Chapter 10: Virtual Memory

Chapter 10: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples

Objectives

- To describe the benefits of a virtual memory system
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames
- To discuss the principle of the working-set model
- To examine the relationship between shared memory and memory-mapped files
- To explore how kernel memory is managed

Virtual Memory

Various memory-management strategies used in computer systems have the same goal: to keep many processes in memory simultaneously to allow multiprogramming. It requires that an entire process be in memory before it can execute.

.

- **Virtual memory** involves the separation of logical memory as perceived by users from physical memory. This separation **allows** an extremely large virtual memory for programmers when only a smaller physical memory is available

- **Virtual memory is a technique that allows the execution of processes that are not completely in memory**

Advantages of Virtual Memory

Major advantage of is that **programs can be larger than physical memory.**

1. Virtual memory abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory

2. This technique frees programmers from the concerns of **memory-storage limitations**

3. Virtual memory also allows processes to **share files easily** and to **implement shared memory**

4. In addition, it provides an **efficient** mechanism for **process creation**.

1. Entire Program Rarely Used

- An executable program might be loaded from disk into memory. Basic requirement is to load the entire program in physical memory at program execution time.
  - Code needs to be in memory to execute, but entire program rarely used
  - Error code to handle unusual error conditions, which seldom occur
  - Arrays, lists, and tables are often allocated more memory than they actually need.
    - An array may be declared 100 by 100 elements, even though it is seldom larger than 10 by 10 elements.

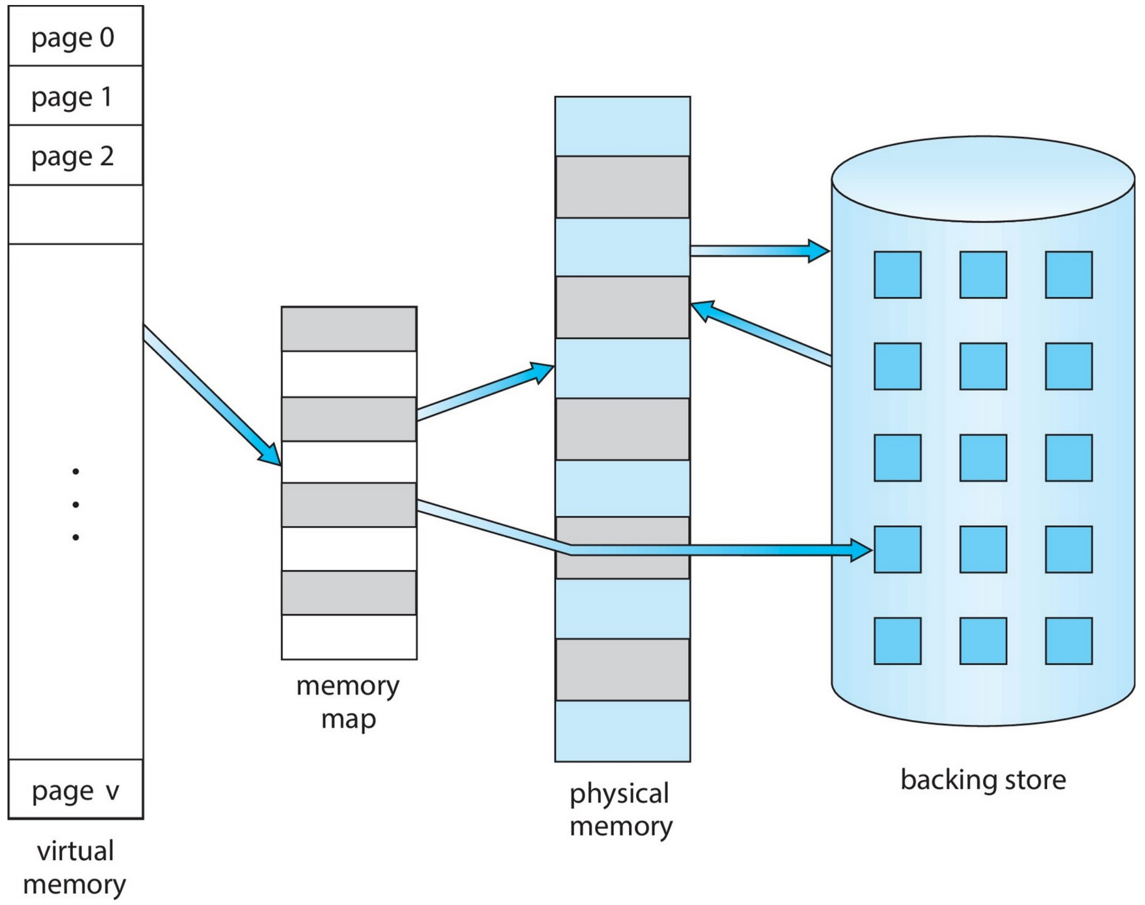
2. Ability to execute partially-loaded program

- Even in cases where entire program is needed, it may not be all be needed at the same time
- The ability to execute a program that is only partially in memory would confer many benefits:
    - Each program takes less memory while running -> more programs running concurrently
    - Increased CPU utilization and
    - throughput with
    - no increase in response time or turnaround time
    - Less I/O needed to load or swap programs into memory -> each user program runs faster

3. Virtual memory that is larger than the physical memory

Separation of logical memory as perceived by users from physical memory allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available.

- Program no longer constrained by limits of physical memory
- Logical address space can be much larger than physical address space



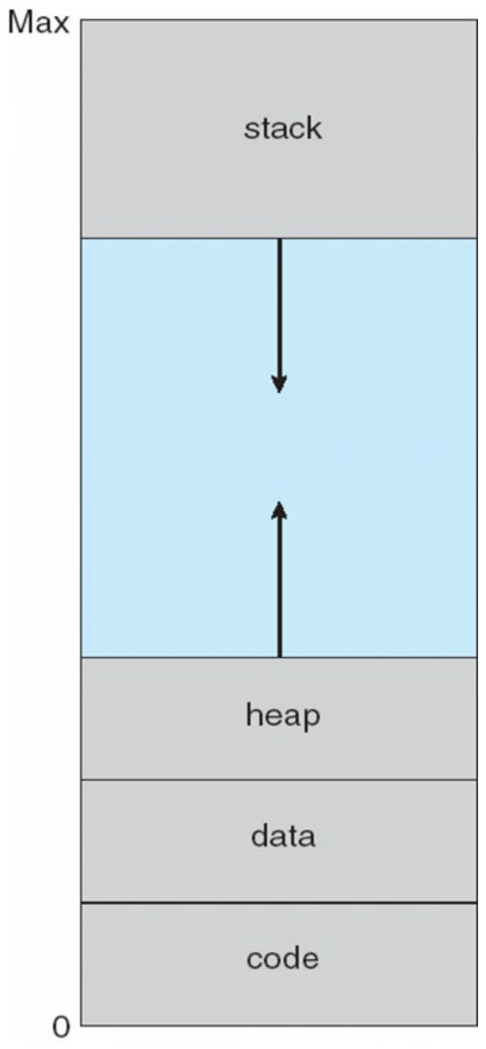
Virtual address space of a process in memory

Virtual address space of a process refers to logical view of how a process is stored in memory.

- Typically, this view is that a process begins at a certain logical address say, address 0, and exists in contiguous memory
- **Stack** grows downward in memory through successive function calls.
- **Heap** grows upward in memory as it is used for dynamic memory allocation.

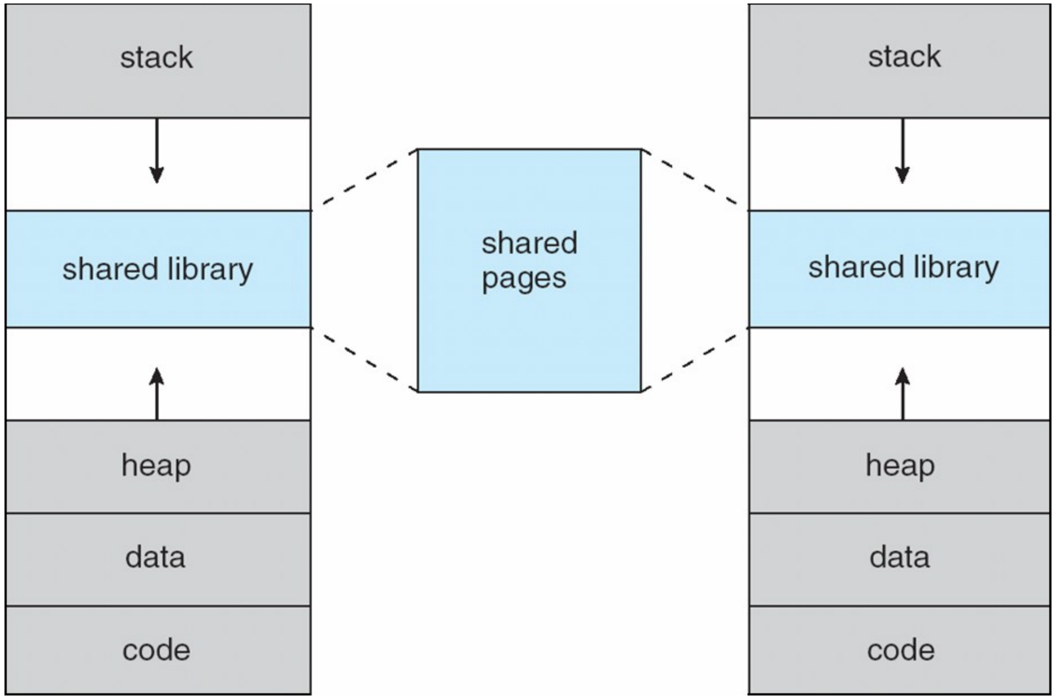
The blank space (or hole) between the heap and the stack is part of the virtual address space but will require actual physical pages only if the heap or stack grows.

- Virtual address spaces that include holes are known as sparse address spaces.
- Using a **sparse address space** is beneficial because the holes can be filled as the stack or heap segments grow or if we wish to **dynamically link libraries** (or possibly other **shared objects**) during program execution



Shared Library Using Virtual Memory

1. System libraries such as C library can be shared by several processes through mapping of the shared object into a virtual address space. A library is mapped read-only into the space of each process that is linked with it.
2. Similarly, processes can share memory. Two or more processes can communicate through the use of shared memory.
3. Pages can be shared during process creation with **`fork()`** `system call`, thus speeding up process creation



Virtual Memory Implementation

Virtual memory can be implemented **via Demand paging**

Consider how an executable program might be loaded from disk into memory.

- Without Virtual memory:
  - Load the entire program in physical memory at program execution time.
    - Example: Loads the executable code for ***all*** options of a list of available options, regardless of whether or not an option is ultimately selected by the user

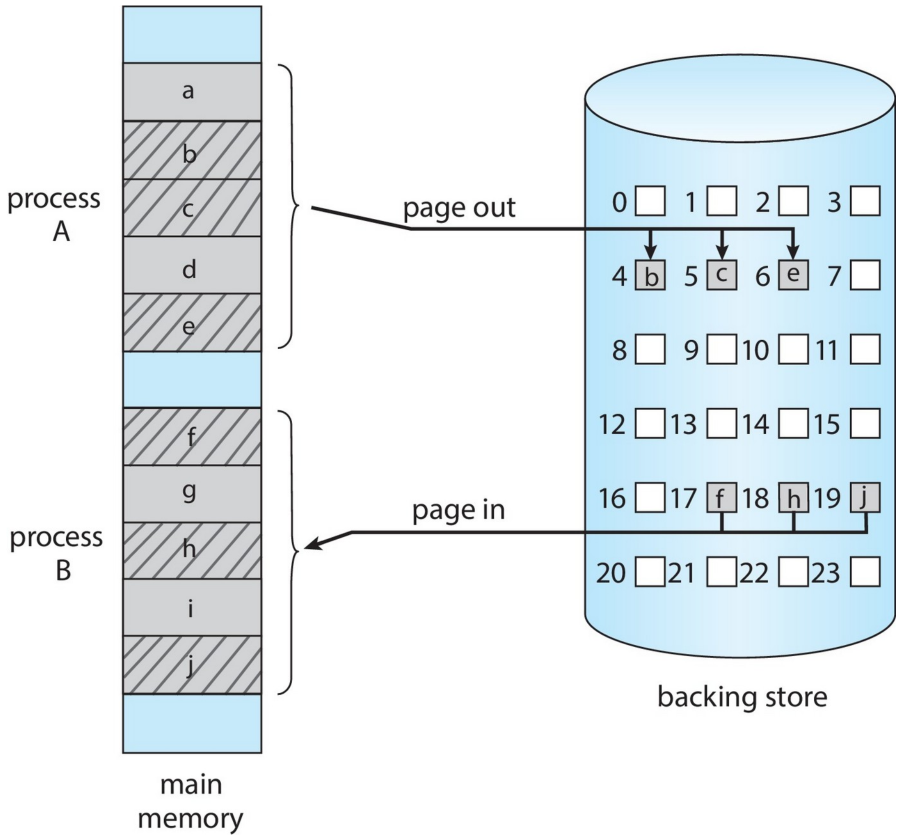
- Virtual Memory Strategy:
  - Processes reside in secondary memory (usually a disk)
  - Load pages to main memory, only as they are needed.

This technique is known as **demand-paging** and is commonly used in virtual memory systems.

- With **demand-paged virtual memory**, **pages are loaded only when they are demanded during program execution**.

Demand-Paging

- Programs reside in secondary memory (usually hard disk drive (HDD) or Non-Volatile Memory (NVM) device.
- Load the portions of the programs that are needed.
- Pages that are never accessed are thus never loaded into physical memory
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Swapping of entire processes is no more used.



Demand-paging is used in most systems, including Linux and Windows

Basic Concepts

- With swapping, pager guesses which pages will be used before swapping out again
- Instead, pager brings in only those pages into memory
- How to determine that set of pages?
  - Need new MMU functionality to implement demand paging
- If pages needed are already **memory resident**
  - No difference from non demand paging
- If page needed and not memory resident
  - Need to detect and load the page into memory from storage
    - Without changing program behavior
    - Without programmer needing to change code

Valid-invalid Bits

Hardware support to distinguish between the pages that are in memory and the pages that are on the disk.

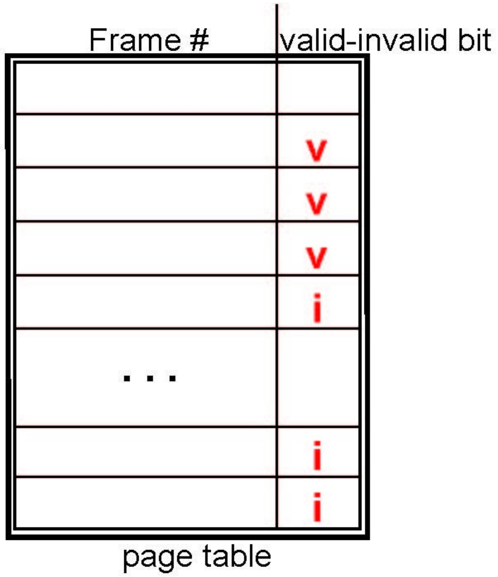
- The valid–invalid bit is set to “valid,” the page is both legal and in memory.
- If the bit is set to “invalid,” the page either is
  - not valid (that is, not in the logical address space of the process) or
  - valid but is currently on the disk.
- The page-table entry for a page that is brought into memory is set as usual
- The page-table entry for a page that is not currently in memory is simply marked invalid

Page is needed ⇒ reference to it

- invalid reference ⇒ abort
- not-in-memory ⇒ bring to memory

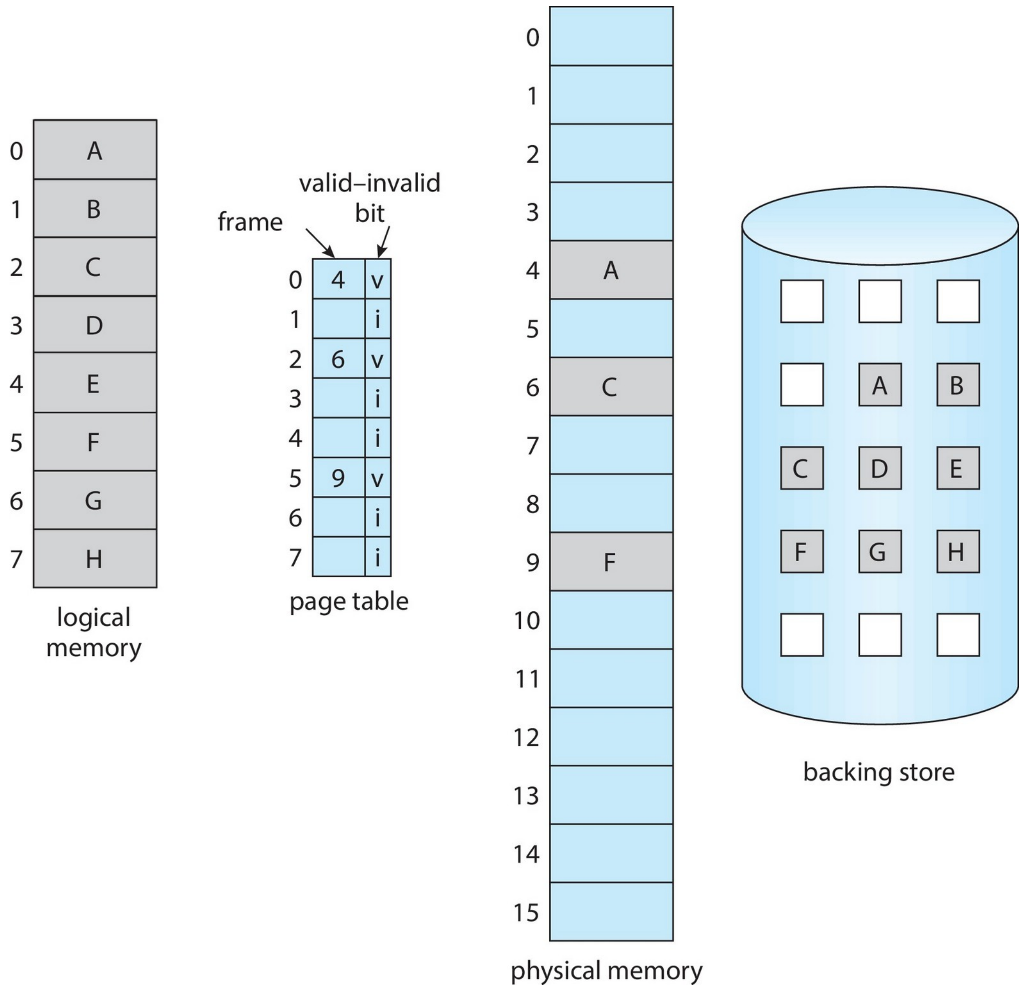
Valid-Invalid Bit

- 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:



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

Page Table When Some Pages Are Not in Main Memory



Page Fault

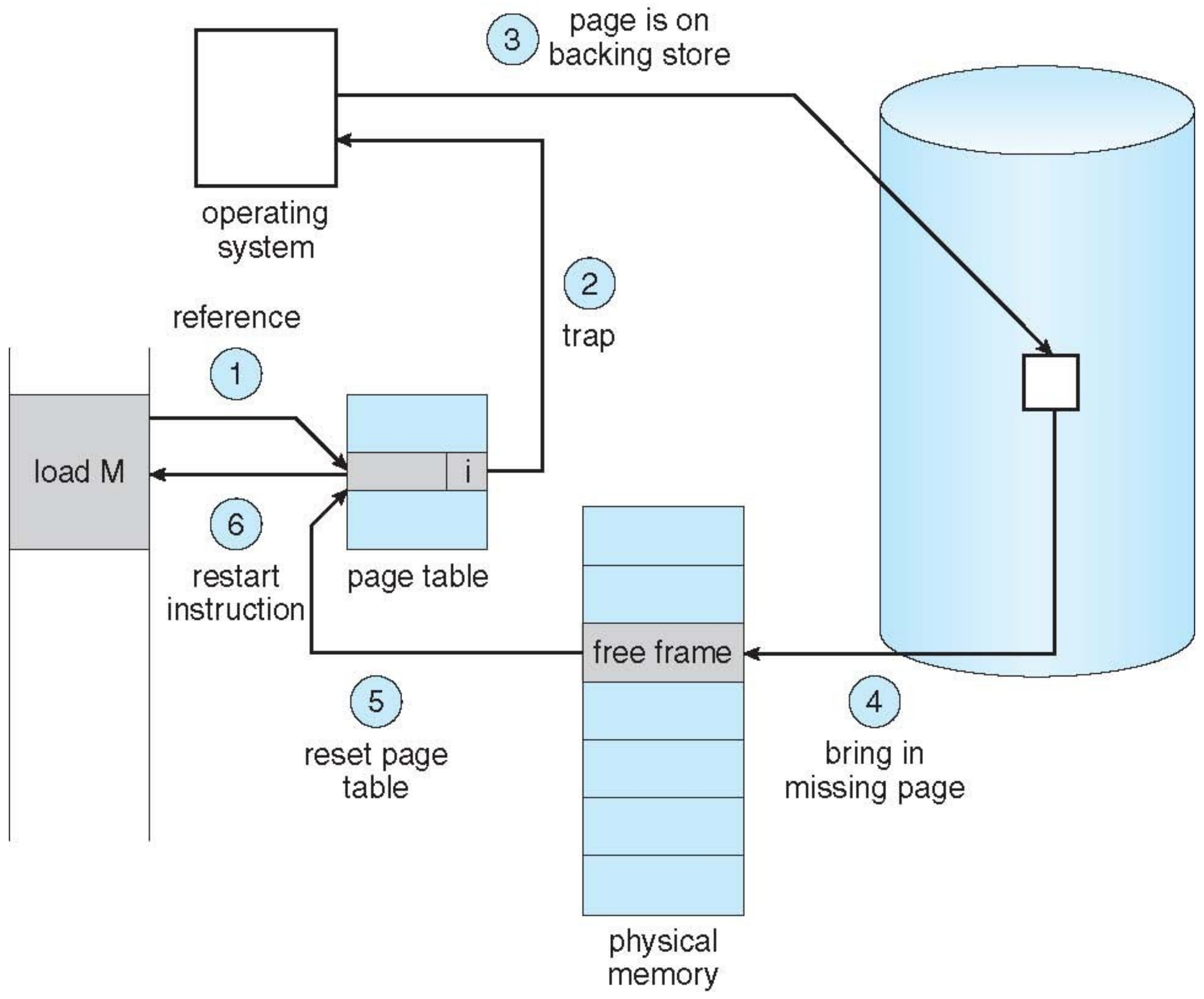
Page fault

- If there is a reference to a page, first reference to that page will **trap** to operating system
- Access to a page marked invalid causes a page fault, and traps to OS

- Operating system check an internal table to decide:
  - Invalid reference ⇒ abort
  - Reference was a valid, but not in memory. Handle page fault.

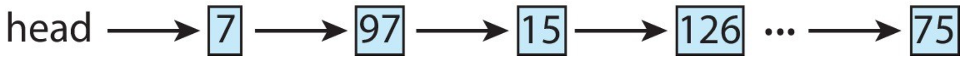
- Handling page fault:
  1. Find a free frame (by taking one from free-frame list)
  2. Read the page page into this frame via scheduled disk operation
  3. Reset tables to indicate page now in memory
     Set validation bit = **v**
  4. Restart the instruction that caused the page fault
  5. The process can now access the page as though it has been in memory

Steps in Handling a Page Fault



Free-Frame List

- When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory.
- Most operating systems maintain a **free-frame list** -- a pool of free frames for satisfying such requests.



- Operating system typically allocate free frames using a technique known as **zero-fill-on-demand** -- the content of the frames zeroed-out before being allocated.
- When a system starts up, all available memory is placed on the free-frame list.

Aspects of Demand Paging

- Extreme case – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And for every other process pages on first access
  - **Pure demand paging**
- Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory. If the page for result is not currently in memory, restart and repeated work will be needed.
  - **Locality of reference** results in reasonable performance
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart

Performance of Demand Paging

Demand paging can significantly affect the performance of a computer system.

- To see why, let’s compute the effective access time for a demand-paged memory.
- If we have no page faults, the effective access time is equal to the memory access time.
- If, however, a page fault occurs, we must first read the relevant page from secondary storage and then access the desired word.
- There are three major task components of the page-fault service time.

  1. Service the page-fault interrupt.
  2. Read in the page.
  3. Restart the process.

  The first and third tasks can be reduced, with careful coding, to several hundred instructions. These tasks may take from 1 to 100 microseconds each. The page switch time will probably be close to 8 milliseconds

Effective access time

- Assume the memory-access time denoted by *ma* nanoseconds.
- Let *p* be the probability of a page fault ($0 \le p \le 1$). We would expect p to be close to zero—that is, we would expect to have only a few page faults.
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
  - The effective access time is:

    EAT = (1 – *p*) x memory access
    + *p x* (page fault overhead
    + swap page out
    + swap page in)
- Effective Access Time (EAT)

  Effective access time = $(1 - p) \times ma + p \times$ page fault time

Demand Paging Example

- Memory access time ma = 200 nanoseconds = 200/1000 microseconds = 0.2
- Average page-fault service time = 8 milliseconds = 8 x 1000 microseconds
- EAT = (1 – p) x 0.2 + p x 8000

  = 0.2 - 0.2 p + 8000 x p

  = 0.2 + 7999.8 p microseconds

milli = 10^3
micro =10^6
nano = 10^{-9}

- If one access out of 1,000 causes a page fault, then p = 1/1000 = .001

  EAT = 0.2 + 7999.8 x .001 = 0.2 + 7.9998

  = 8.2 microseconds.

  This is a slowdown by a factor of 40!! i.e., 7.9998 / .2) = 40

- If we want performance degradation < 10 percent
  - 0.22 > 0.2 + 7999.8 x p

    0.02 > 7999.8 x p

    0.02 / 7999.8 > p

0.2 if no page fault.
With 10% additional time
0.2 x 1.1 = 0.22

- i.e., p < 1/399,990.
- To keep the slowdown due to paging at a reasonable level, we can allow only fewer that one memory access out of 399,990 to page-fault
  - Less than one page fault in every 400,000 memory accesses

Demand Paging Optimizations

- I/O to swap space is faster than to the file system because
  - Swap space is allocated in much larger blocks and less management needed than file system
- One option: copy entire file image into the swap space at process load time
  - Then page in and out of swap space
  - Disadvantage is the copying of the file image at start-up.
- Second option: Demand page from a file system initially but to write the pages to swap space as they are replaced.
  - Used in several operating systems, including Linux and Windows
- Swap space must still be needed for
  - Pages not associated with a file (like stack and heap) known as **anonymous memory**
- Mobile systems
  - Typically don’t support swapping
  - Instead, demand page from file system and reclaim read-only pages (such as code)

Copy-on-Write

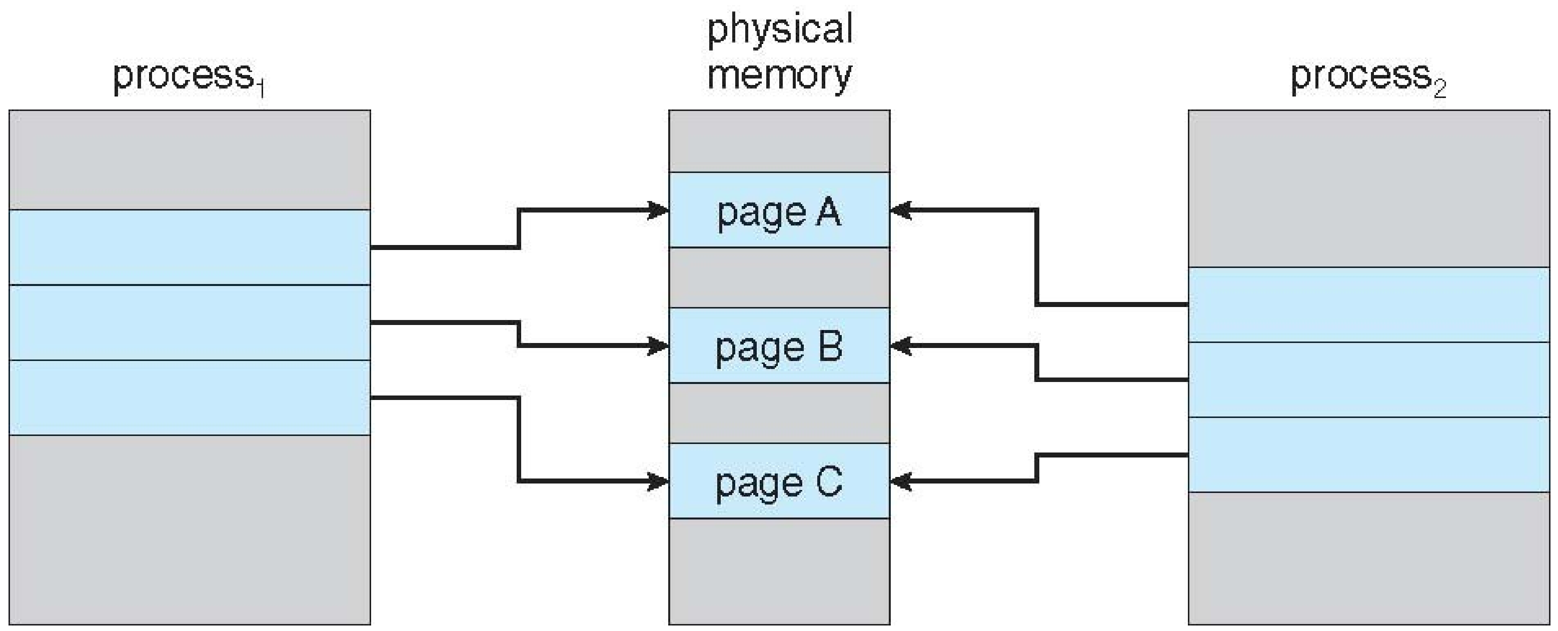
Process creation using fork() may **initially bypass the need for demand paging.**

- A new process is created by the fork() system call. The child process consists of a copy of the parent's address space for the parent
- Many child processes invoke the **exec()** system call immediately after creation, making copying of the parent's address space unnecessary.
- Instead, use a technique known as **copy-on-write,** which works by allowing the parent and child processes initially to share the same pages.
- Pages that cannot be modified (pages containing executable code) can be shared by the parent and child.
- Pages that can be modified are marked as copy-on-write pages, meaning that if either process writes to a shared page, a copy of the shared page is created.

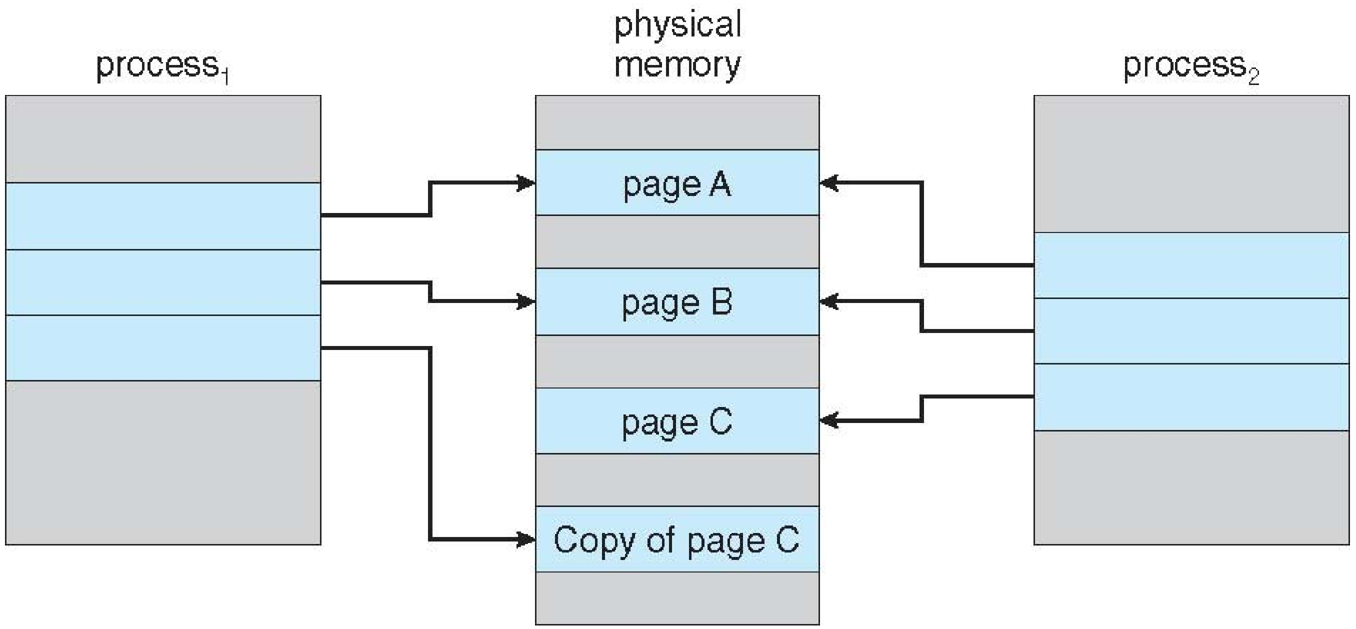
The technique similar to page sharing provides rapid process creation and minimizes the number of new pages that must be allocated to the newly created process.

- Copy-on-write is a common technique used by several operating systems, including Windows, Linux, and macOS.

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



`vfork()` virtual memory fork

- Several versions of UNIX (including Linux, macOS, and BSD UNIX) provide a variation of the fork() system call
  - **Designed to have child call `exec()`**
  - Very efficient
- With vfork(), the parent process is suspended, and the child process uses the address space of the parent.
- Because vfork() does not use copy-on-write, if the child process changes any pages of the parent's address space, the altered pages will be visible to the parent once it resumes.
- Must be used with caution to ensure that the child process does not modify the address space of the parent.
- Intended to be used when the child process calls exec() immediately after creation.
- Because no copying of pages takes place, vfork() is an extremely efficient method of process creation and is sometimes used to implement UNIX command-line shell interfaces.

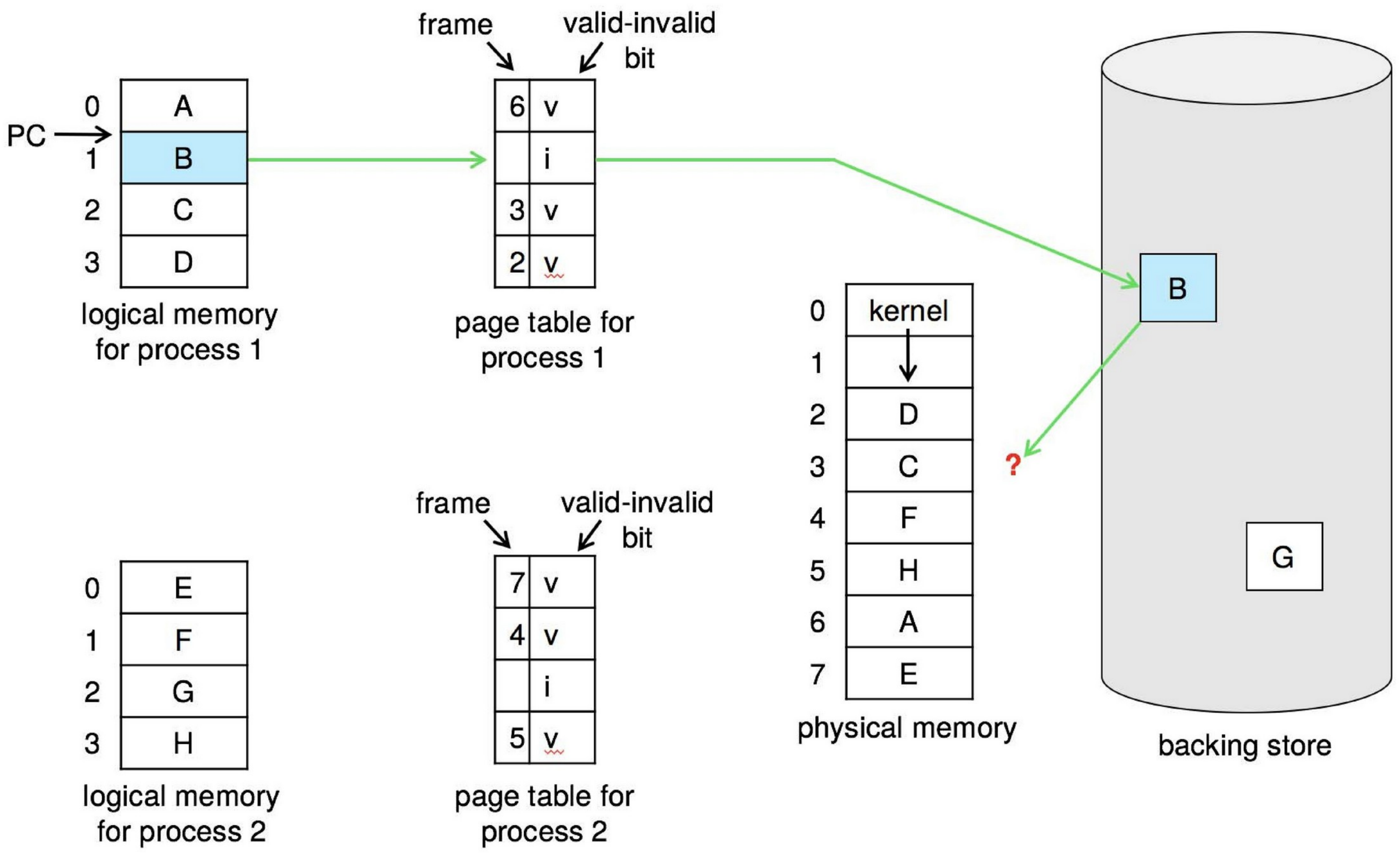
What Happens if There is no Free Frame?

- Used up by process pages
- Also, in demand from the kernel, I/O buffers, etc.
- How much to allocate to each?
- **Page replacement** – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

Page Replacement

- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement
- Use **modify** (**dirty**) **bit** to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

Need For Page Replacement

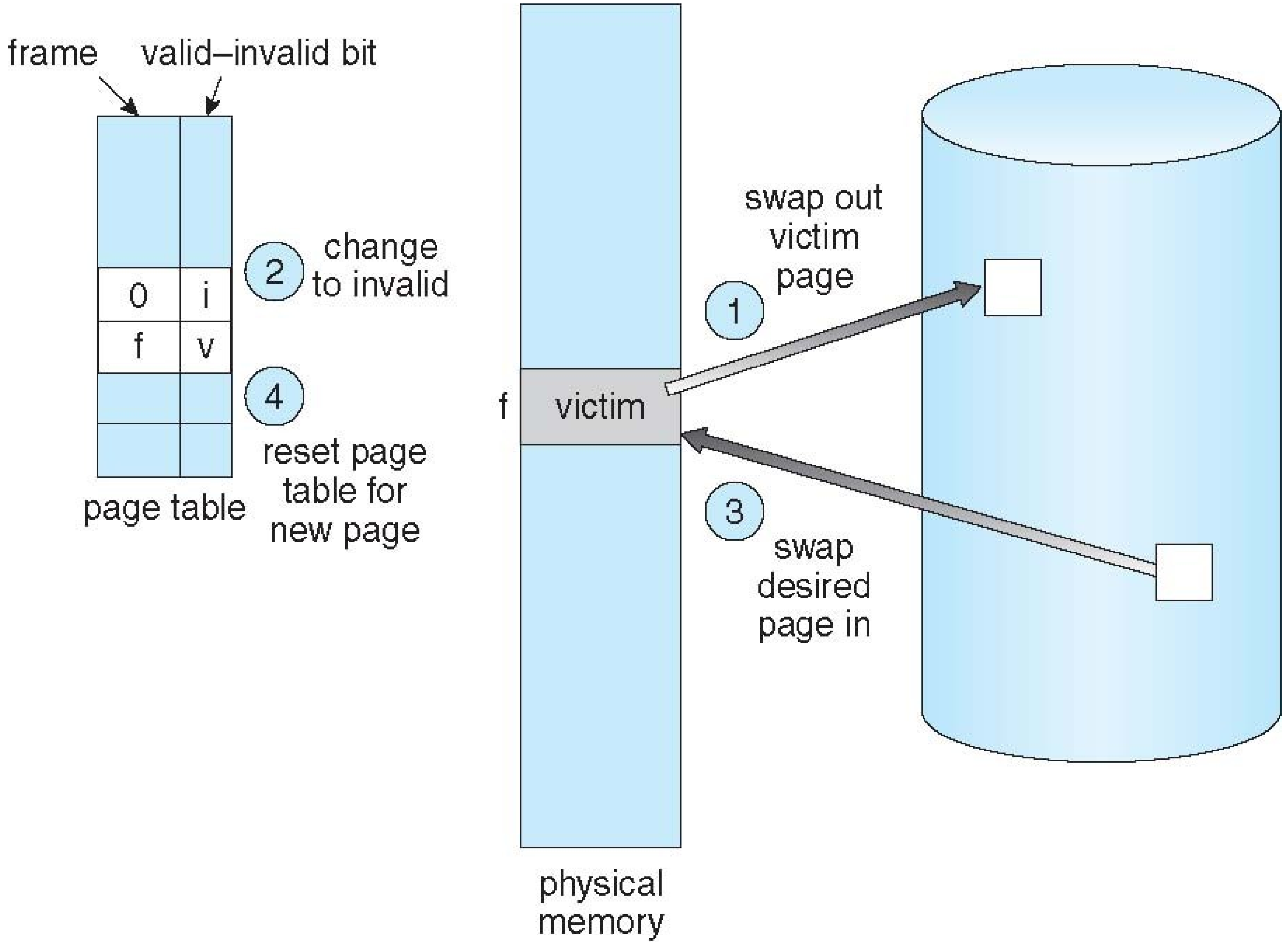


Basic Page Replacement

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**
     - Write victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap

- Note that, if no frames are free, ***two*** page transfers (one out and one in) are required.
- This effectively doubles the page-fault service time and increases the EAT (effective access time)

Page Replacement

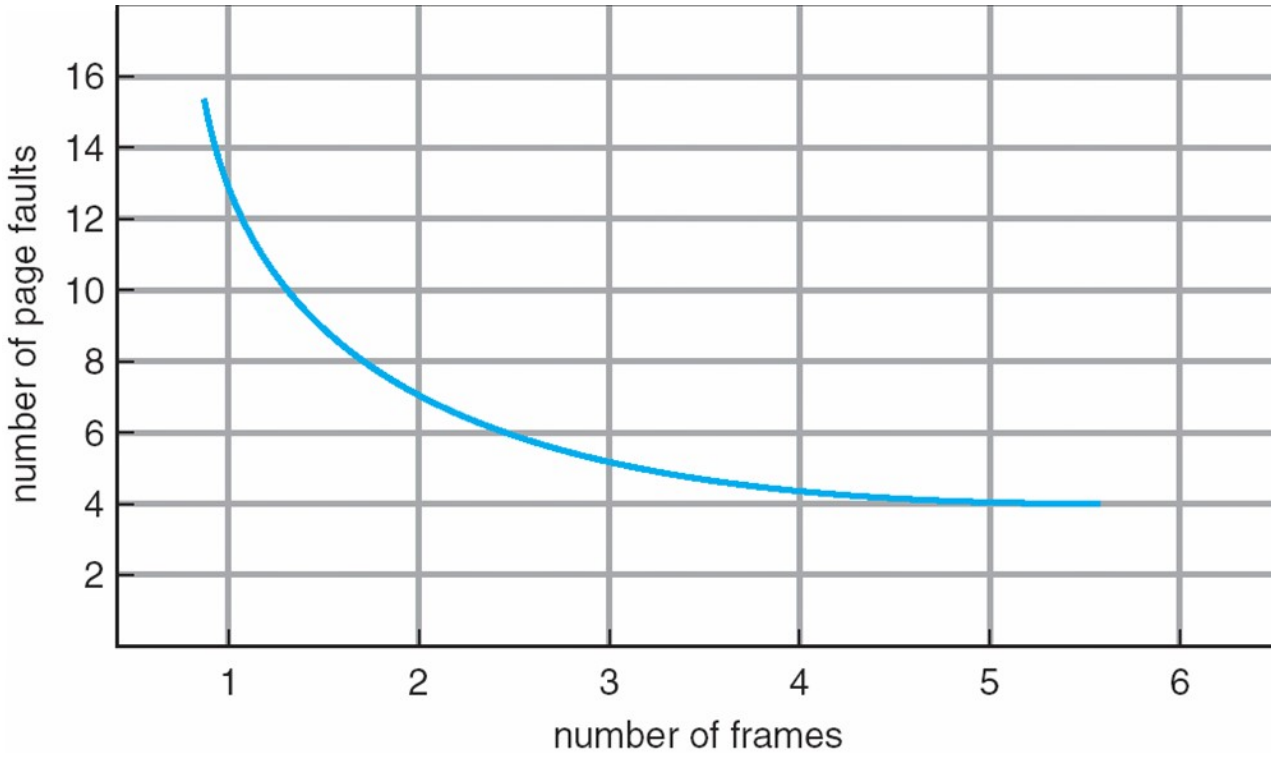


Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available
- In all our examples, the **reference string** of referenced page numbers is

  7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

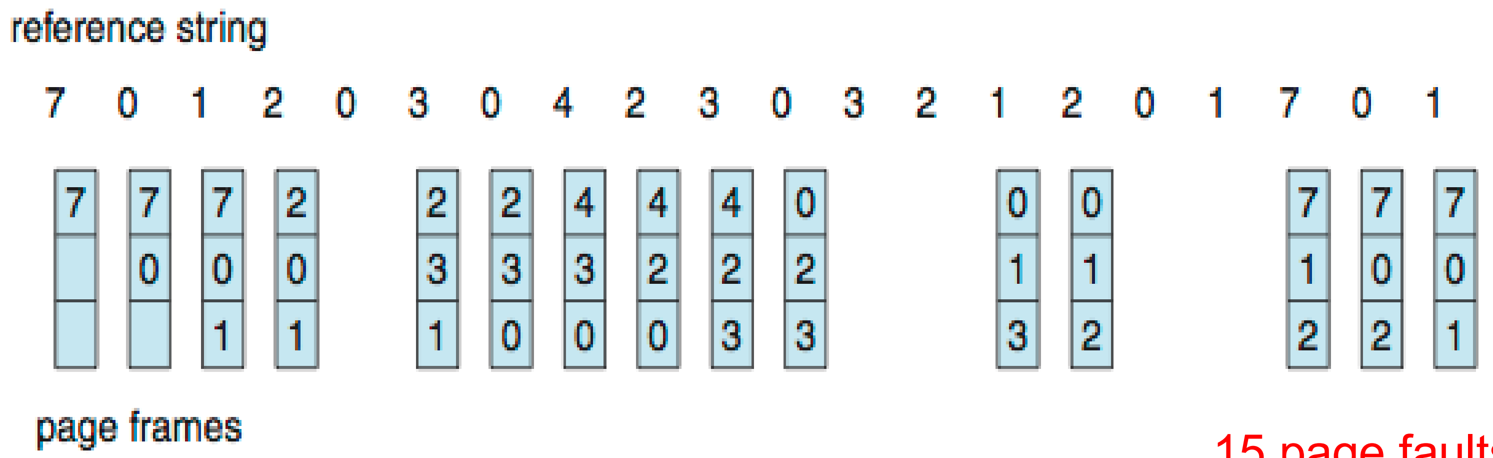
Graph of Page Faults Versus The Number of Frames



- As the number of frames available increases, the number of page faults decreases.
- With only one frame available would have a replacement with every reference
- As the number of frames increases, page faults drops to some minimal level.

First-In-First-Out (FIFO) Algorithm

- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



7	7	7	2	2	2	4	4	4	0	0	0	7	7	7
	0	0	0	3	3	3	2	2	2	1	1	1	0	0
		1	1	1	0	0	0	3	3	3	2	2	2	1

15 page faults

- The simplest page-replacement algorithm is the first-in, first-out (FIFO) algorithm.
  - Easy to understand and program.
  - Performance is not always good.
  - Suffers from Belady’s anomaly

FIFO

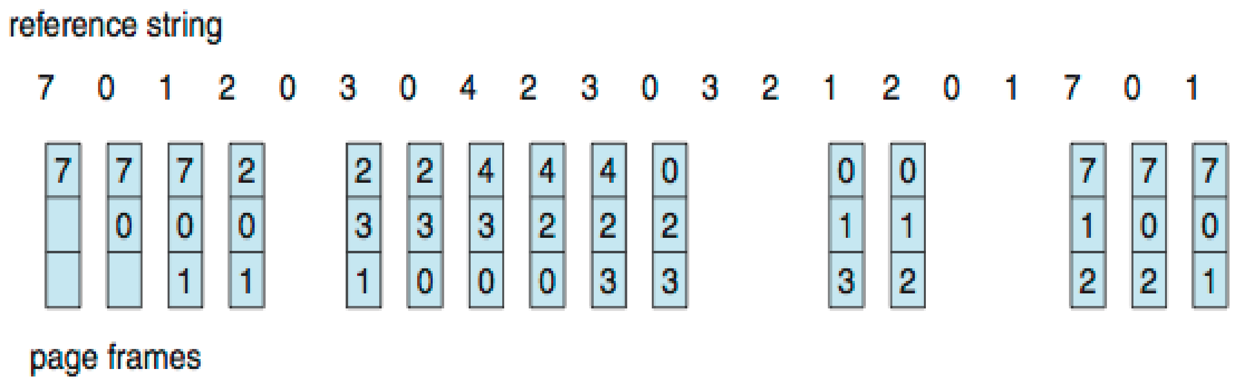
A FIFO replacement algorithm associates with each page the time when that page was brought into memory.

When a page must be replaced, the oldest page is chosen.

- How to track ages of pages?
  - Use a FIFO queue to hold all pages in memory.
  - When a page is brought into memory, insert it at the tail of the queue.
  - Replace the page at the head of the queue.

- High number of page faults
  - After replacing an active page with a new one, a fault could occur almost immediately to retrieve the active page back.
  - The page replaced could contain a heavily used variable that was initialized early and is in constant use.

FIFO



15 page faults

- For the example reference string, three frames are initially empty.
- The first three references (7, 0, 1) cause page faults and are brought into these empty frames.
- The next reference (2) replaces page 7, because page 7 was brought in first.
- Since 0 is the next reference and 0 is already in memory, we have no fault
- The first reference to 3 results in replacement of page 0, since it is now first in line.
- Because of this replacement, the next reference, to 0, will fault. Page 1 is then replaced by page 0.
- This process continues as shown in Figure
- Every time a fault occurs, we show which pages are in our three frames
- There are fifteen faults altogether

Reference string shaded in green as top row. Newly inserted pages in red

2 Frames: Page faults = 12

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

3 Frames: Page faults = 9

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

4 Frames: Page faults = 10

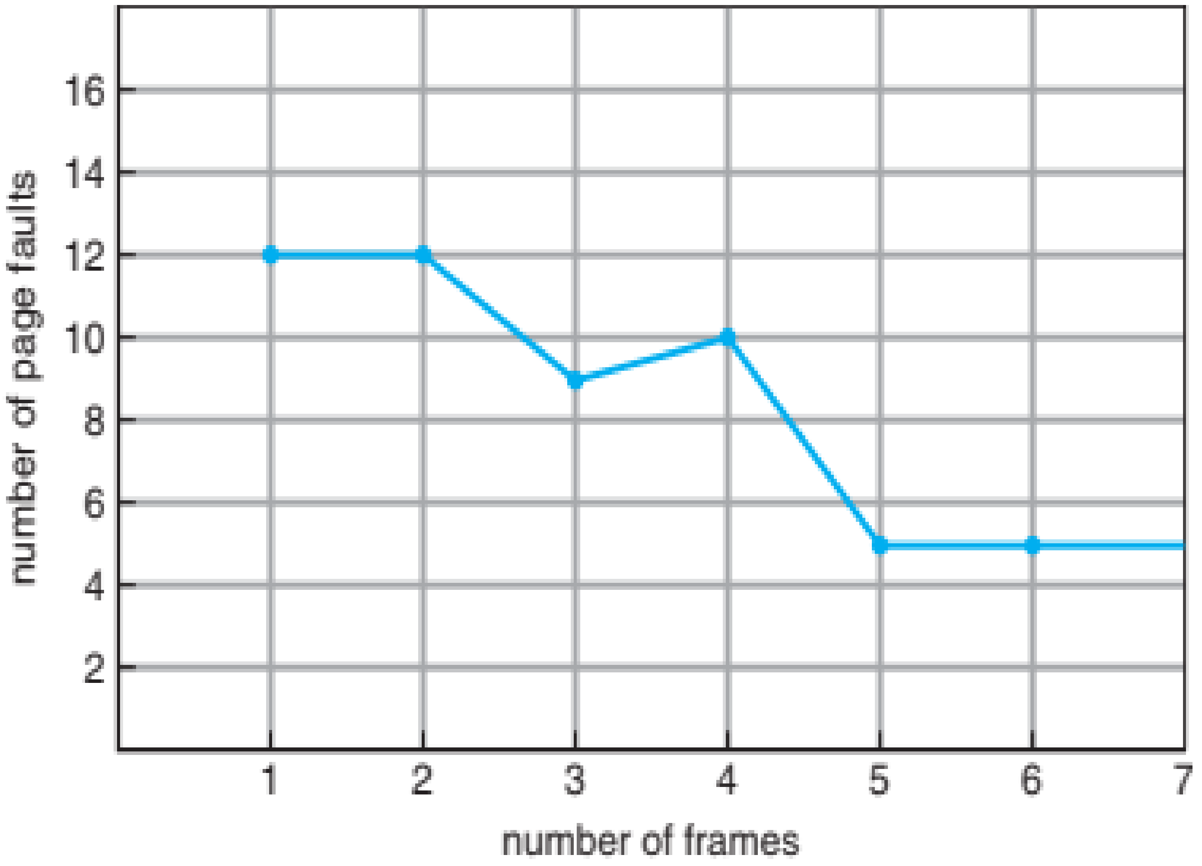
1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	1			5	5	5	5	4	4
	2	2	2			2	1	1	1	1	5
		3	3			3	3	2	2	2	2
			4			4	4	4	3	3	3

FIFO Illustrating Belady's Anomaly

Adding more frames can cause more page faults!
Reference string: 1,2,3,4,1,2,5,1,2,3,4,5

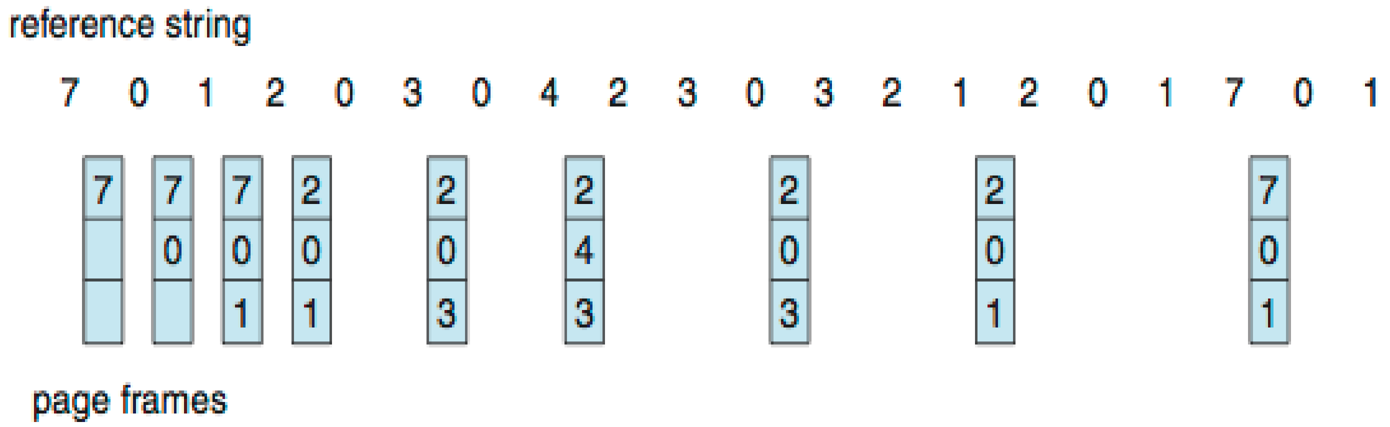
Belady's Anomaly

- Note that the number of faults for four frames (ten) is ***greater*** than the number of faults for three frames (nine)!
- **Belady's anomaly**: For some page-replacement algorithms, the page-fault rate may ***increase*** as the number of allocated frames increases

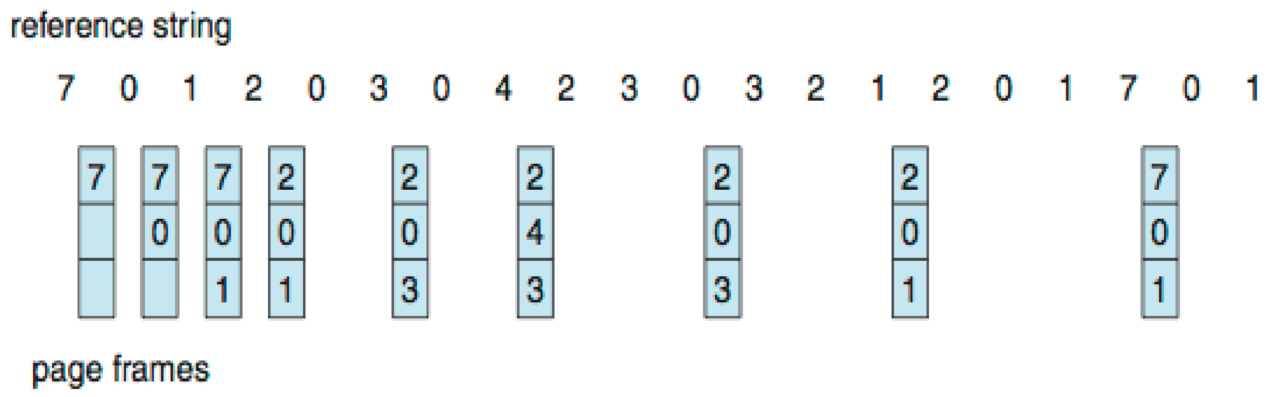


Optimal Algorithm

- Replace page that will not be used for longest period of time
  - Least number of page faults
  - **Best page-replacement algorithm**
- Not feasible to implement because future knowledge of the reference string is required
  - Can’t read the future
  - Used mainly for comparison studies for measuring how well algorithm performs



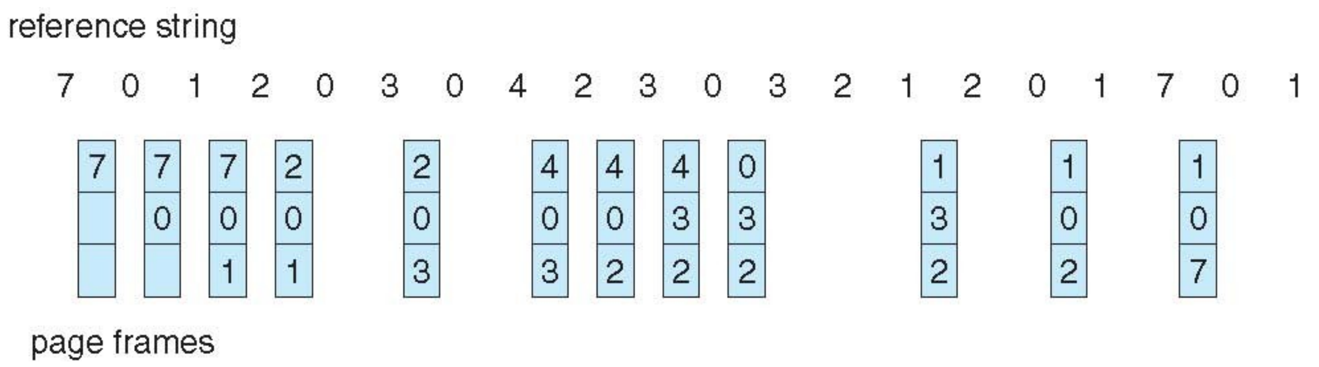
Optimal



- For the given reference string, the Optimal Page-replacement Algorithm would yield nine page faults
- All algorithms have the same page faults in filling the initial empty frames.
- The reference to page 2 replaces page 7, because page 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14.
- Since 0 is the next reference and 0 is already in memory, we have no fault
- The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again.
- This process continues as shown in figure
- Ignoring the first three that all suffer, optimal replacement is twice as good as FIFO replacement.

Least Recently Used (LRU) Algorithm

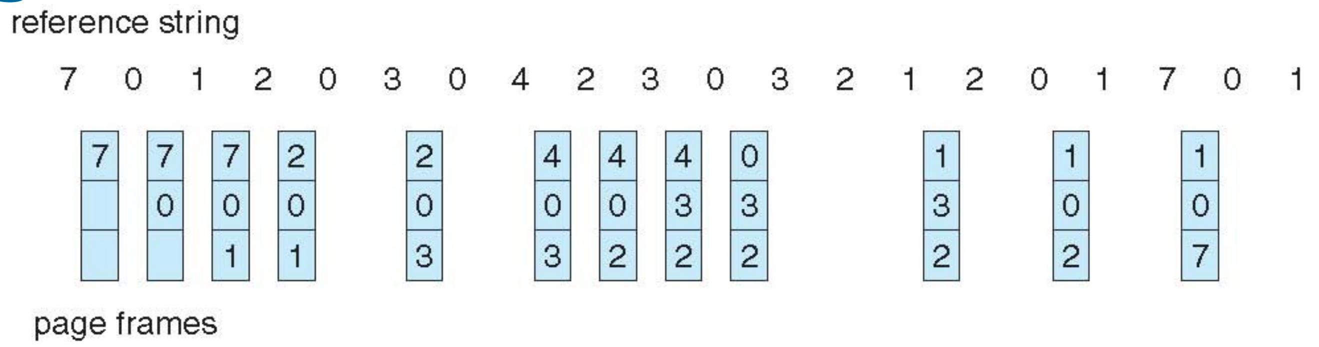
- Use past knowledge rather than future
- LRU replacement associates with each page the time of that page's last use
- Think of this strategy as the optimal page-replacement algorithm looking backward in time, rather than forward
- Replace page that has not been used for the longest period of time



12 faults – better than FIFO but worse than OPT

- Generally good algorithm and frequently used
- But how to implement?

LRU



12 page faults

- All algorithms have same page faults in filling the initial empty frames.
- The reference to page 2 replaces page 7, the least recently of the 3
- Since 0 is next, and 0 is already in memory, we have no fault
- The reference to page 3 replaces page 1, as page 1 was the earliest of the three pages in memory referenced.
- When reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used.
- When it then faults for page 2, the LRU algorithm replaces page 3, since it is now the least recently used of the three pages in memory.
- LRU replacement with twelve faults is much better than FIFO replacement with fifteen.

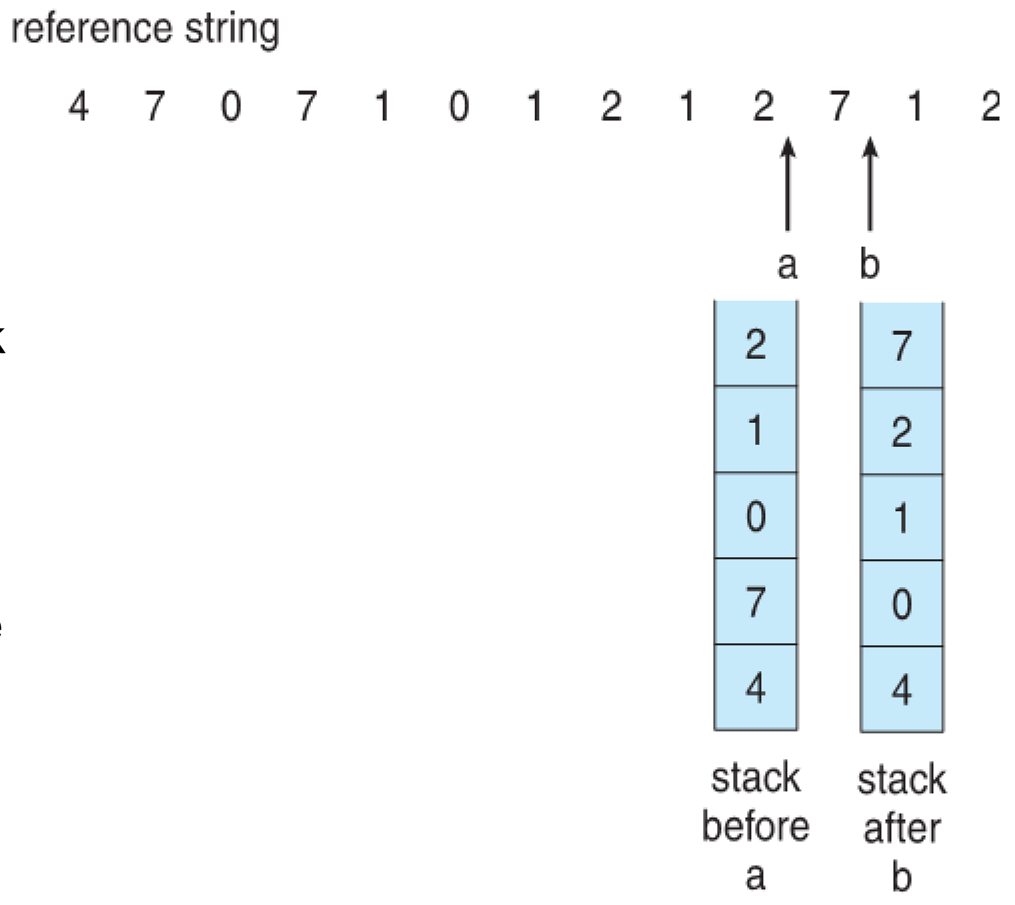
Implementing LRU Algorithm

- **The LRU policy is often used as a page-replacement algorithm and is considered to be good**.
    - ***How*** to implement LRU replacement?
    - May require substantial hardware assistance.
- Counter implementation
    - Logical clock or counter is incremented for every memory reference.
    - Replace the page with the smallest time value (least)
- Stack implementation
    - Keep a stack of page numbers.
    - When a page is referenced:
        - Move it to the top.
        - Best to use a doubly linked list. Requires changing 6 pointers at worst
- LRU and OPT are cases of **stack algorithms** that don’t have Belady’s Anomaly

Use of a Stack to Record Most Recent Page References

- Whenever a page is referenced, it is removed from the stack and put on the top

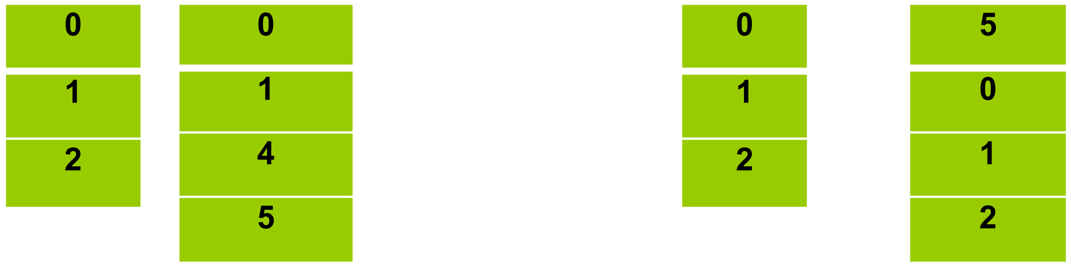
- Most recently used page is always at the top of the stack and the least recently used page is always at the bottom



Stack-based Algorithm

- Set of pages in memory for N frames is always a subset of the set of pages that would be in memory with N + 1 frames.
- For LRU replacement, the set of pages in memory would be the *n* most recently referenced pages. If the number of frames increases then these *n* pages will still be the most recently referenced and so, will still be in the memory.
- FIFO does not follow a stack page replacement policy and therefore suffers Belady's Anomaly.

FIFO LRU



The diagram illustrates that given the set of pages i.e. {0, 1, 2} in 3 frames of memory is not a subset of the pages in memory – {0, 1, 4, 5} with 4 frames and it is a violation in the property of stack based algorithms.

LRU Approximation Algorithms

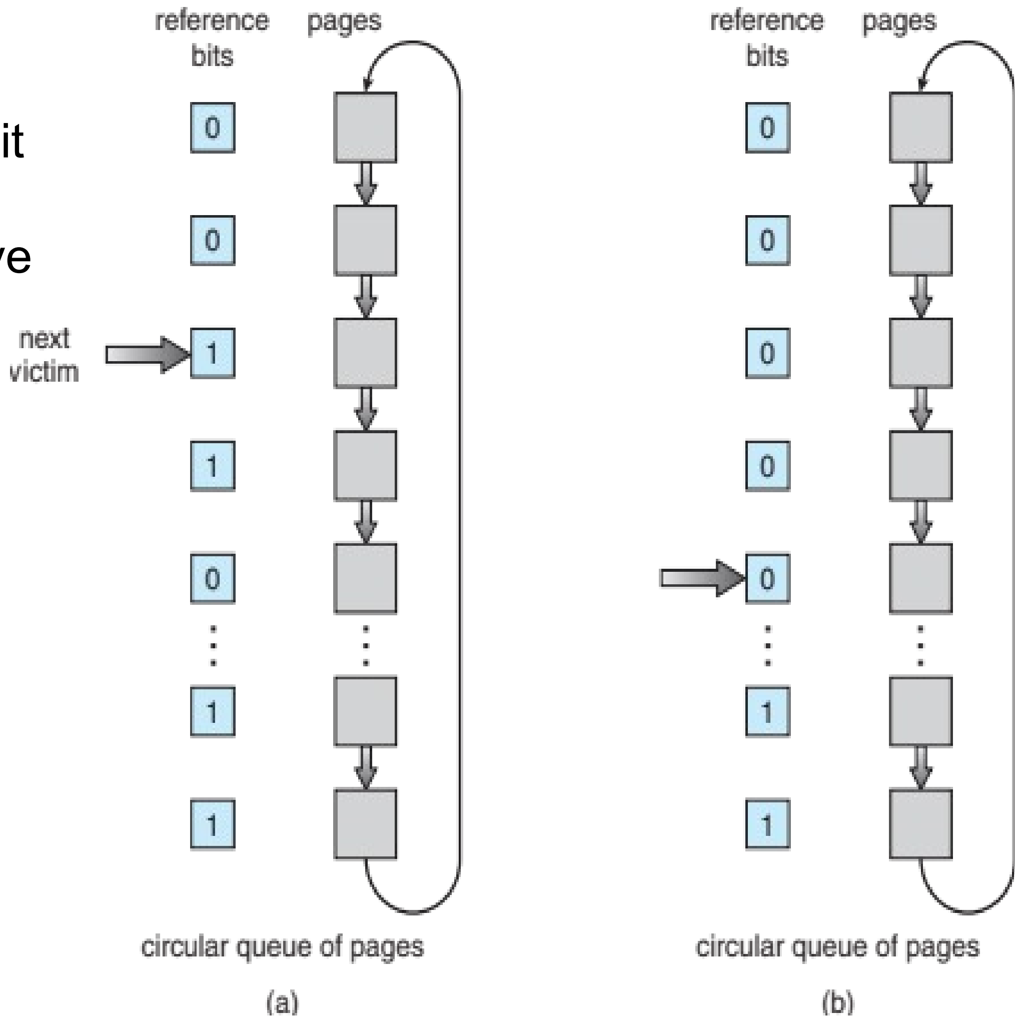
- LRU needs special hardware and still slow
- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)
    - We do not know the order, however
- **Second-chance algorithm**
    - FIFO, plus hardware-provided reference bit
    - Also referred as **Clock algorithm**
  - When all bits are set, the pointer cycles through the whole queue, giving each page a second chance. It clears all the reference bits before selecting the next page for replacement.
  - Degenerates to FIFO replacement if all bits are set.

Second-Chance (clock) Page-Replacement Algorithm

Algorithm:

If page to be replaced has

- reference bit = 0 -> replace it
- reference bit = 1 then:
  - set reference bit 0, leave page in memory
  - replace next page, subject to same rules

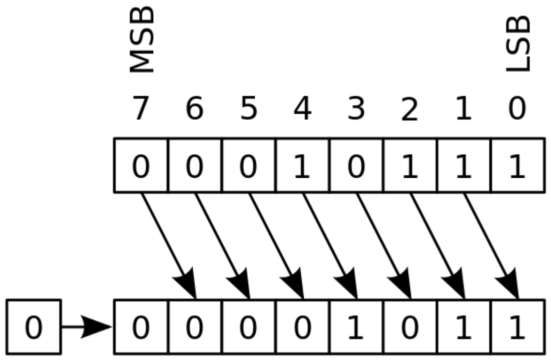


Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) in concert
- Take ordered pair (reference, modify):
  - (0, 0) neither recently used not modified – best page to replace
  - (0, 1) not recently used but modified – not quite as good, must write out before replacement
  - (1, 0) recently used but clean – probably will be used again soon
  - (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement
- When page replacement called for, use the clock scheme but use the four classes and replace page in lowest non-empty class
  - Might need to search circular queue several times

Additional-Reference-Bits Algorithm

- Each page has an 8-bit shift register
- The bit associated with each page referenced is set (to 1) by the hardware
- At regular intervals, (say, every 100 milliseconds), OS shifts the reference bit into the most significant bit (MSB), shifts the other bits right (÷2) by 1 bit, and discards the least significant bit (LSB).
- These 8-bit shift registers contain the history of page use for the last eight time periods
- If the shift register contains 00000000, then the page has not been used for eight time periods. A page that is used at least once in each period has a shift register value of 11111111. A page with a register value of 11000100 has been used more recently than one with a value of 01110111.



The page with the lowest number is the LRU page,

Page-Buffering Algorithms

- Keep a pool of free frames
- When a page fault occurs, a victim frame is chosen as before. However, the desired page is read into a free frame from the pool before the victim is written out.
- This procedure allows the process to restart as soon as possible, without waiting for the victim page to be written out.
- When the victim is later written out, its frame is added to the free-frame pool

Expansion ideas:

- Maintain list of modified pages
  - When paging device otherwise idle, a modified page is selected and is written to the secondary storage
- Possibly, keep free frame contents intact and remember what is in them
  - Old page can be reused directly from the free-frame pool if it is needed before that frame is reused. No I/O is needed.

Applications and Page Replacement

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge – i.e. databases
- **Memory intensive applications** can cause double buffering
  - OS keeps copy of page in memory as I/O buffer
  - Application keeps page in memory for its own work
- **Raw Disk**
  - Operating system can give special programs the ability to use a secondary storage partition as a large sequential array of logical blocks, without any file-system data structures.
  - This array is sometimes called the **raw disk**, and I/O to this array is termed **raw I/O**.

Allocation of Frames

- Each process needs ***minimum*** number of frames
- The minimum number of frames per process is defined by the computer architecture
- The maximum number is defined by the amount of available physical memory.

Two major **allocation schemes**

1. **fixed allocation**
2. **priority allocation**
    - high-priority process gets more memory to speed up its execution

- Many variations

Fixed Allocation

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - Keep some as free frame buffer pool
- Proportional allocation – Allocate according to the size of process
  - Dynamic as degree of multiprogramming, process sizes change
    - s_i = size of process p_i
    - $S = \sum s_i$
    - m = total number of frames
    - a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$m = 62$

$s_1 = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \times 62 \approx 4$

$a_2 = \frac{127}{137} \times 62 \approx 57$

Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - Process execution time can vary greatly
  - Pages in memory for a process depends the paging behavior of all processes.
  - Greater throughput, so more commonly used method
- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

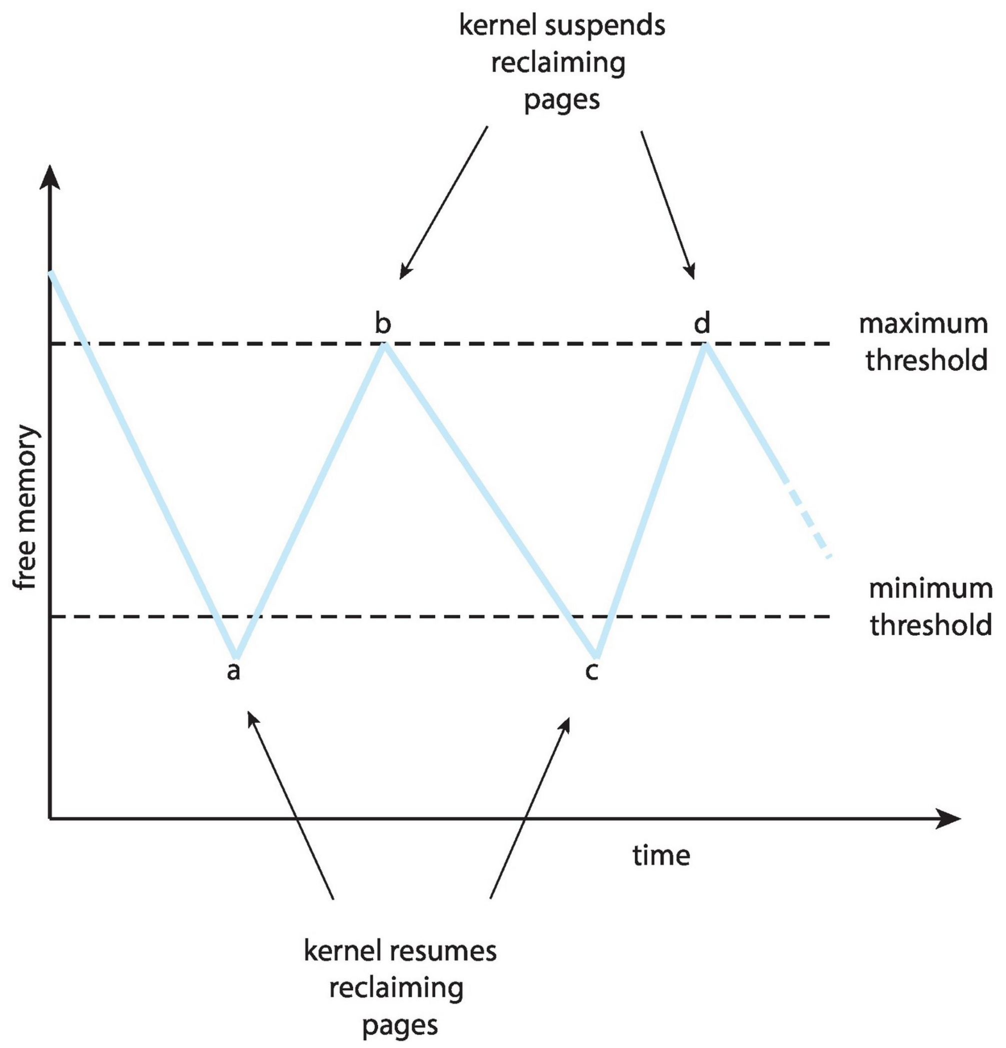
Reclaiming Pages

A strategy to implement **global page-replacement** policy

- All memory requests are satisfied from the free-frame list
- When the list falls below a minimum threshold, a kernel routine known as **reapers** is triggered that reclaims pages from all processes (excluding kernel.)
- Reaper routine suspends when free memory reaches the maximum threshold
- This strategy attempts to ensure there is always sufficient free memory to satisfy new requests.
- Aggressive measures if reaper is unable to maintain the level. Examples:
    - Suspend second-chance algorithm
    - Linux's out-of-memory killer routine terminate a process & reclaim

Kernel Reaper typically use LRU approximation for page replacement.

Reclaiming Pages Example



a. Point a: free memory drops below minimum threshold
   Kernel begins reclaiming pages and adding them to free-frames list
b. Point b: maximum threshold reached.
   Reaper suspended.
   Additional requests for memory drops free memory
c. Point c: free memory drops below minimum threshold again
   Page reclaiming resumes
d. Point d: maximum threshold reached.
   Reaper suspended.

Thrashing

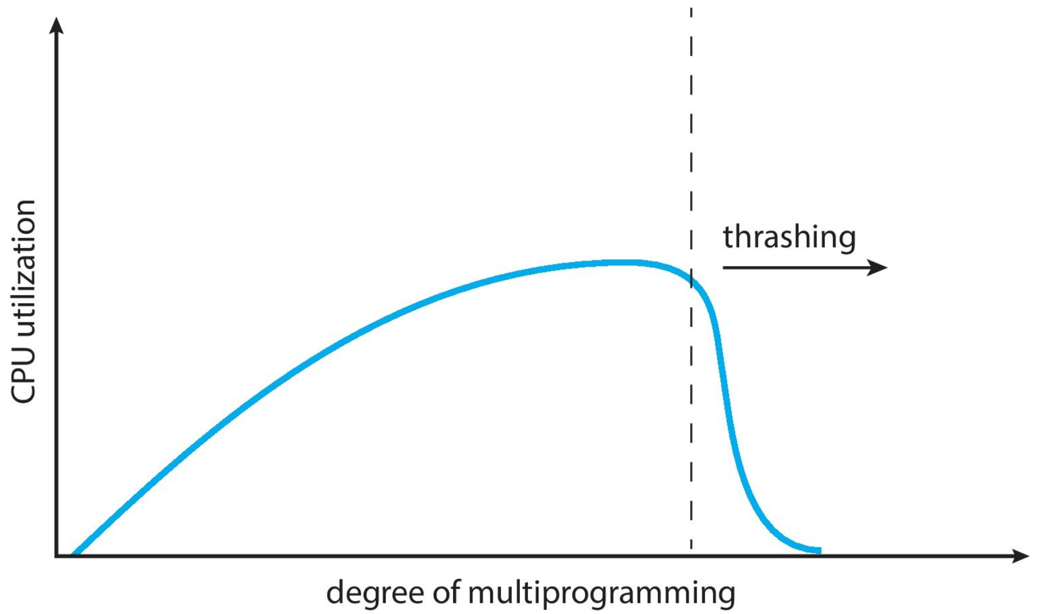
OS monitors CPU utilization. If CPU utilization is too low, OS increases the degree of multiprogramming by introducing a new process from ready queue.

- A global page-replacement algorithm is used. The new process starts faulting, replaces pages from other processes, which needing their pages, also fault
- Processes queue up for paging device. Ready queue empties
- CPU utilization degreases
- CPU scheduler sees the decreasing CPU utilization and ***increases*** the degree of multiprogramming by bringing new processes, causing more page faults and longer queue
- CPU utilization drops even further, and the CPU scheduler tries to increase the degree of multiprogramming even more.

- This high paging activity is called **thrashing**.
- A process is thrashing if it is spending more time paging than executing

Thrashing (Cont.)

- **Thrashing.** A process is busy swapping pages in and out



- As the degree of multiprogramming increases, CPU utilization also increases, until a maximum is reached.
- If the degree of multiprogramming is increased even further, thrashing sets in, and CPU utilization drops sharply.
- To increase CPU utilization and stop thrashing, ***decrease*** the degree of multiprogramming.

Thrashing Problems

- Problems with thrashing:
  - The system throughput plunges.
  - The page fault rate increase tremendously.
  - Effective memory-access time increases.
- Limit thrashing:
  - Use a **local replacement** algorithm (or Priority replacement algorithm)
    - If one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash as well.
- Prevent thrashing:
  - Provide a process with as many frames as it needs.
    - But how do we know how many frames it "needs"?
      - By looking at how many frames a process is actually using

Locality Model of Process Execution

- **Locality**
  - Locality defines the set of pages that are actively used together.
    - A running program is composed of several different localities that may overlap
    - As a process executes, it migrates from one locality to another
      - For example, when a function is called, it defines a new locality.
      - When function returns, it leaves this locality, since local variables and instructions of the function are no longer in active use
- Allocate enough frames to a process to accommodate its current locality.
  - It will fault for the pages in its locality until all these pages are in memory;
  - Then, it will not fault again until it changes localities.
- If we do not allocate enough frames to accommodate the size of the current locality, the process will thrash, since it cannot keep in memory all the pages that it is actively using.

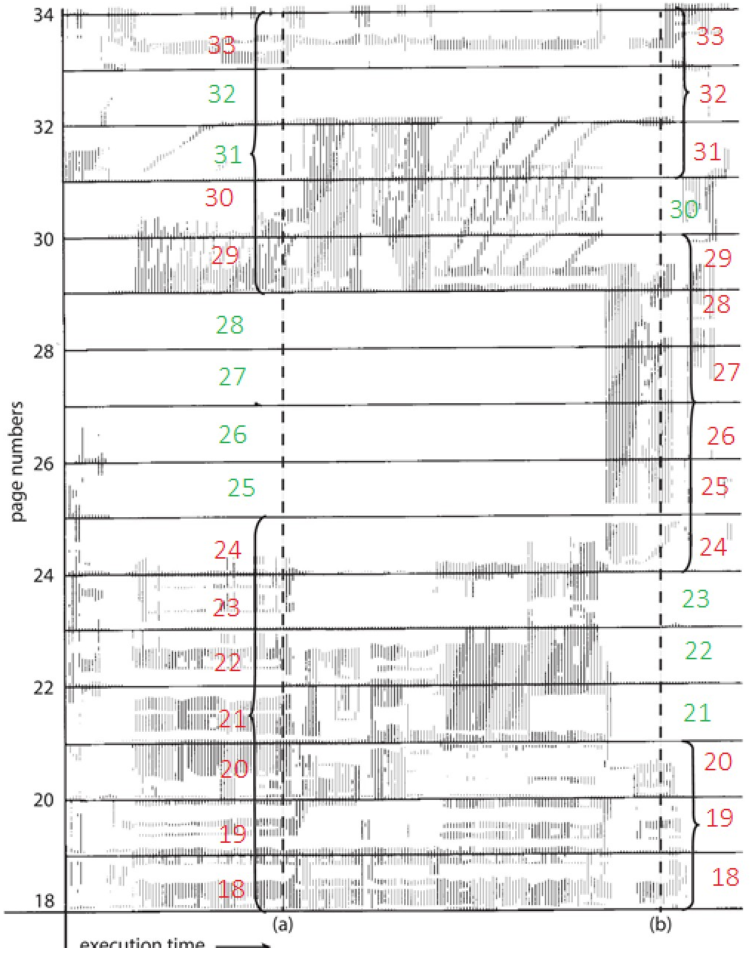
.

Process Locality at times t_a & t_b

At time (a) locality is the set of pages:
{18, 19, 20, 21, 22, 23, 24, 29, 30, 33}
At time (b) locality is the set of pages:
{18, 19, 20, 24, 25, 26, 27, 28, 29, 31, 32, 33}

Overlap:
{18, 19, 20, 24} are part of both localities

- Principle behind cashing.



Working-Set Model

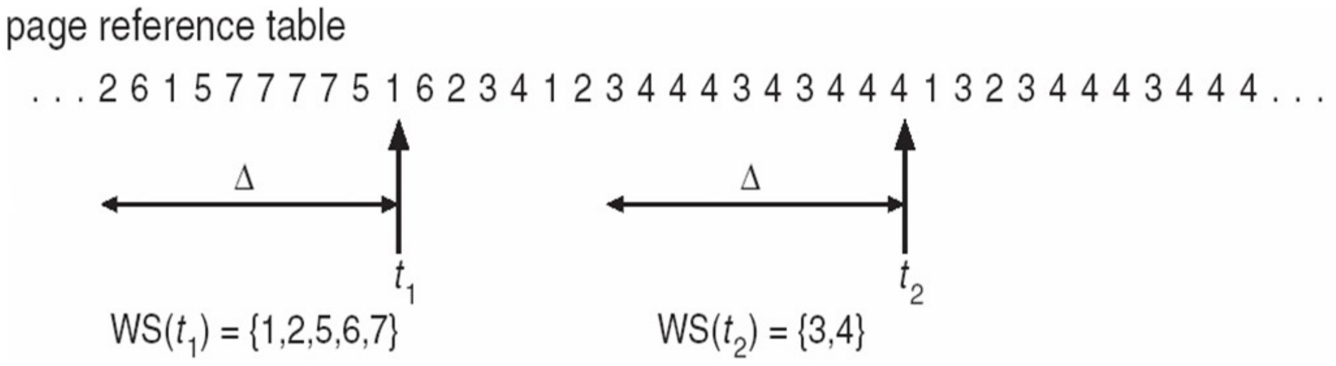
- **Working-set model** is based on the assumption of **locality**
- Define $\Delta \equiv$ working-set window
- **Working-set:**
  - The set of pages in the most recent Δ page references.
    - It is an approximation of the locality of the process
    - If a page is in active use, it will be in the working set
- WSS_i (working set of Process P_i) is the number of pages referenced by it in the most recent Δ time
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- Working-Set Size is the number of frames a process needs
- Total demand D for frames for all processes is the sum of all WS sizes

  $D = \Sigma\ WSS_i$

Working-Set Strategy Benefits

- Prevents thrashing
  - Thrashing occurs if total demand *D* > available frames *m*
  - Policy: if *D* > m, then suspend one of the processes
- Optimizes CPU utilization
  - If there are enough extra frames, initiate another process
  - Thus, it keeps the degree of multiprogramming as high as possible.

Working-Set Model (Cont.)



The working-set window is a moving window.

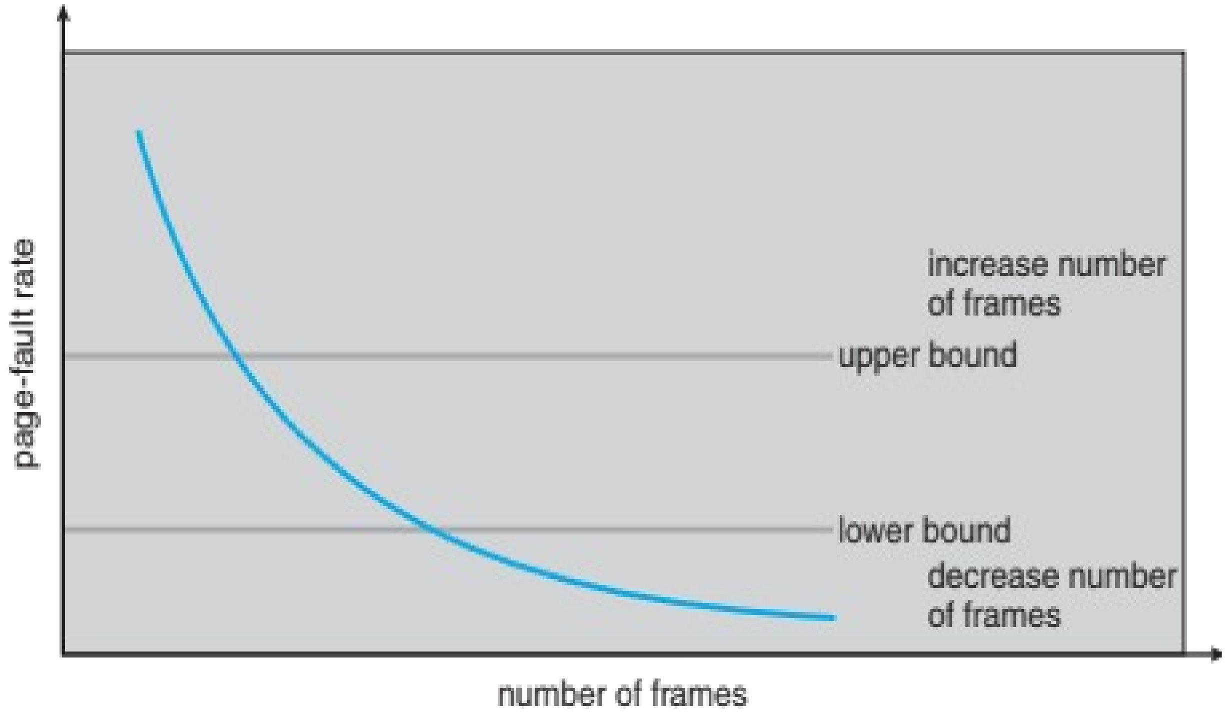
Given the sequence of memory references shown and if Δ = 10 memory references,

The working set at time t_1 is {1, 2, 5, 6, 7}.

By time t_2, the working set has changed to {3, 4}.

Page-Fault Frequency

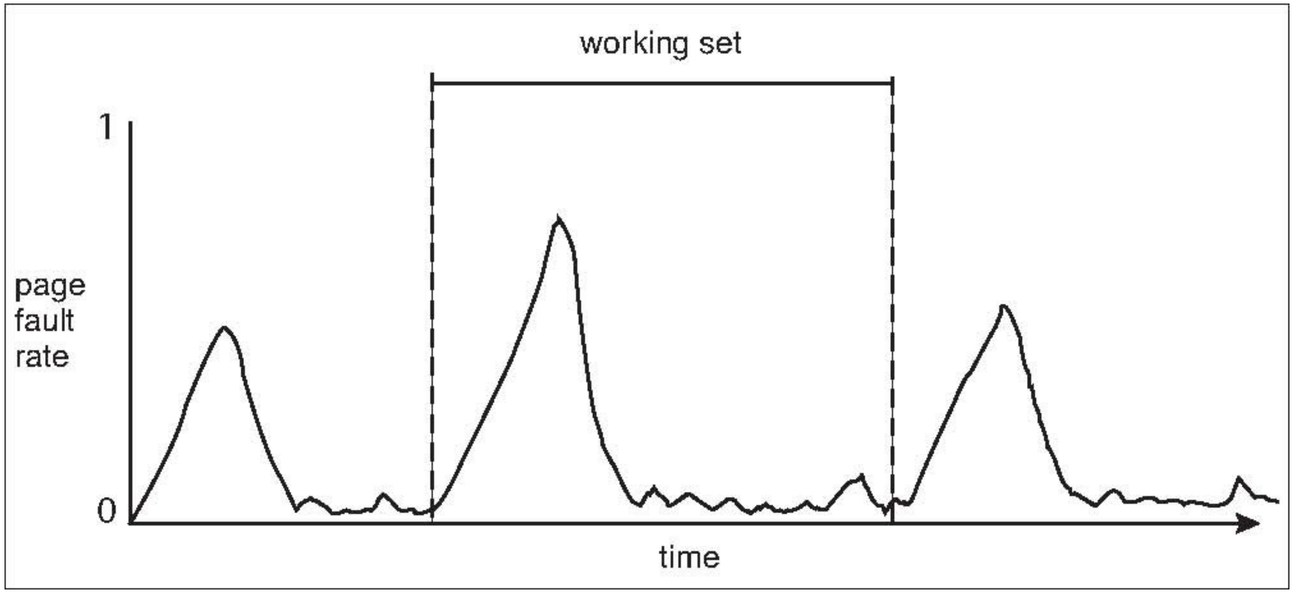
- More direct approach than WS model
- Establish "acceptable" **page-fault frequency** (**PFF**) rate and use local replacement policy
  - If page-fault rate falls below the lower limit, process may have too many frames. Remove a frame.
  - If actual page-fault rate exceeds the upper limit, allocate another frame



Working Sets and Page Fault Rates

Direct relationship between working set of a process and its page-fault rate

- Assuming there is sufficient memory to store the working set of a process (that is, the process is not thrashing), the page-fault rate of the process will transition between peaks and valleys over time.
- A peak in occurs when we begin demand-paging a new locality.
- Once the working set of the new locality is in memory, page-fault rate falls.
- When the process moves to a new working set, the page-fault rate rises toward a peak once again.



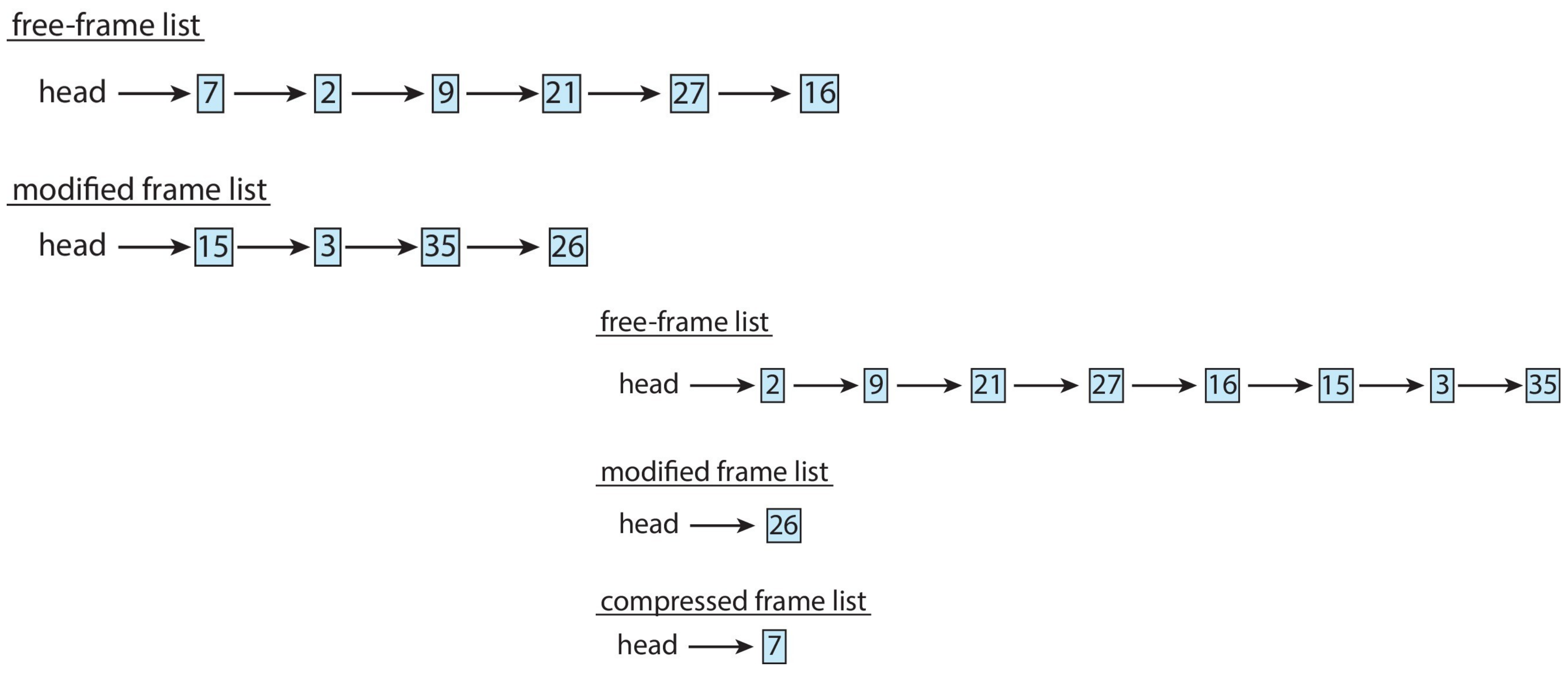
Memory Compression

- **Memory compression** -- rather than paging out modified frames to swap space, we compress several frames into a single frame, enabling the system to reduce memory usage without resorting to swapping pages.
- Mobile systems do not support swapping or swapping pages.
  - Memory compression is the strategy used in Android and iOS
- Faster than paging

- Consider the following free-frame-list consisting of 6 frames
  - Assume that this number of free frames falls below a certain threshold that triggers page replacement.
  - The replacement algorithm (say, an LRU approximation algorithm) selects four frames -- 15, 3, 35, and 26 to place on the free-frame list.
  - It first places these frames on a modified-frame list.

Memory Compression (Cont.)

- Frames 15, 3, & 35 are compressed and stored in frame 7, which is then stored in the list of compressed frames.
- Frames 15, 3, and 35 are moved to the free-frame list
- If a compressed page is referenced, compressed frame is decompressed, restoring the three pages 15, 3, & 35 in memory.



Allocating Kernel Memory

- An OS kernel typically has special memory allocation needs.

  The OS may require kernel code and data to be entirely resident in physical memory at all times, even though virtual memory is implemented for user processes.
- Kernel memory is often allocated from a free-memory pool different from the list used to satisfy ordinary user-mode processes, because:
  - OS may need memory for data structures of varying sizes. To minimize internal fragmentation, do not subject kernel code or data to paging
  - Certain hardware devices may require memory residing in physically contiguous pages.
- To facilitate conservation of memory, usually there are special memory-allocation techniques and methods available to the kernel. Two such strategies for managing free memory that is assigned to kernel processes:
  - Buddy system and
  - Slab allocation

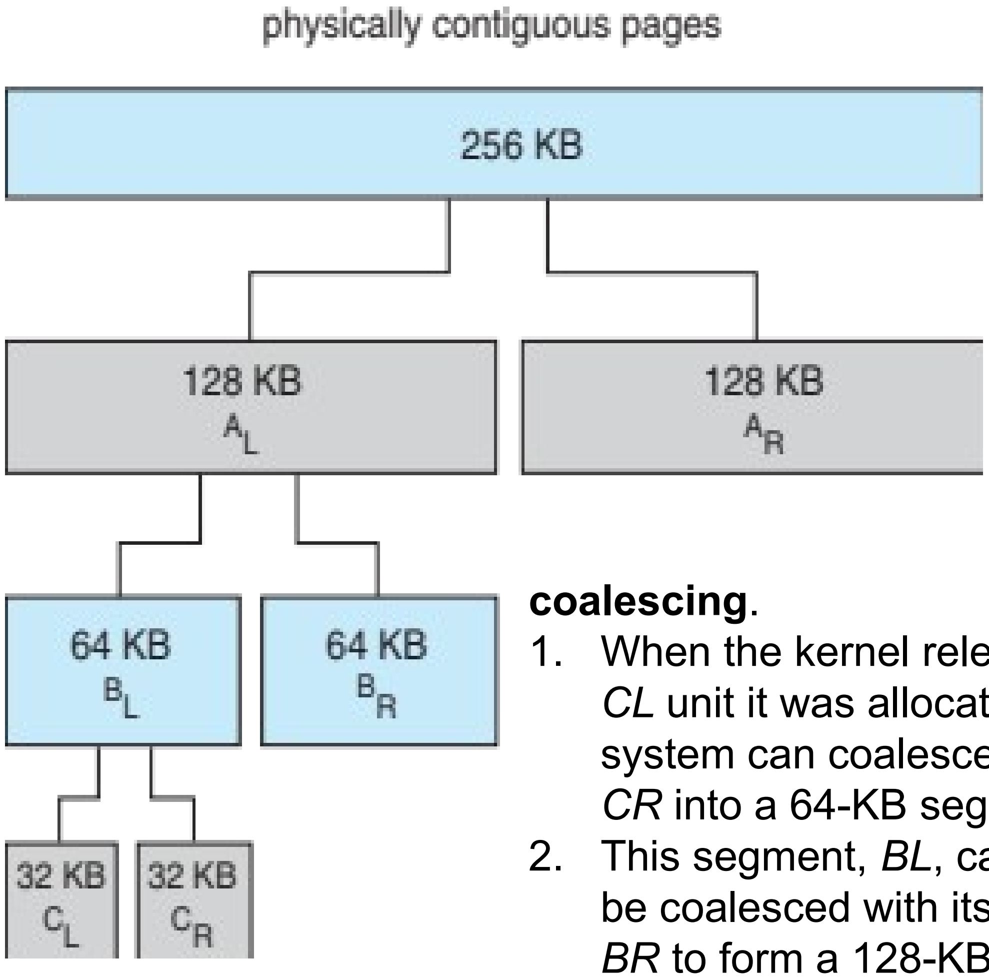
Buddy System

- 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
- Advantage – **quickly coalesce** unused chunks into larger chunk
- Disadvantage - rounding up to the next highest power of 2 is very likely to cause **fragmentation** within allocated segments. Example: a 33-KB request can only be satisfied with a 64-KB segment.

Buddy System Allocator

Size of memory segment is initially 256 KB

1. Kernel requests 21 KB of memory.
2. The segment is initially divided into two **buddies** —which we will call *AL* and *AR*—each 128 KB in size.
3. One of these buddies is further divided into two 64-KB buddies— *BL* and *BR*.
4. The next-highest power of 2 from21 KB is 32 KB. Either *BL* or *BR* is again divided into two 32-KB buddies, *CL* and *CR*.
5. One of these buddies is used to satisfy the 21-KB request.



coalescing.

1. When the kernel releases the *CL* unit it was allocated, the system can coalesce *CL* and *CR* into a 64-KB segment.
2. This segment, *BL*, can in turn be coalesced with its buddy *BR* to form a 128-KB segment.
3. Ultimately ends up with the original 256-KB segment.

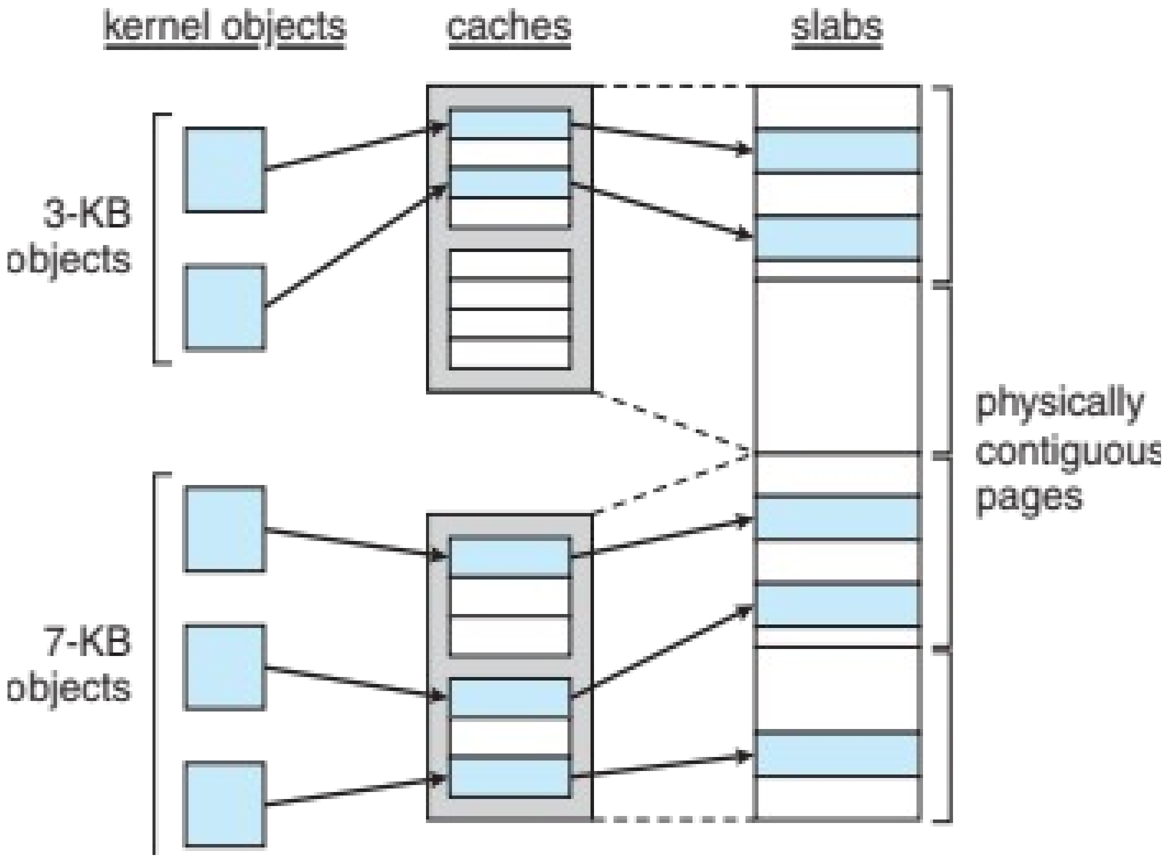
Slab Allocator

- Alternate strategy for allocating kernel memory
- There is a single cache for each unique kernel data structure, Example:
    - a separate cache for the data structure representing process descriptors,
    - a separate cache for file objects,
    - a separate cache for semaphores, and so forth.
- Each cache populated with **objects** that are instantiations of the kernel data structure the cache represents

Main benefits:

1. **No** memory is wasted due to **fragmentation**
2. Memory **requests** can be **satisfied quickly**

Slab Allocation



A **Slab** is made of one or more physically contiguous pages

A **Cache** consists of one or more slabs

Separate cache for each **kernel object** type

Slab Allocator in Linux

- For example process descriptor is of type `struct task_struct`
- Approx 1.7KB of memory
- New task -> allocate new struct from cache
  - Will use existing free `struct task_struct`
- Slab can be in three possible states
  1. Full – all used
  2. Empty – all free
  3. Partial – mix of free and used
- Upon request, slab allocator
  1. Uses free struct in partial slab
  2. If none, takes one from empty slab
  3. If no empty slab, create new empty

Other Considerations

- Prepaging
- Page size
- TLB reach
- Inverted page table
- Program structure
- I/O interlock and page locking

Page Size

- Page sizes are invariably powers of 2, usually in the range **2^{12} (4,096 bytes) to 2^{22} (4,194,304 bytes)**
- Defined by the computer hardware, sometimes OS designers have a choice, especially if running on custom-built CPU
- Arguments for small page size:
  - Minimize internal fragmentation,
  - Match program locality more accurately
  - Reduced I/O
  - Better resolution
- Favoring large page size:
  - Reduced page table size
  - Minimize page faults
- Historical trend is toward larger page sizes, even for mobile systems.

TLB Reach

- TLB Reach - **The amount of memory accessible from the TLB**
- **TLB Reach = (TLB Size) x (Page Size)**
- Ideally, the working set of each process is stored in the TLB
  - Otherwise there is a high degree of page faults
- 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

Program Structure

Demand paging is designed to be transparent to the user program. But, system performance can be improved if the user (or compiler) has an awareness of the underlying demand paging.

- Program structure
  - `int[128][128] data;`
  - Each row is stored in one page
- Program 1

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

128 x 128 = 16,384 page faults

- Program 2

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

128 page faults

1. The array is stored row major; that is, the array is stored data[0][0], data[0][1], · · ·, data[1][0], data[1][1], · · For pages of 128 words, each row takes one page. Thus, the preceding code zeros one word in each page, then another word in each page, and so on. If the operating system allocates fewer than 128 frames to the entire program, then its execution will result in 128 × 128 = 16,384 page faults.

2. This code zeros all the words on one page before starting the next page, reducing the number of page faults to 128.

End of Chapter 10