to disk?
    - No, as the disk already has the correct copy.
- This optimization is implemented by associating a modify bit (or dirty bit) with each page (1 bit in PTE (page table entry) is reserved for this).
  - This bit is set when something is written to a page, indicating that the page has been modified since it was read from disk.
  - We write the page back to disk only if its modified bit has been set (as the memory and disk have different versions of the page).
- This technique can be applied to read-only pages as well.
- Advantage: reduce the I/O time by half, if the page has not been modified.
- Let us now solve a problem.

Page Replacement

- We evaluate a page replacement algorithm by running it on a particular string of memory references and computing the number of page faults.
  - Low or high number of page faults is desired?
- This string is called a reference string.
- We make two observations regarding the reference string:
  - For simplicity, we will focus on the page number and not the actual memory address.
  - If we have a reference to page *p*, then any references to *p* that immediately follows will never cause a page fault.
- The address sequence - 0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105 becomes:
- 1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1.
- How many page faults if we allocate:
  - 3 frames?
  - 1 frame?

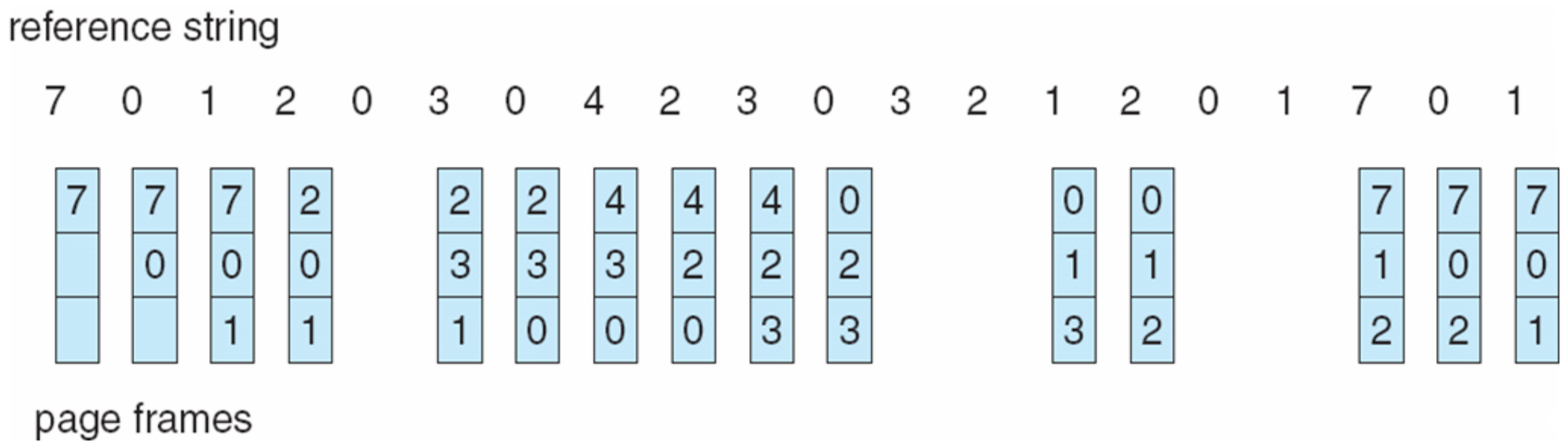
Page Faults Versus Number of Frames



FIFO Page Replacement

- Simplest page replacement algorithm – First In First Out.
- How does this work?
- Idea – "**always replace the oldest page**".
- We can create a FIFO queue to hold all pages in memory.
- Replace the page at the head of the queue.
- When a page is brought into memory, insert it at the tail of the queue.
- Let's see the algorithm in action on the following reference string:
  - 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1.
- Assume that we have three frames.
- Performance is, unfortunately, not very good → high number of page faults.
  - No priority for pages.
  - Page replaced may be heavily used or very lightly used.

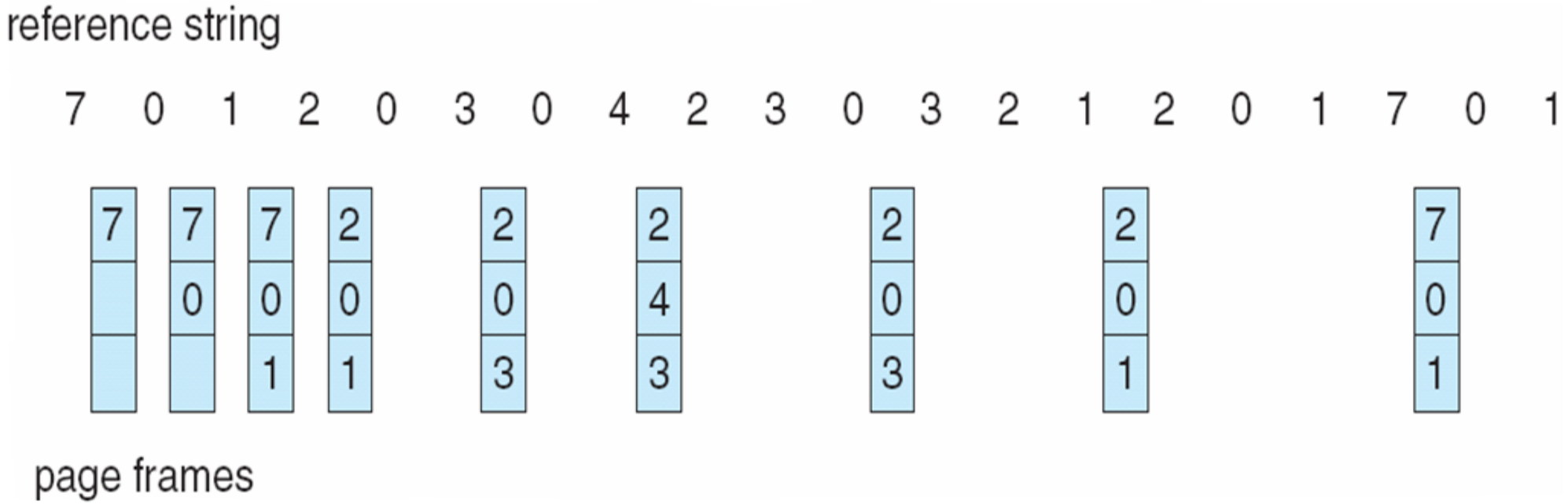
FIFO Page Replacement



Optimal Algorithm

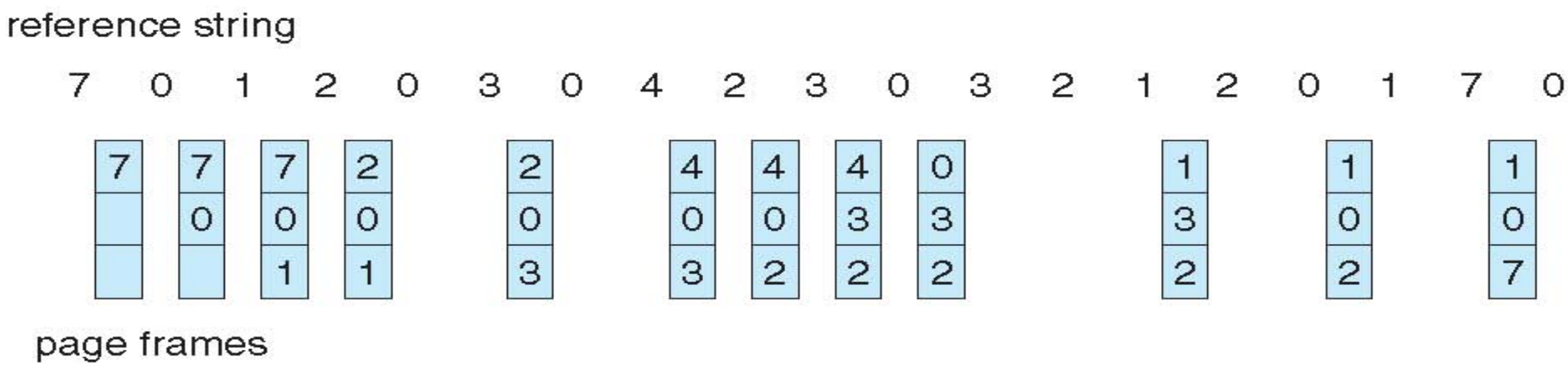
- **What could the optimal algorithm be?**
- Idea - "**replace page that will not be used for longest period of time**".
  - *Page referenced after 10 instructions vs. page referenced after 1 million instructions.*
- Need to "**look ahead**" in the reference string.
- This algorithm is optimal → it offers the lowest page fault rate for a given reference string and set of available frames.
- For the sample string, we get 9 page faults (vs. 15 from FIFO).
- However, a drawback is that it is difficult to predict future page requests.
  - Similar situation was encountered in the Shortest Job First algorithm.
- This makes the algorithm virtually impossible to implement.
- Can be used for comparison purposes.
  - For example, algorithm X may not be optimal, but it is within 12.3 % of the optimal algorithm.

Optimal Page Replacement



Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Idea – **"replace page that has not been used in the most amount of time".**
- Associate time of last use with each page



- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?

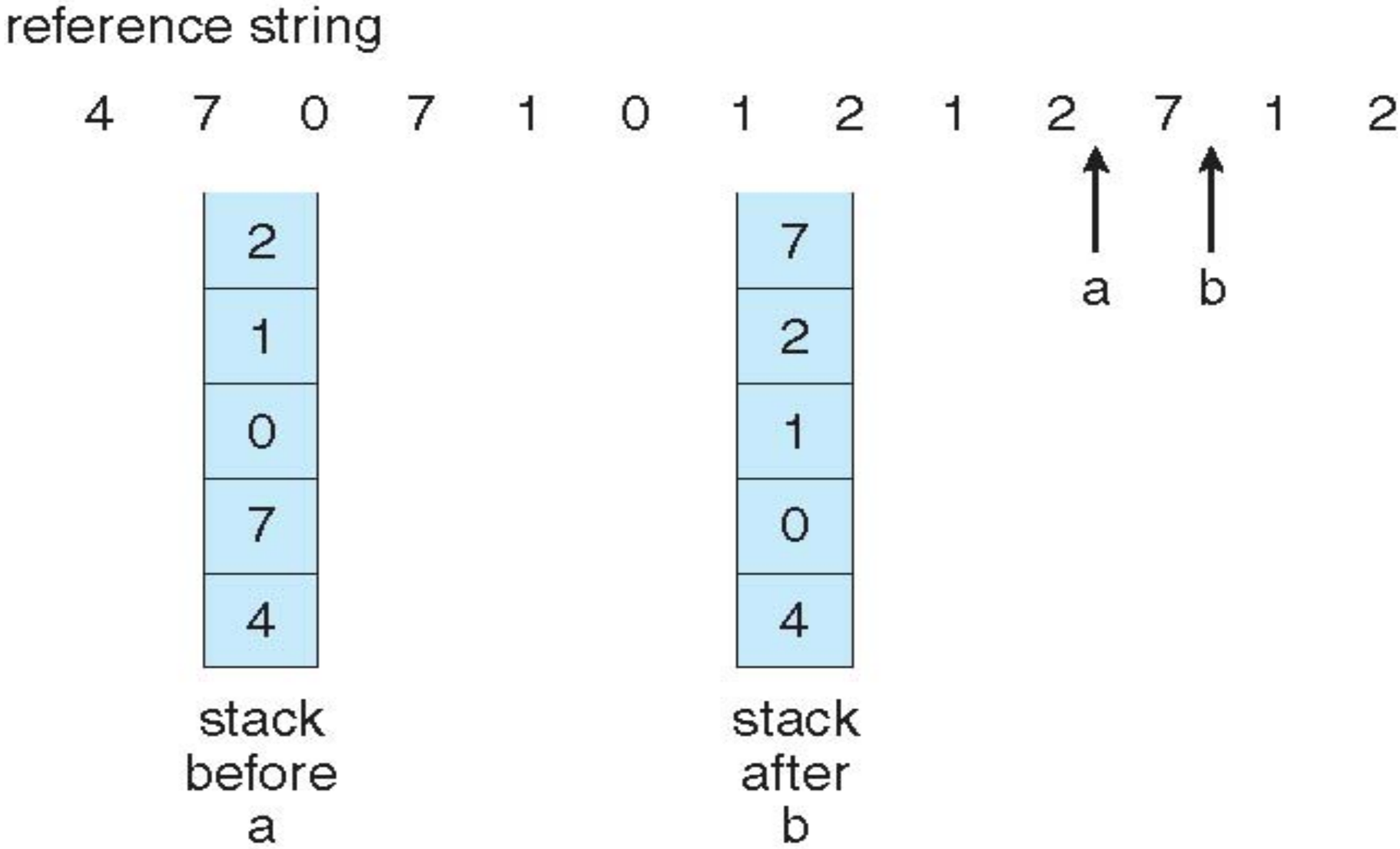
LRU Implementation

- Instead of looking at the future, look at the past (like FIFO).
- The goal here is to determine an order of frames defined by the time of last use.
- Idea – "**least recently used frame is selected for eviction".**
- Counter based implementation.
  - Each page-table entry is associated with a time-of-use field.
  - Also need a clock counter.
  - Counter is incremented for every memory reference.
  - Whenever a reference to a page is made, the contents of the counter are copied to the time-of-use field in the page-table entry.
  - This way, we always have a time of the last reference to each page.
  - Which page do we pick for eviction?
  - Page with the smallest time-of-use value is chosen for eviction.
  - Need to search the page table for the page with the smallest time-of-use value.
  - Maintain time-of-use when page tables are changed (due to scheduling).

LRU Implementation

- Stack based implementation
  - We can keep a stack of page numbers.
  - When ever a page is referenced, it is put at the top of the stack.
    - If the page is referenced for the first time, it is added to the top.
    - Else, referenced page is moved to the top (as it has been referenced before, it is already in the stack).
  - **The most recently used page is at the top.**
  - **The least recently used page is at the bottom**
- Unlike the counter implementation, no search operation. Why?
  - Can evict the page that is at the bottom.
- Although, the insertion updates have on overhead.

Stack based LRU Implementation



LRU Approximation

- Note that both the counter and stack based implementations of the LRU algorithm require significant hardware assistance, which few machines may have.
- How frequently do we need to update the counter field or the stack?
  - For every memory reference (would slow down processes considerably)!
- As it is expensive to support “pure LRU”, the OS resorts to LRU Approximation Page Replacement.
- Idea – “have a reference bit associated with each page”.
- When a page is referenced, this bit is set to 1.
  - Reference could be a read or a write.
- Initially, all reference bits are 0.
- As a process executes, the bit is set to 1 for the appropriate pages.
- After some time, we can determine which pages have been used and which pages have not.
- However, note that we do not know the order in which they have been used.
- We can still use this information as basis of LRU approx. algorithms.

Additional Reference Bits Algorithm

- **How can we gain additional ordering information for reference bits?**
- Can gain additional ordering info by recording the reference bits at regular intervals (say, every 100 milliseconds).
- Keep 1 byte (8 bits) for each page in a table.
- At regular intervals, the OS shifts the reference bit for each page into the higher order bit of its byte, shifting the other bits right by 1 bit, discarding the lowest order bit.
- The byte contains a history of page use for the last eight time periods.
  - 00000000 → page not used for last eight periods.
  - 11111111 → page has been used in every period.
  - 11000100 (196_{10}) has been used more recently than 01110111 (119_{10}).
  - Converting to decimal, the page with the lowest number is the LRU page, and can be replaced.

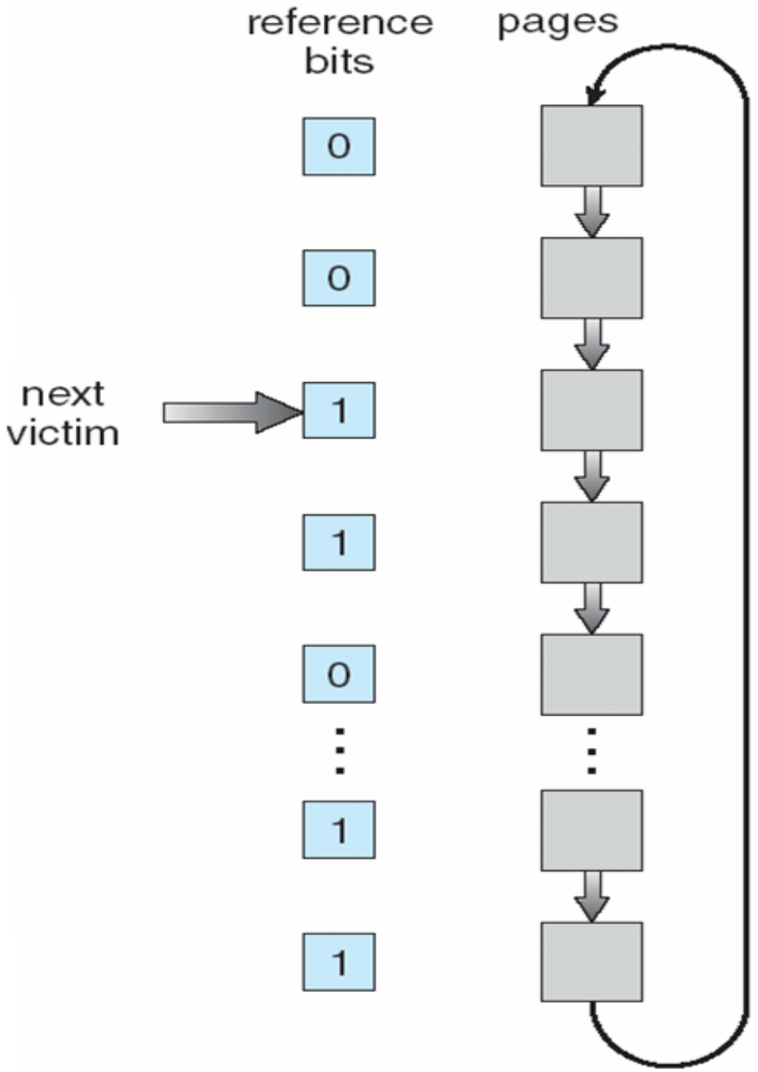
Additional Reference Bits Algorithm

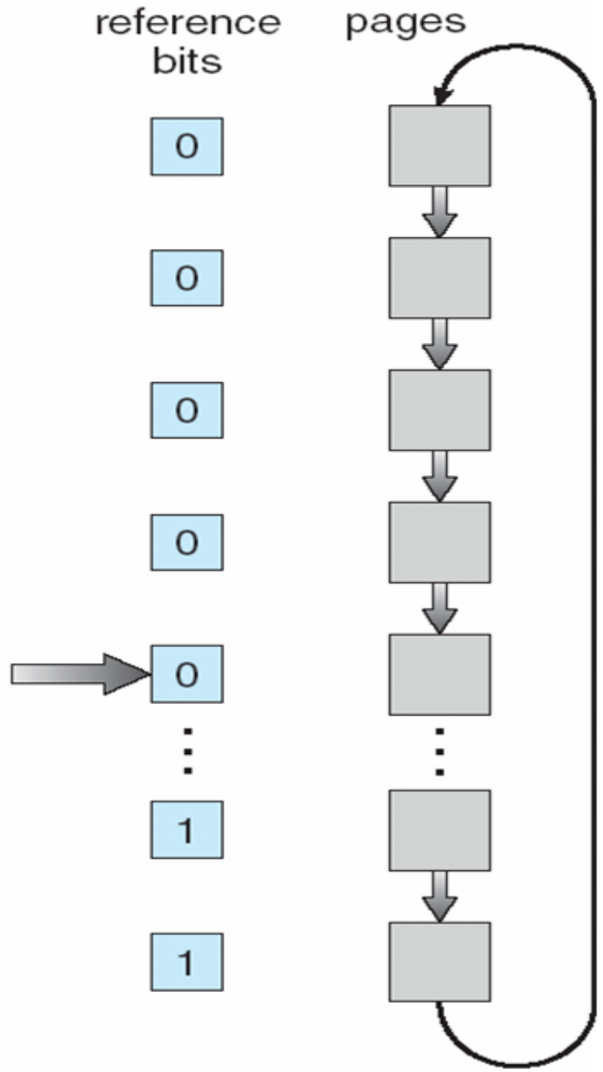
Page	R bits for pages 0-5, clock tick 0	R bits for pages 0-5, clock tick 1	R bits for pages 0-5, clock tick 2	R bits for pages 0-5, clock tick 3	R bits for pages 0-5, clock tick 4
	1 0 1 0 1 1	1 1 0 0 1 0	1 1 0 1 0 1	1 0 0 0 1 0	0 1 1 0 0 0
0	10000000	11000000	11100000	11110000	01111000
1	00000000	10000000	11000000	01100000	10110000
2	10000000	01000000	00100000	00100000	10001000
3	00000000	00000000	10000000	01000000	00100000
4	10000000	11000000	01100000	10110000	01011000
5	10000000	01000000	10100000	01010000	00101000
	(a)	(b)	(c)	(d)	(e)

Second Chance Algorithm

- Extends FIFO to add information about page usage.
- When a page is selected, inspect its reference bit.
  - If bit value is 0, proceed to replace the page (has not been referenced).
    - New page is put in its position.
  - If bit value is 1, give the page a second chance and proceed to the next page.
    - Set the bit value to 0 and set the arrival time to current time.
- Page that is given a second chance will not be replaced until all other pages have been replaced (or given second chances).
- Page sets its r-bit to 1 consistently will never be replaced.
- This algorithm is also called a clock page-replacement algorithm.
- Can be implemented as a circular queue.

Second-Chance (clock) Page-Replacement Algorithm





Thrashing

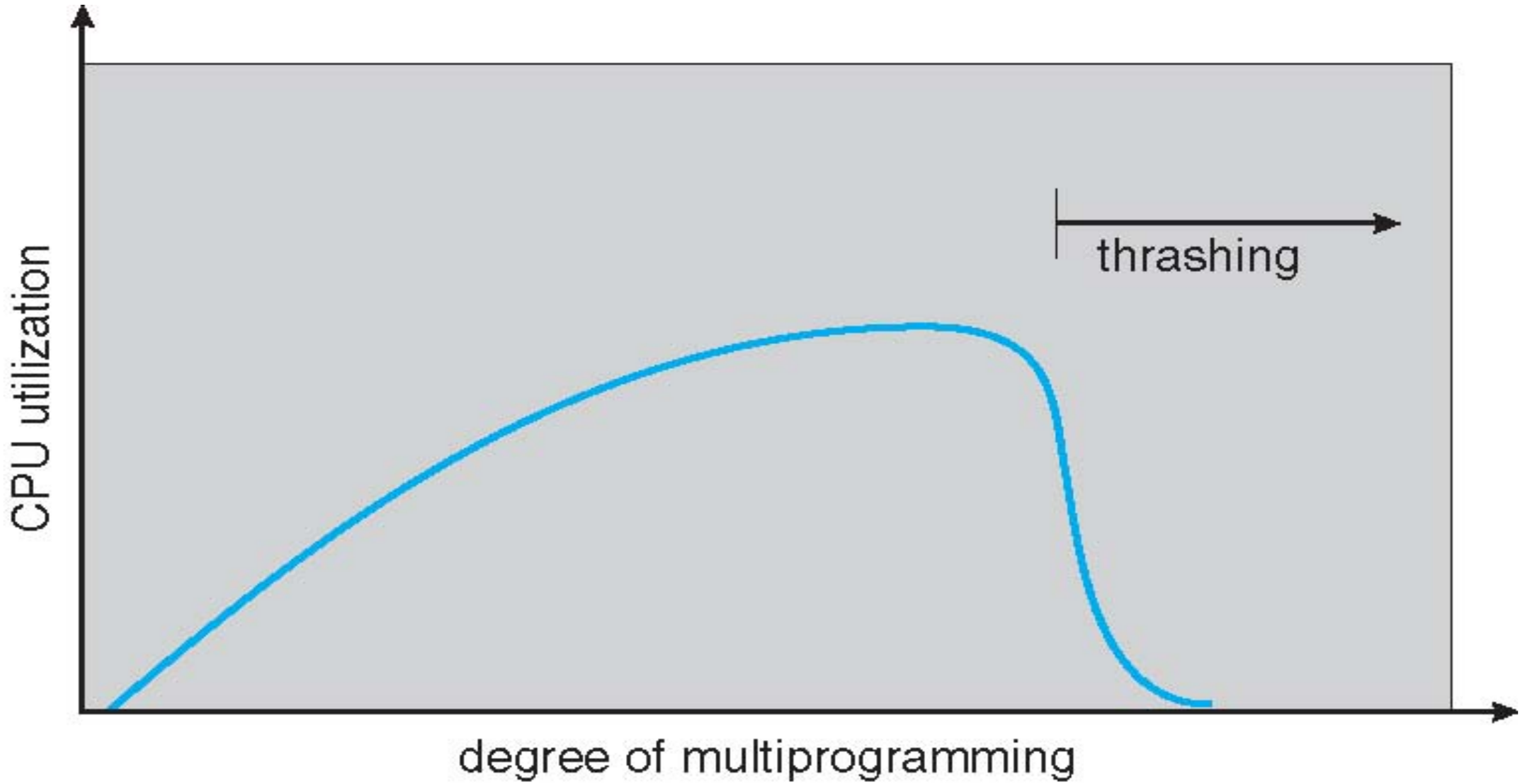
- How many frames is enough for a process?
  - What is number is too low, too high?
- Think about what would happen to a process that does not have “enough” frames.
- If it does not have enough frames to support all its active pages, it will quickly page fault.
  - Let’s say that a process has 10 active pages, but is allocated only 5 frames in the memory.
- Page fault follows. Time to remove a resident page. Which one?
- Note that all its pages are in active use.
- So, the page it replaces will be needed again very soon.
- Consequently, it page faults again, and again.
  - Basically, it replaces pages that it must bring back immediately.
- This high paging activity is called thrashing.
  - Large chunk of time is spent on paging as opposed to executing.
- **Thrashing is not good!**

Thrashing

- Thrashing results in severe performance problems.
- Lets see a possible chain of events.

1. Thrashing causes rapid reduction in CPU utilization.
2. If CPU utilization is too low, OS > degree of multiprogramming.
3. A global page replacement algorithm is used and it replaces pages that belong to other processes.
4. Assume that such a process starts a new phase of execution and needs more frames.
5. If not available, it would start faulting and taking pages away from other processes.
6. This would lead to the other processes faulting as well, taking frames from other processes.
7. All faulting processes need to use a paging queue to access the disk. This queue becomes over populated.
8. Hence, the ready queue empties.
9. Thrashing leads to a severe reduction in the CPU utilization.

Thrashing (Cont.)



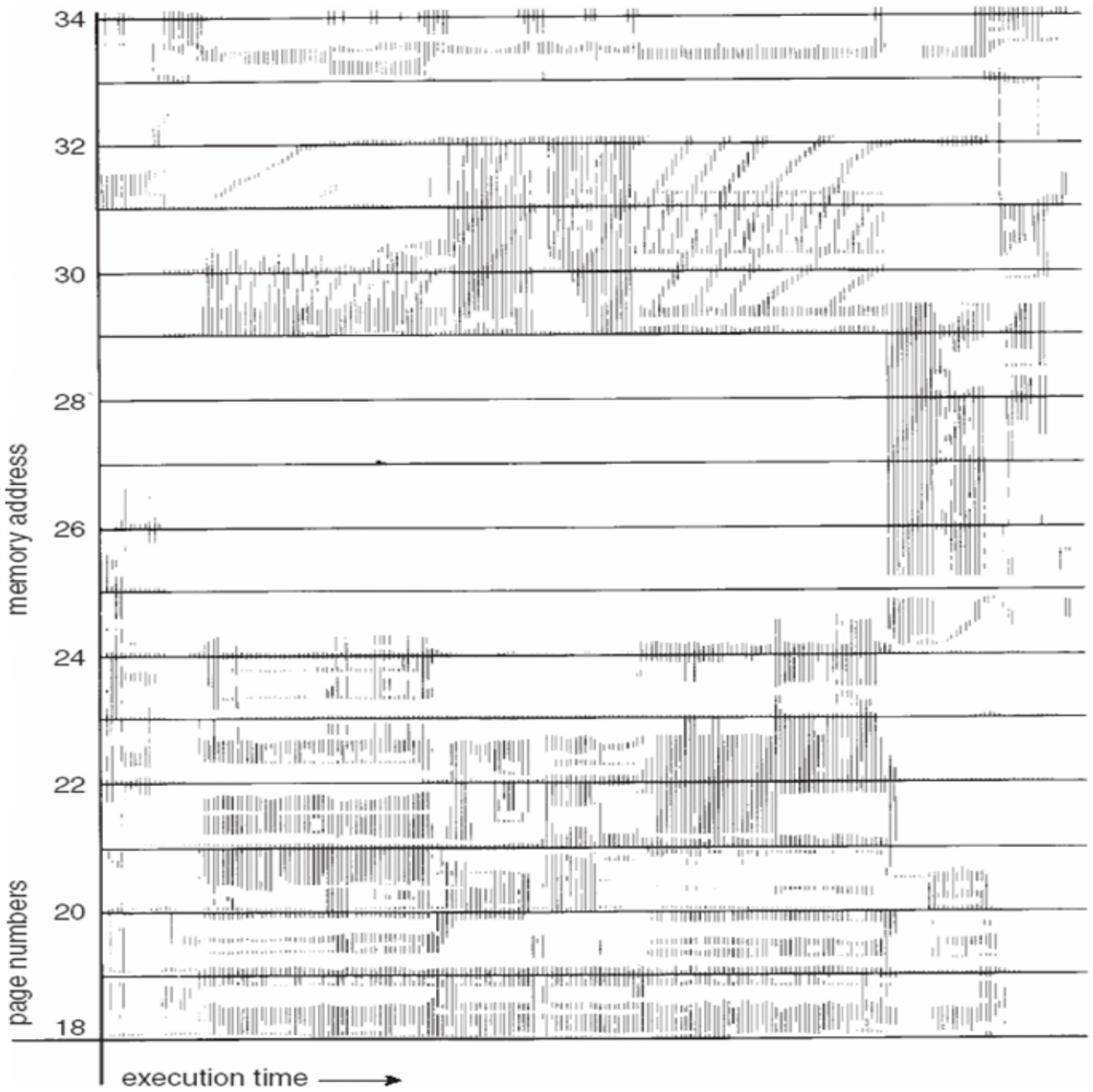
Prevent Thrashing

- How do we prevent thrashing?
- We must provide a process with as many frames as it needs.
- This brings us to the next question – how many frames does a process actually need?
- One solution is to use the working-set strategy.
- This technique starts by looking at how many frames a process is actually using. This defines the **locality** of process execution.
- As a process executes, it moves from locality to locality.
  - A locality is a set of pages that are actively used together.
  - A process may have several localities.
- For example, a function call defines a new locality.
- Memory references are made to instructions of the function call, its local variables, and a subset of the global variables.
- On exiting the function, the process moves to another locality.
  - Function call instructions and local variables are no longer in active use.
- Localities are defined by the program structure and its data structures.

Locality

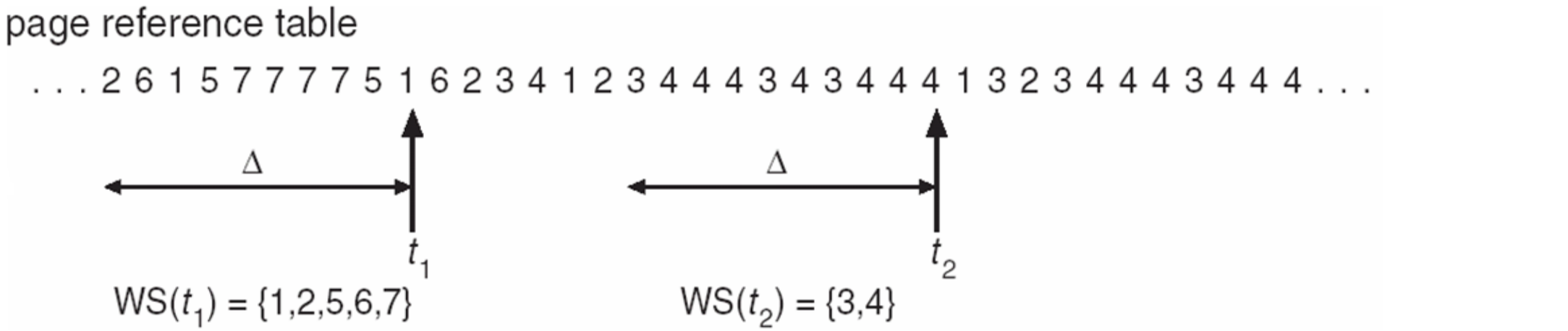
- Idea: **"to allocate enough frames for a process such that it can accommodate its current locality"**.
  - Consists of pages in active use.
- Process will fault only after it leaves the current locality.
- If we do not allocate enough frames to accommodate the current locality?
- The process will thrash.
- How does the working set model work?
- Uses a parameter Δ to define the working set window. The idea is to examine the most recent Δ page references. The set of pages in the most recent Δ page references is the working set.
- The idea is that if a page is in active use, it will be in the working set.
- Hence, it should not be evicted.

Locality In A Memory-Reference Pattern



Locality changes over time.

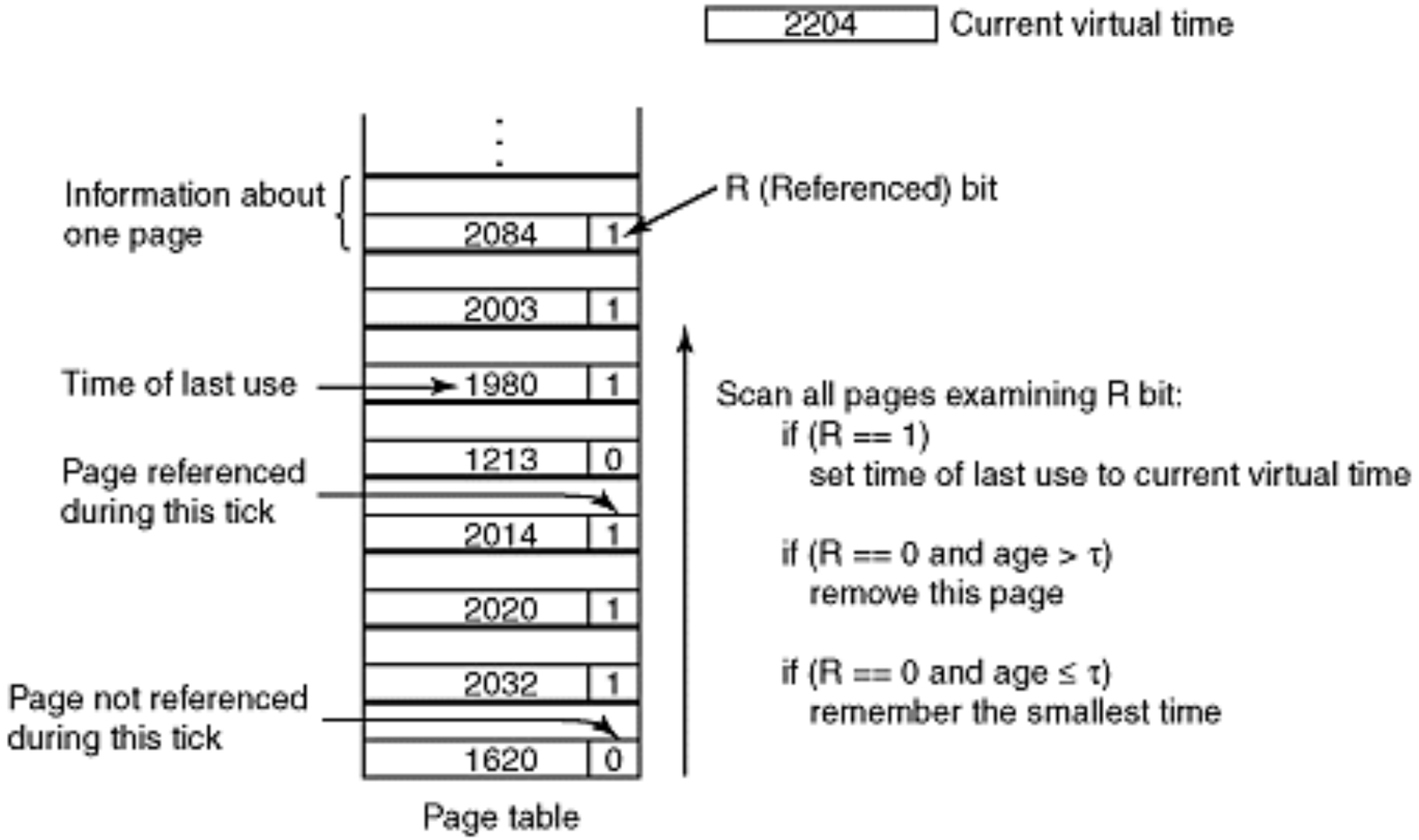
Working-set model



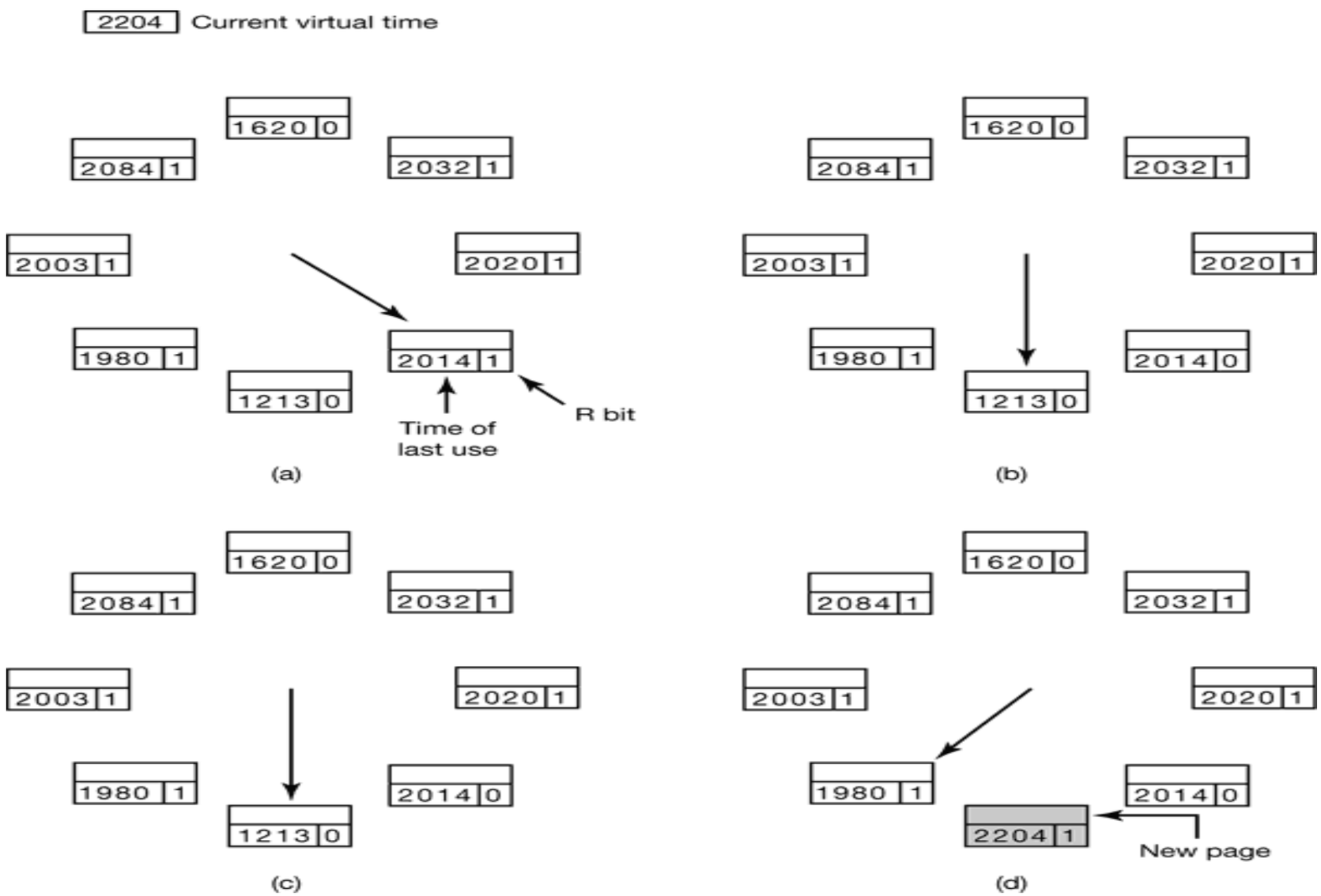
Working-Set Model

- Accuracy of the working set depend on the value of Δ.
- What if Δ is too small?
  - It will not cover the entire locality.
- What is Δ is too large?
  - It will overlap several localities.
- Beyond a value, Δ will cover all the pages touched during process execution.
- We can compute a working set size, WSS_i for each process in the systems, such that $D = \Sigma\ WSS_i$, where D is the total demand for frames.
- Process i needs WSS_i frames, and so on.
- If total demand is greater than the number of frames available ($D > m$), thrashing will occur.
  - Because some processes will not have enough frames to cover their working set.

Working set page replacement algorithm



WSClock Page Replacement



WSClock Page Replacement

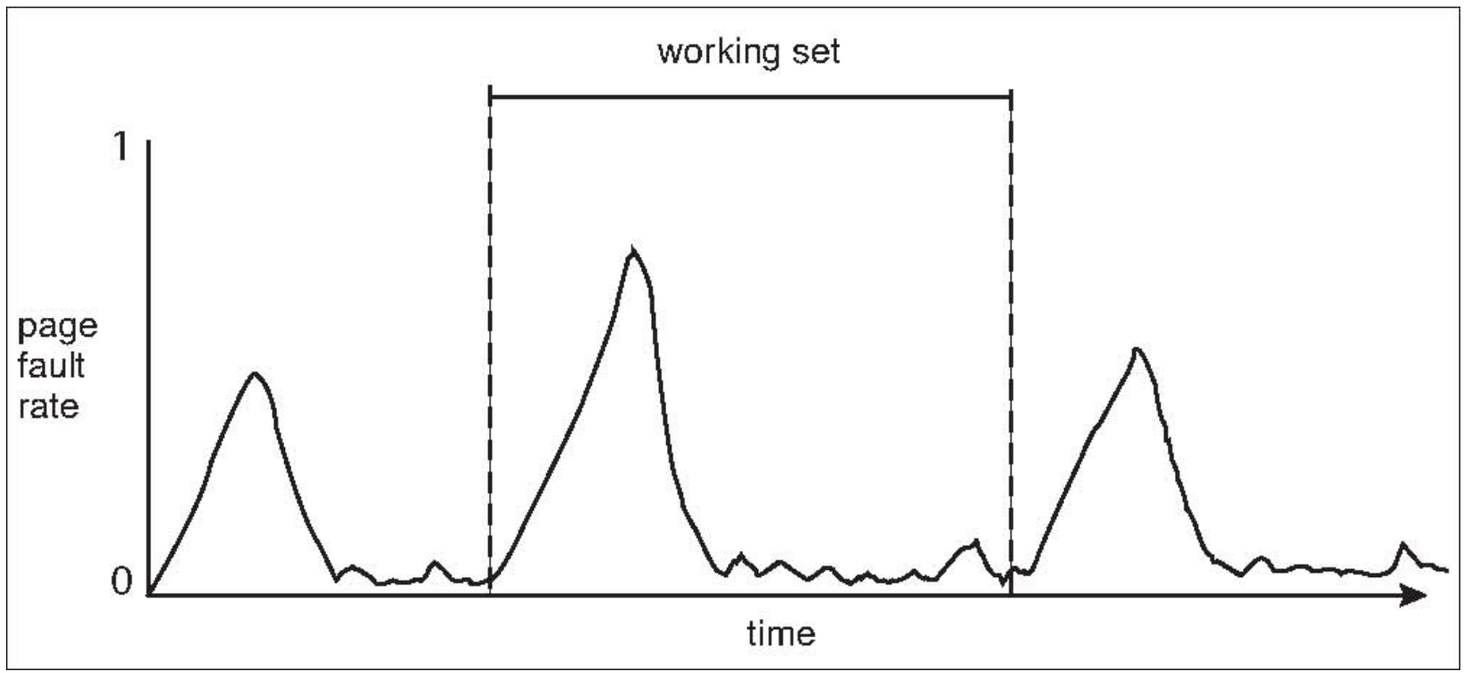
- Similar to working set algorithm.
- For each process P, keep record of active pages in circular list with three pieces of data: `time-of-last-use`, `R-bit`, `M-bit`.
- List is initially empty. When is the list populated?
- At each page fault, page pointed to by clock hand is examined first.
- If R-bit = 1, page accessed during current clock tick. So, do not evict, set R-bit = 0, and move to next page.
- If R-bit = 0, page is a candidate for eviction.
  - If page age > tau and page is clean => evict page.
  - If page age > tau and page is dirty, schedule disk write, and keep examining (hoping for an old and clean page).
- What if we reach beginning of clock round?
  - At least one disk write scheduled, evict that page.
  - No disk writes scheduled (all pages in working set), evict current/oldest/any clean page.
- May not need to search entire list (so is an LRU approximation).

WSClock Algorithm

- To summarize:
- Evict old and clean page (first choice).
- Next choice is old and dirty page.
- No page is “old” enough (age > tau), evict oldest (whether clean or dirty).

Working Sets and Page Fault Rates

- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time



Allocation of Frames

- Discussed algorithms that choose which page to replace when a page fault occurs.
- V. important related issue – how much memory should we allocate among competing runnable processes?
- See figure in next slide.
- Suppose process A gets a page fault.
- Which page to evict?
  - A5 (local replacement).
  - B3 (global replacement).
- Local -> process gets fixed fraction of memory.
- Global -> frames allocated dynamically to runnable processes.
- Which option sounds better?
- Global is more flexible.

Local vs. Global page replacement

	Age
A0	10
A1	7
A2	5
A3	4
A4	6
A5	3
B0	9
B1	4
B2	6
B3	2
B4	5
B5	6
B6	12
C1	3
C2	5
C3	6

(a)

A0
A1
A2
A3
A4
A6
B0
B1
B2
B3
B4
B5
B6
C1
C2
C3

(b)

A0
A1
A2
A3
A4
A5
B0
B1
B2
A6
B4
B5
B6
C1
C2
C3

(c)

Global and Local Allocation

- Some page replacement algorithms can work with either a global or a local allocation. Examples?
  - FIFO: need to replace the oldest page, so we can replace the oldest page in all of memory (global) or the oldest page owned by the current process (local).
  - LRU: need to replace the page that has not been used the longest, so that page could be among all pages in memory (global) or among the pages owned by the current process (local).
- Some algorithms are better suited for a specific allocation scheme.
  - The working set algorithm refers to the working set of a specific process (local).
  - Uses the concept of locality.
  - Would not work well using global scheme.
    - No working set for the system as a whole.
    - Would lose the locality process.

Allocation Algorithms

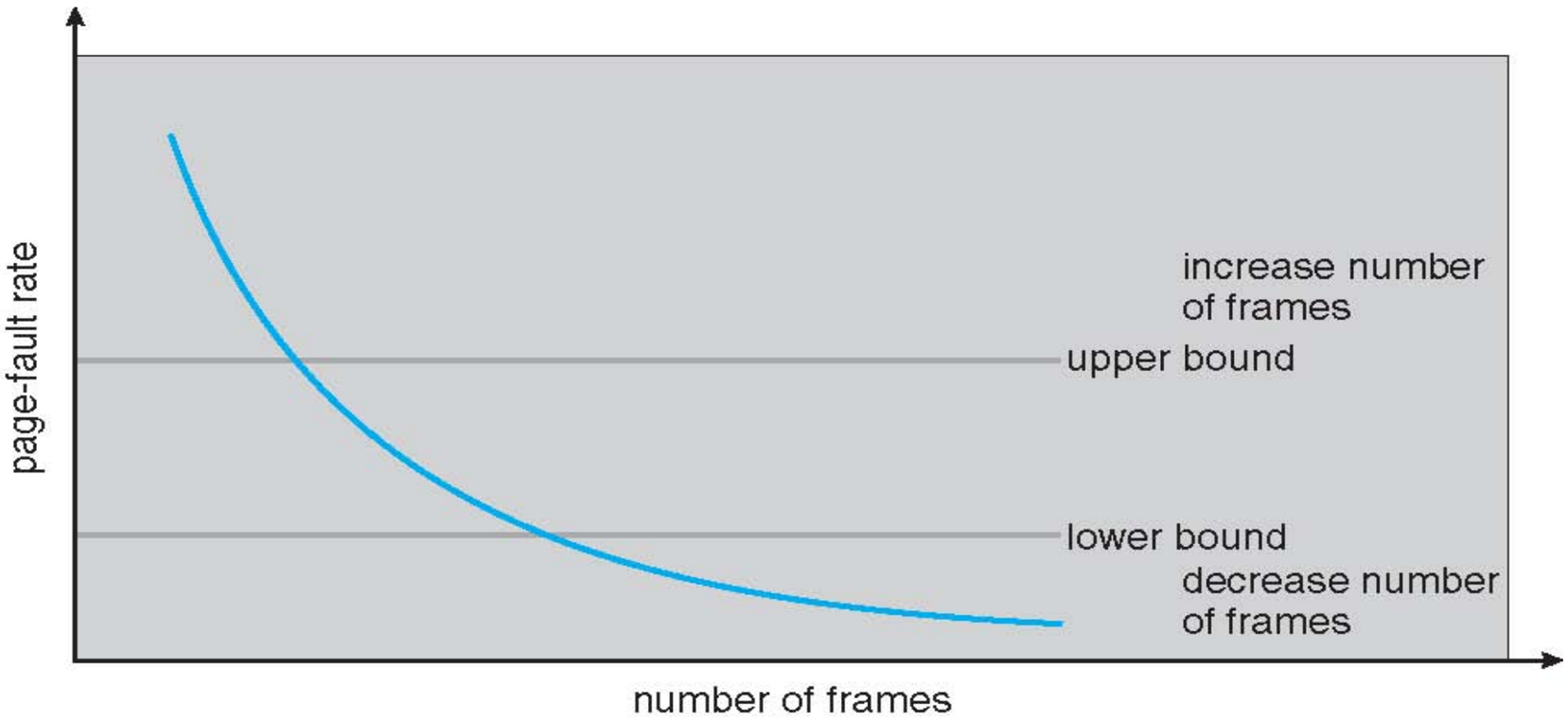
- Problem statement is simple: split *m* frames among *n* processes.
- A basic, naïve algorithm is the equal allocation algorithm.
  - Give every process equal share.
  - How many frames would each process get?
  - *m*/*n*.
- If m = 93 and n = 5, each process gets 18 frames. The 3 leftover frames become a part of the free-frame pool.
- This scheme is simple to implement.
- What is the drawback?
  - All processes are not identical, they have different memory requirements.
  - Compare a programing editor to a web browser.
  - What if both got equal number of frames?

Allocation Algorithms

- Can we come up with a better allocation algorithm?
- A better alternative is to use proportional allocation, i.e. allocate based on the process size.
- Let the size of virtual memory for process p_i be s_i. Then, the sum of the memories of all processes is, $S = \Sigma\ s_i$.
- Then, is the total number of available frames is m, we allocate a_i frames to process p_i, where a_i is approximately = (s_i/S) x m.
- Using this scheme, we would split 62 frames among two process in the following manner:
  - Lightweight process gets 10/137 x 62 = 4.
  - Heavyweight process gets 127/137 x 62 = 57.

Page-Fault Frequency Algorithm

- Global allocation: need to > or < frame allocation to process dynamically. How?
- Measure the page fault frequency/rate (# page faults / unit time).
- Establish “acceptable” **page-fault frequency** rate and use local replacement policy
  - If actual rate too low, process loses frame(s).
  - If actual rate too high, process gains frame(s).



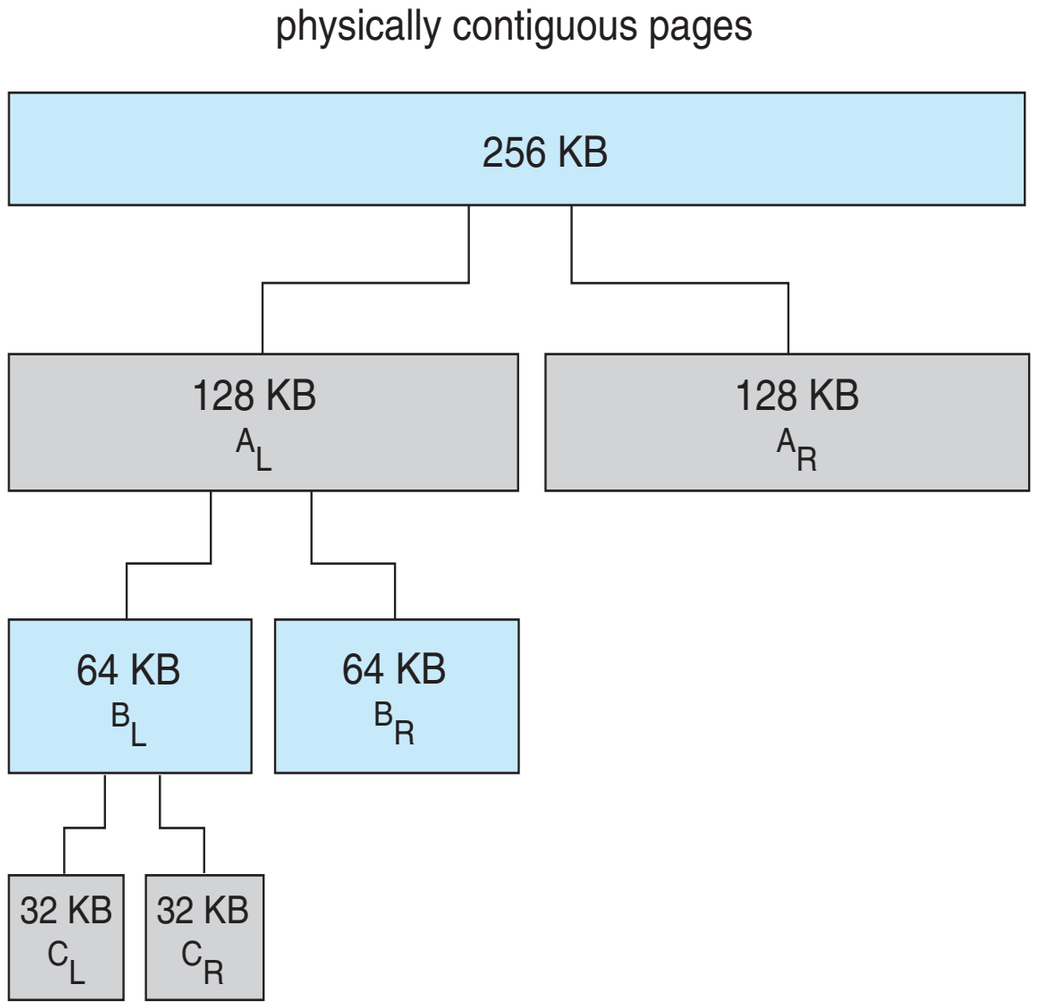
Allocating Kernel Memory

- Kernel memory is treated differently from user memory. Why?
- Often allocated from a free-memory pool different than pool for user processes.
  - **Kernel requests memory for structures of varying sizes.**
    - Some of those structures may be very small (much smaller than a page).
  - **Some kernel memory needs to be contiguous**
    - For performance, kernel interacts directly with physical memory (no virtual memory, like for user processes).
    - Kernel pages may require contiguous physical frames.

Buddy System

- Allocates memory from fixed-size segment consisting of physically-contiguous pages.
- Memory allocated using **power-of-2 allocator**
  - Satisfies requests in units sized as power of 2.
  - Request rounded up to next highest power of 2.
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2.
    - Continue until appropriate sized chunk available.
- For example, assume 256KB chunk available, kernel requests 21KB
  - Split into A_L and A_R of 128KB each
    - One further divided into B_L and B_R of 64KB
      - One further into C_L and C_R of 32KB each – one used to satisfy request
- Advantage: quickly **coalesce** unused chunks into larger chunk
- Disadvantage?

Buddy System Allocator



Other Considerations

- Major decisions for paging system:
  - Page replacement algorithm and frame allocation policy.
- Lets briefly discuss some other considerations:
- Pre-paging .
- Page size (small vs. large page size).
- TLB reach.

Prepaging

- Pure demand paging: no pages in memory at the start.
- What if we could reduce the large number of page faults that occurs at process startup?
- *Pre-page all or some of the pages a process will need, before they are referenced*.
  - E.g. pages in sequential file access.
- Key question: which pages to pre-page?
- If pre-paged pages are unused, I/O and memory was wasted.
- Assume *s* pages are pre-paged and fraction *α* of the pages is used.
  - Is cost of ***s * α*** saved pages faults > or < than the cost of pre-paging ***s * (1- α)*** unnecessary pages*?*
  - ***α*** near zero (0) ⇒ ?.
  - ***α*** *near one (1)* ⇒ ?

Page Size

- What should the page size be: small page size (4K) or large page size (4MB)?
- Page size selection must take into consideration:
  - Fragmentation: small vs. large?
  - Page table size : small vs. large?
  - I/O overhead: small vs. large?
  - Resolution and locality: small vs. large?
  - Number of page faults: small vs. large?
  - TLB size and effectiveness: small vs. large?
- On average, growing over time.

Small vs. large page sizes

- Disk I/O overhead:
  - = data transfer time (very less) + seek time (very large, e.g. 8 msec).
  - At 50 Mbps, 512 bytes transfer time = ?
  - 0.01 msec., so total I/O overhead = 8.01.
  - For 1K page, it is = 8.02.
  - Time to read 2 separate pages of 512 bytes each = ?
  - 16.02.
  - Hence, large page size is better for disk I/O overhead.
- Locality and resolution:
  - Smaller page size may map locality at a finer grain.
  - Close to optimum memory utilization.
  - 200 K vs 1 byte page. Which has better utilization?

Small vs. large page sizes

- Number of page faults.
  - Previous example: small page, better memory utilization.
  - Drawback?
  - Much larger number of page faults.
  - 200K page size, process uses 100K.
  - Page faults for 200K page size?
  - Page faults for 1 byte page size?
  - Hence, larger page size means smaller number of page faults.

TLB Reach

- TLB Reach: the amount of memory accessible from the TLB
- TLB Reach = (# of TLB entries) X (Page Size).
- Ideally, the working set of each process may be stored in the TLB.
- How to increase TLB reach?
- More TLB entries (greater TLB size). Expensive.
- Increase the Page Size.
  - This may lead to an increase in fragmentation as not all applications require a large page size.
- Provide Multiple Page Sizes
  - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation.
  - For example, the kernel.

Virtual Memory Management in Linux

- Uses demand paging with global page-replacement policy with LRU clock approximation.
- Uses two lists: `active-list` and `inactive-list`.
- What would these two lists contain?
- How are these lists used?
- Page accessed, set its R-bit, add to end of active-list.
- Periodically, R-bits of all pages in active-list set to zero.
- Where will least recently used page be in active-list? At the front.
- This is migrated to end of inactive-list.
- Page in inactive-list is accessed, move it to end of active-list.
- Try to keep these lists in balance.
- If active-list >>>> inactive-list, migrate from front of active to back of inactive (to reclaim the “inactive” pages).

Linux VM Management

- Kwapd (kernel swap daemon) process runs periodically.
- If free memory < system threshold, kswapd scans pages from inactive list to reclaim.
- Daemon: background process not in user control. Name ends with a ‘d’.

Virtual Memory in Windows 10

- Windows uses demand pages with clustering.
- Brings in a cluster of pages on a page fault (recognizing locality).
  - Page that faulted + pages preceding and succeeding the faulted page.
- Two parameters used: working set maximum (wsmax, value 345) and working set minimum (wsmin, value 50).
- Guaranteed to have wsmin, may go up to wsmax.
- These have soft limits and one can go below wsmin and beyond wsmax.
- Windows performs working set trimming in memory pressure situations.
- Bring down to wsmin if extra pages allocated (may even go below this).
- Large idle processes targeted before small active ones.

End of Chapter 9