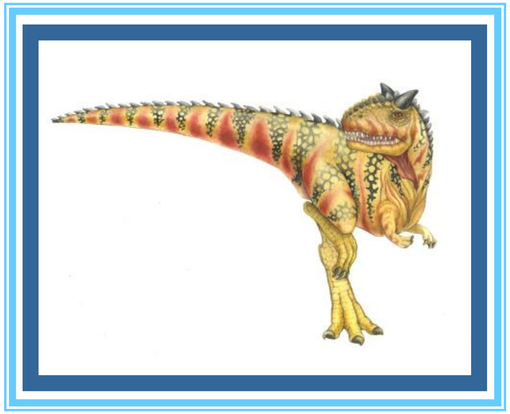
Chapter 10: Virtual Memory



Chapter 10: Virtual Memory

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

Objectives

- Define virtual memory and describe its benefits.
- Illustrate how pages are loaded into memory using demand paging.
- Apply the FIFO, optimal, and LRU page-replacement algorithms.
- Describe the working set of a process, and explain how it is related to program locality.
- Describe how Linux, Windows 10, and Solaris manage virtual memory.
- Design a virtual memory manager simulation in the C programming language.

Background

- Code needs to be in memory to execute, but entire program is rarely used
  - Error handling code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
  - Program is no longer constrained by limits of physical memory
  - Each program takes less memory while running -> more programs run at the same time
    - 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

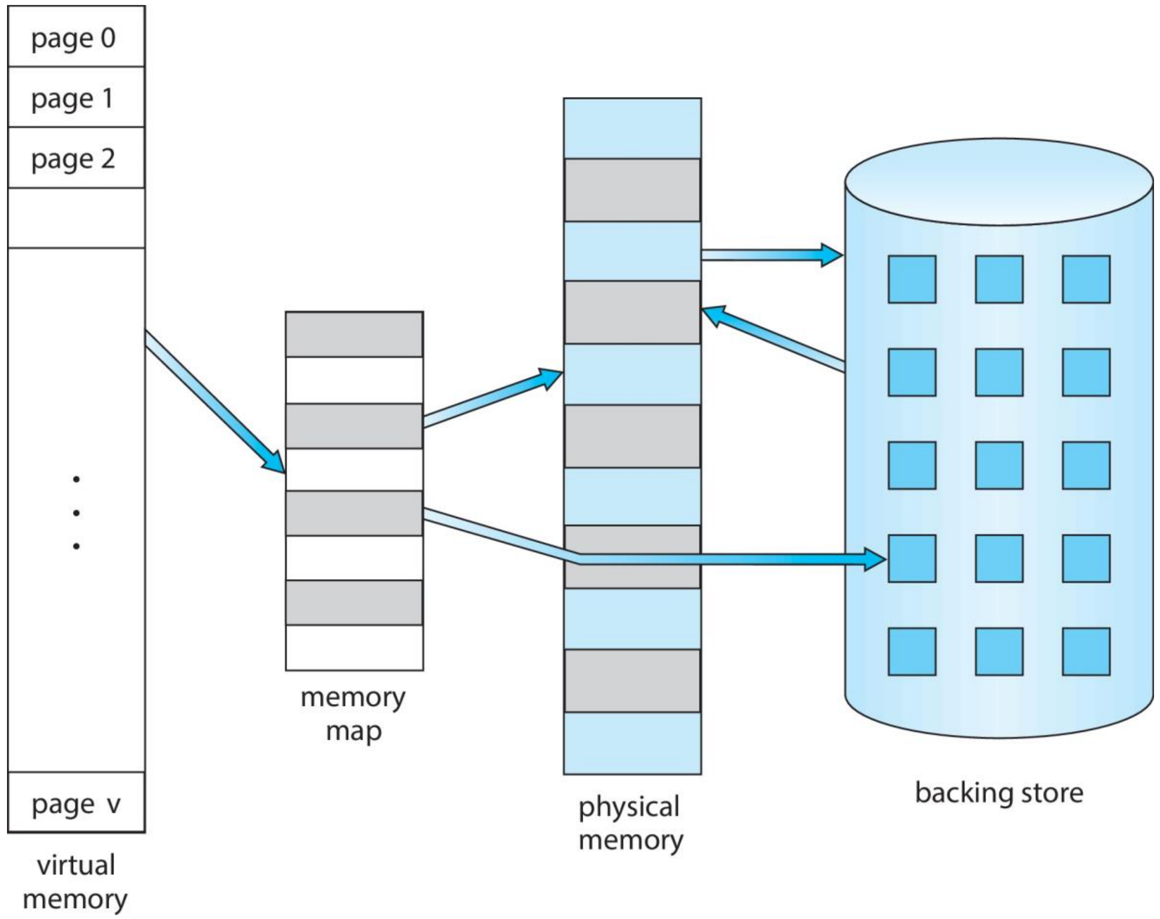
Virtual memory

- **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 address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes

Virtual memory (Cont.)

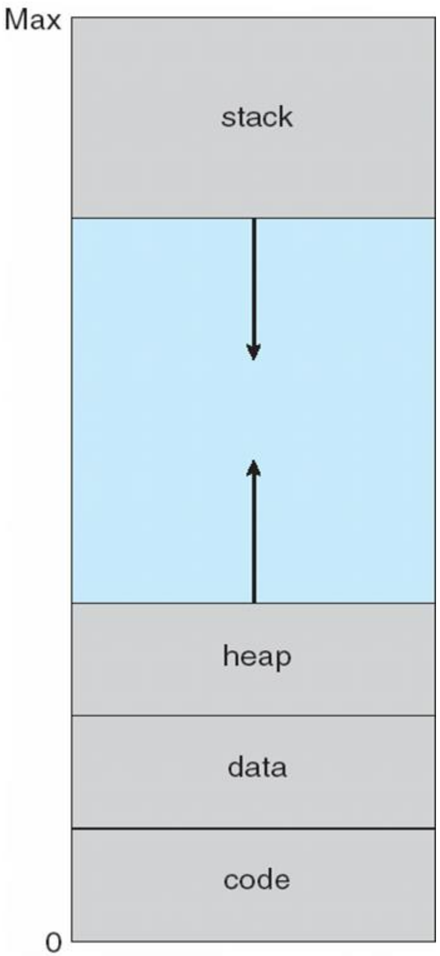
- **Virtual address space** – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical address
- Virtual memory can be implemented via:
  - Demand paging

Virtual Memory That is Larger Than Physical Memory

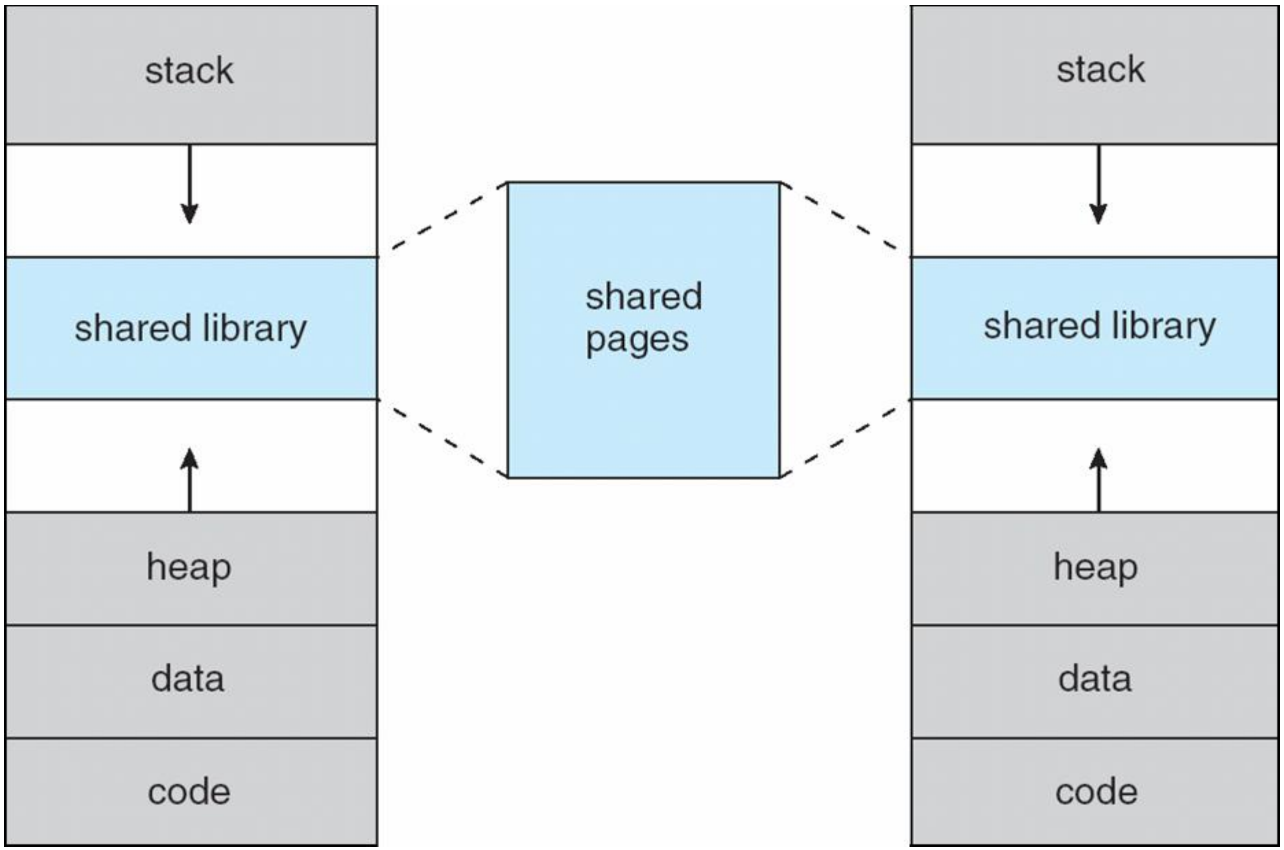


Virtual-address Space

- Usually design logical address space for stack to start at Max logical address and grow "down" while heap grows "up"
  - Maximizes address space use
  - Unused address space between the two is hole
    - No physical memory needed until heap or stack grows to a given new page
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc.
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during `fork()`, speeding process creation



Shared Library Using Virtual Memory



Demand Paging

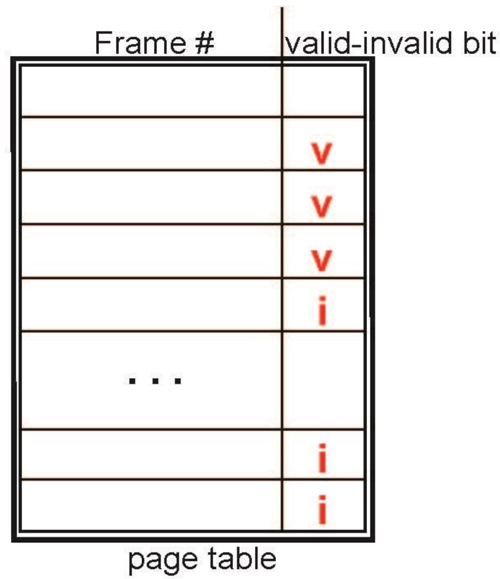
- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Page is needed $\Rightarrow$ reference to it
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ bring to memory

Basic Concepts

- Load a page in memory only when it is needed
- We need some form of hardware support to distinguish which pages are already **memory resident** and which are in secondary storage.
- Need to detect and load the page into memory from storage
  - Without changing program behavior
  - Without programmer needing to change code
- The valid-invalid bit scheme described before can be used for this purpose
  - Valid: the associated page is both legal and in-memory
  - Invalid: the page is not in-memory, but could be
    - in secondary storage, or
    - illegal (not in the logical address space)

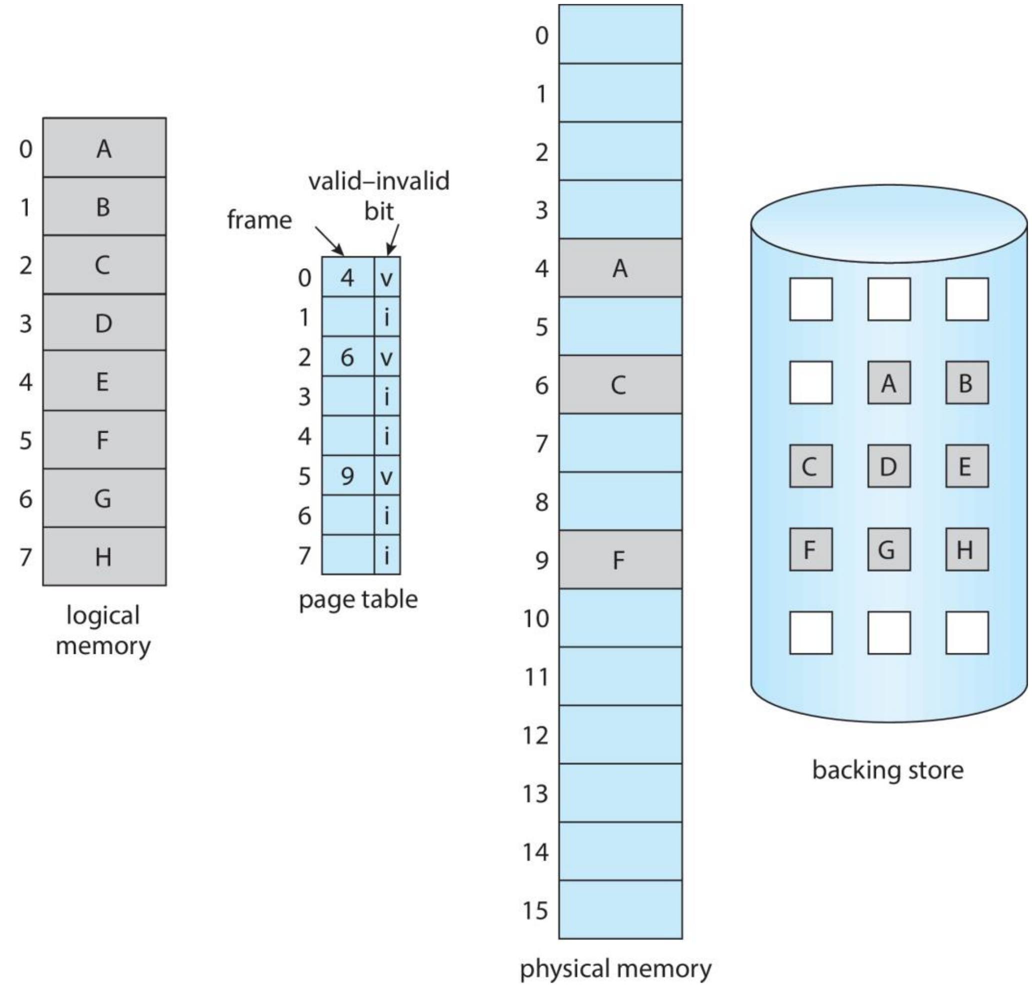
Valid-Invalid Bit

- 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

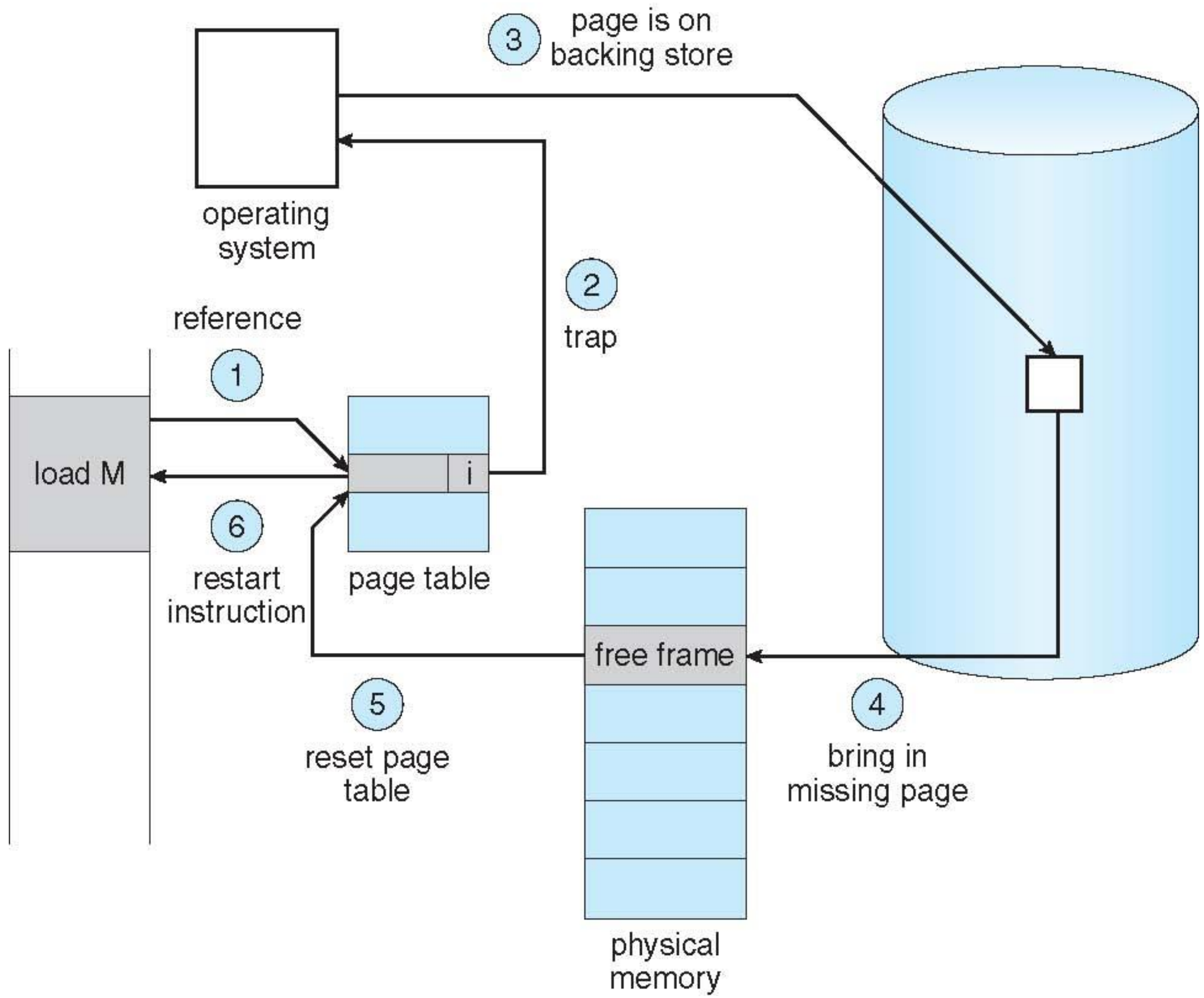
Page Table When Some Pages Are Not in Main Memory



Steps in Handling Page Fault

1. If there is a reference to a page, first reference to that page will trap to operating system (since it's invalid)
   - Page fault
2. Operating system looks at an internal table (memory limits, kept with the PCB) to decide:
   - Invalid reference ⇒ abort
   - Just not in memory ⇒ page it in
3. Find a free frame from the free-frame list
4. Swap page into frame via scheduled disk operation
5. Reset tables to indicate page now in memory
   Set validation bit = **v**
6. Restart the instruction that caused the page fault

Steps in Handling a Page Fault (Cont.)

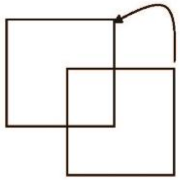


Aspects of Demand Paging

- 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** (never bring in a page until it's needed)
- Actually, a given instruction could access multiple pages -> multiple page faults
  - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
  - Though this behavior is unlikely because of **locality of reference**
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart

Instruction Restart Issues

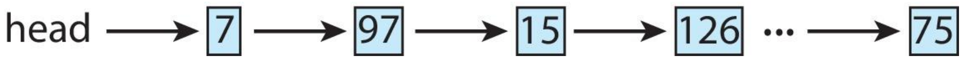
- Consider an instruction that could access several different locations
  - Block move (chunk of data moved between memory blocks)



    Auto increment/decrement location
  - Restart the whole operation?
    - What if source and destination overlap? Source block may have been modified – can't be restarted
  - Solution
    - 1) check both ends first – trigging page fault beforehand but no more faults during actual moving
    - 2) use temporary register to hold the values of modified memory – restore them if page fault occurs

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.

Performance of Demand Paging

- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed (1~100 microseconds)
  - Read the page – lots of time
  - Restart the process – again just a small amount of time
- Page Fault Rate $0 \le p \le 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

  EAT = (1 – p) x memory access time + p (page fault overhead time)

Demand Paging 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 (=> proportional to page fault rate)
- If one access out of 1,000 causes a page fault, then

  EAT = 8,200 ns = 8.2 microseconds.

  This is a slowdown by a factor of 40!! (8200/200 = 41)
- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x p
  - p < .0000025 = 1/400,000
  - < one page fault in every 400,000 memory accesses

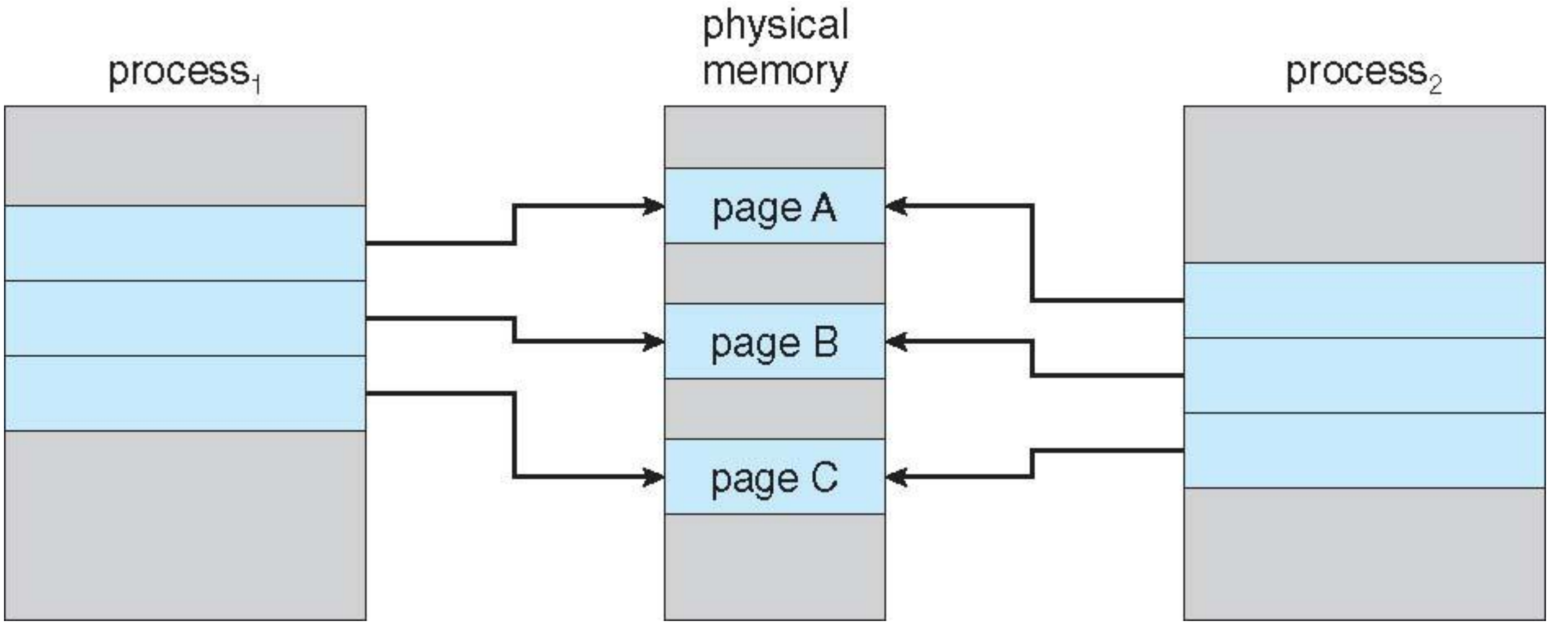
Demand Paging Optimizations

- Swap space I/O faster than file system I/O even if on the same device
  - Swap allocated in larger chunks; less management needed than file system
- Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used in older BSD Unix
- Demand paging from the file system initially (only needed pages are read) but
  - Write to the swap space as they are replaced, and read back from there
  - Used by Linux and Windows
- Demand page in from program binary on disk, but discard rather than paging out when freeing frame (good for code)
  - Used in Linux and BSD Unix
  - Still need to write to swap space for -
    - Pages not associated with a file (like stack and heap) – **anonymous memory**
    - Pages modified in memory but not yet written back to the file system
- Mobile systems
  - Typically don't support swapping
  - Instead, demand page from file system and reclaim read-only pages (such as code)
  - Under iOS, anonymous memory pages are never reclaimed (released when done)

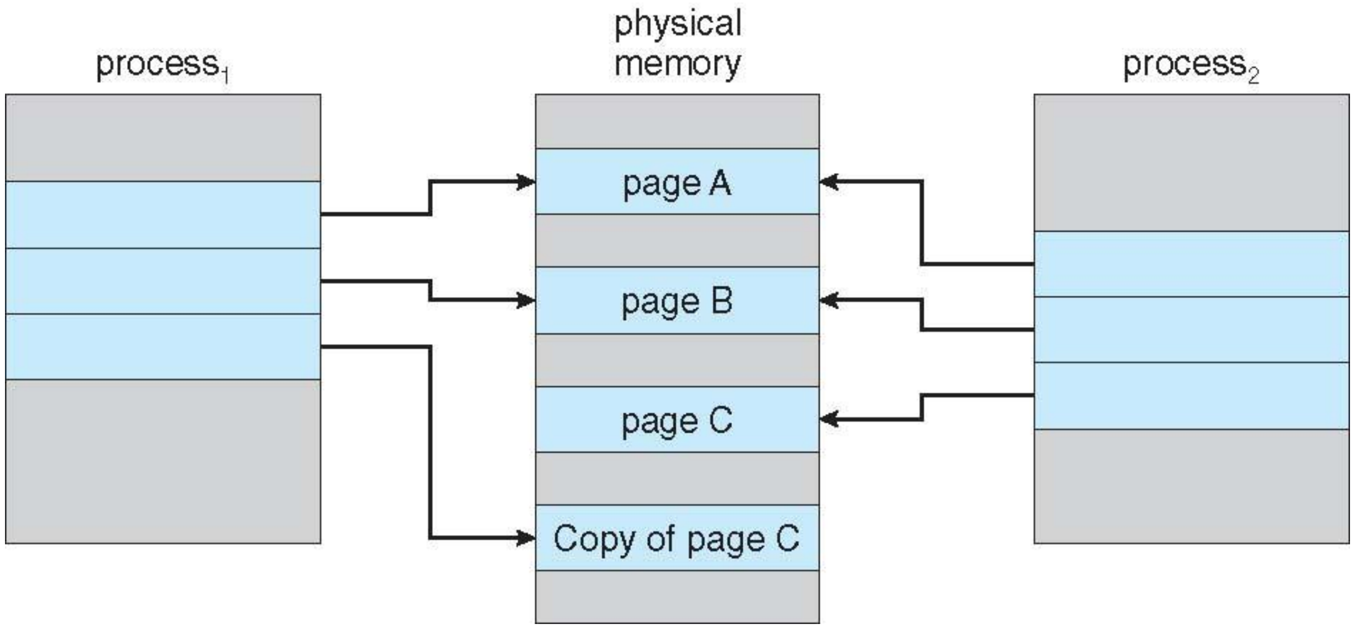
Copy-on-Write

- **Copy-on-Write** (COW) allows both parent and child processes to initially ***share*** the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
  - Pool should always have free frames for fast demand page execution
    - Don't want to have to free a frame as well as other processing on page fault
  - Why zero-out a page before allocating it?
- `vfork()` variation on `fork()` system call has parent suspend and child using address space of parent without modifying it
  - Designed to have child immediately call `exec()`
  - Very fast as no copy is done

Before Process 1 Modifies Page C



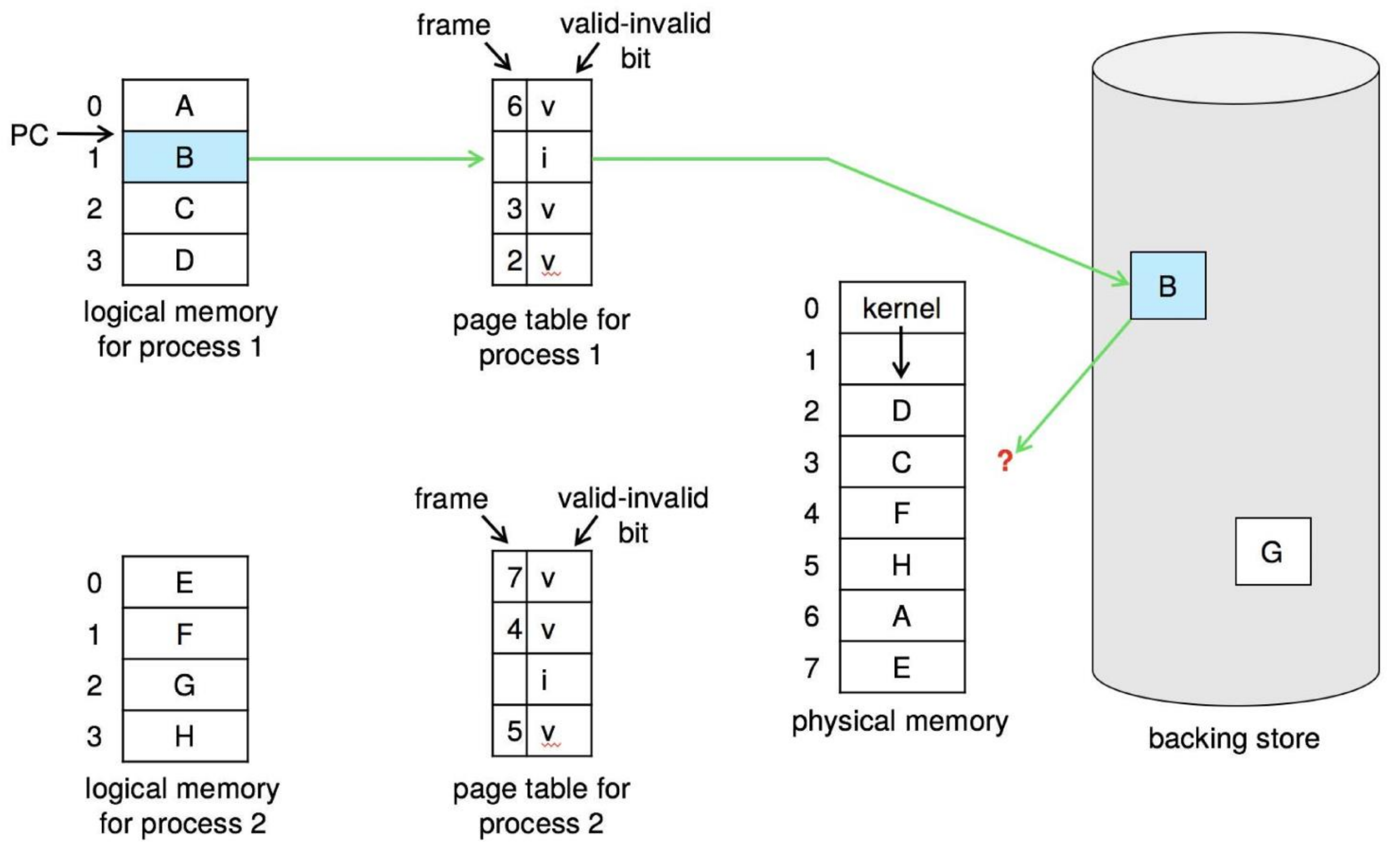
After Process 1 Modifies Page C



What Happens if There is no Free Frame?

- Free frames can be used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each? (fixed % or free competing)
- Page replacement – find some page in memory, but not really in use, page it out
  - Algorithm – terminate the process? 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

Need For Page Replacement



Page Replacement

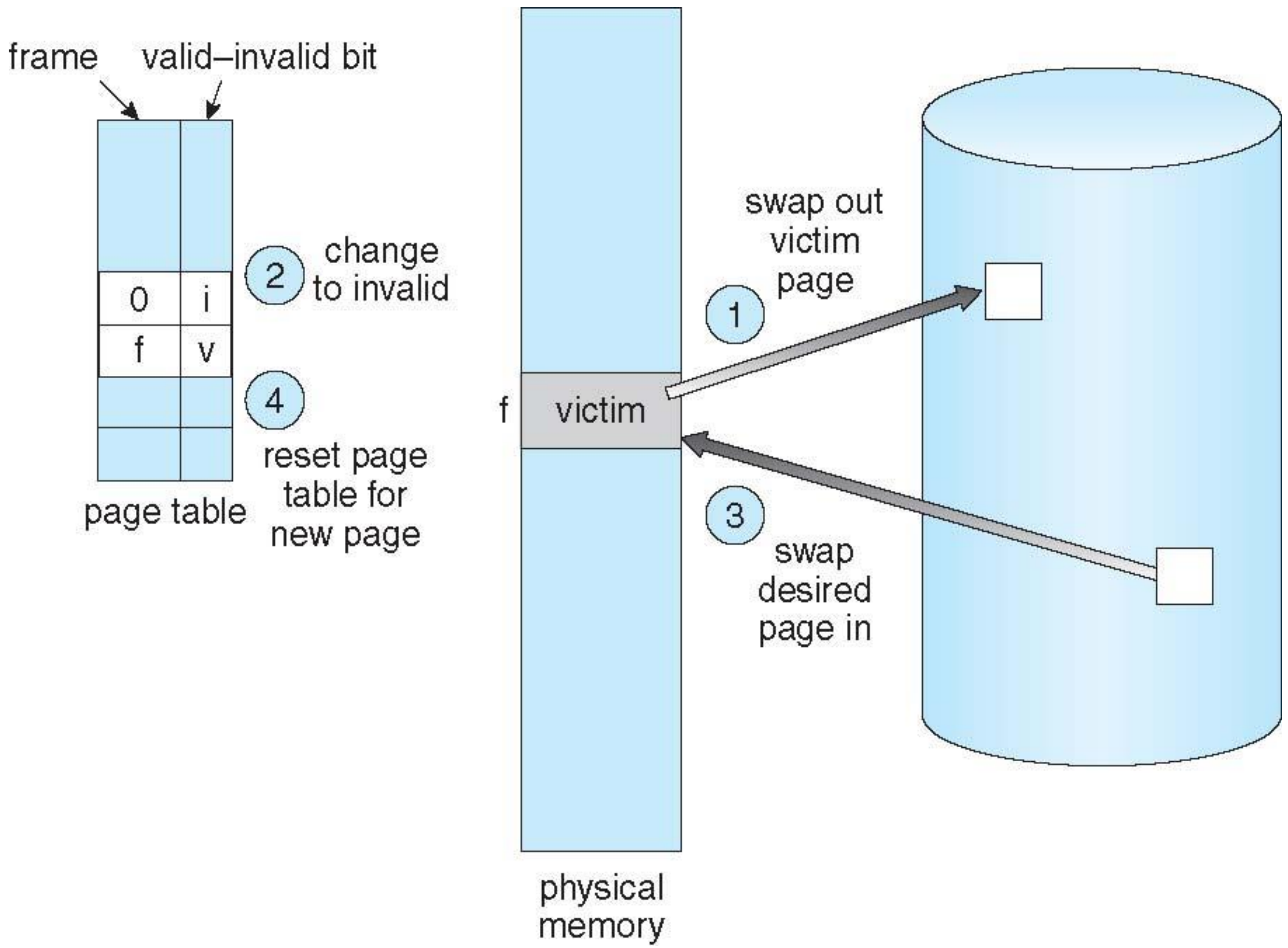
- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement
- Use **modify** (**dirty**) **bit** to reduce overhead of page transfers – only modified pages are written to disk
  - In other word, an unmodified page doesn’t need to be paged out – it can be discarded
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

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 now potentially 2-page transfers for page fault – increasing EAT

Page Replacement

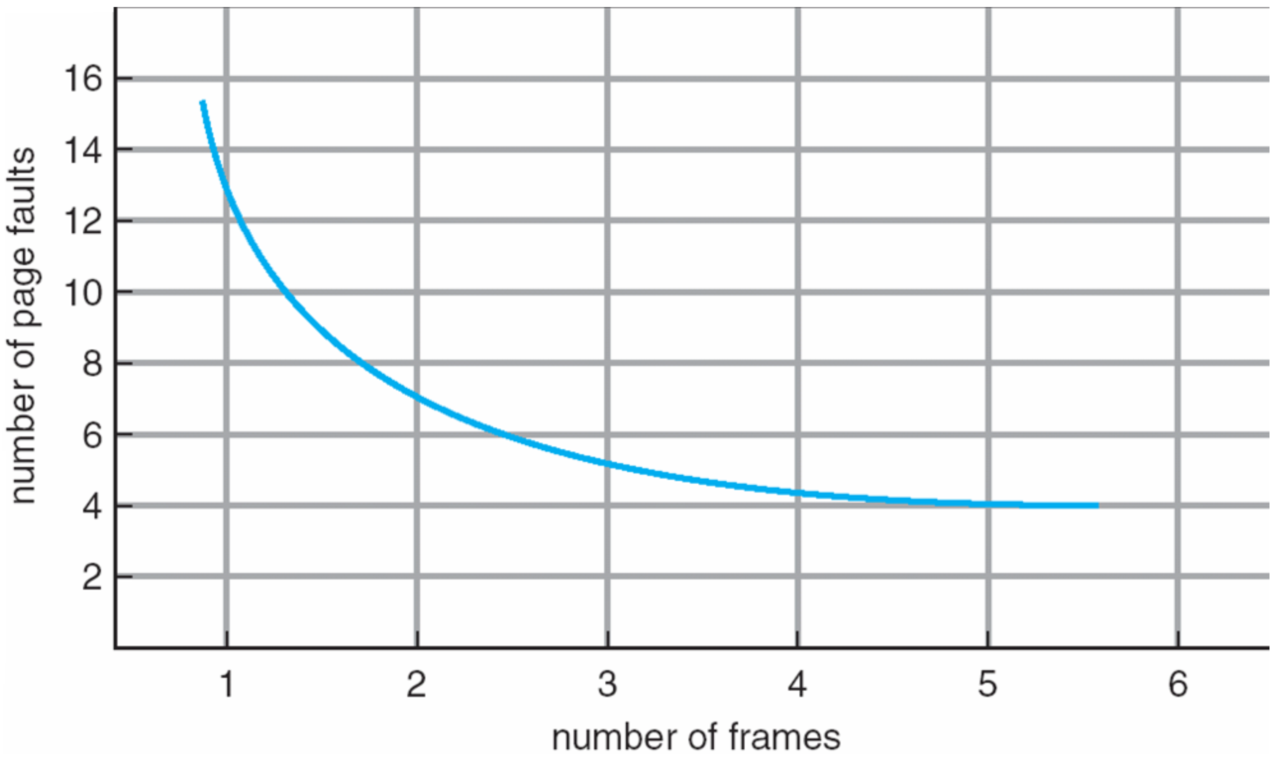


Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
  - How many frames to give each process
- **Page-replacement algorithm**
  - Select which frames to replace
  - 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

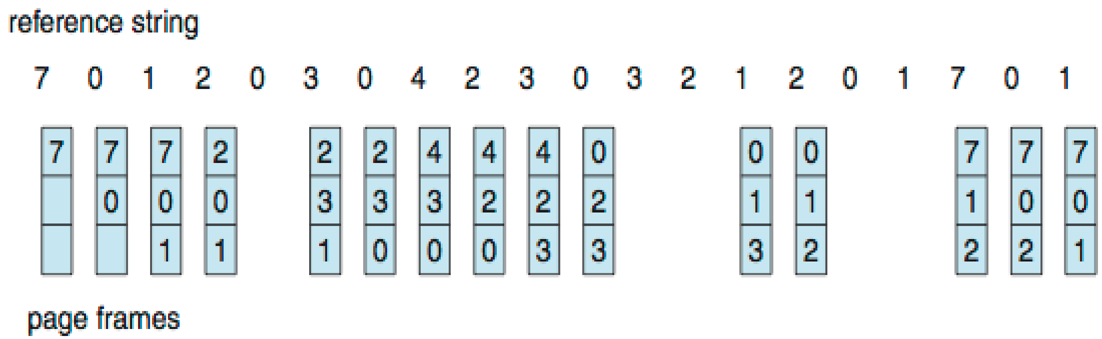
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

Graph of Page Faults Versus the Number of Frames



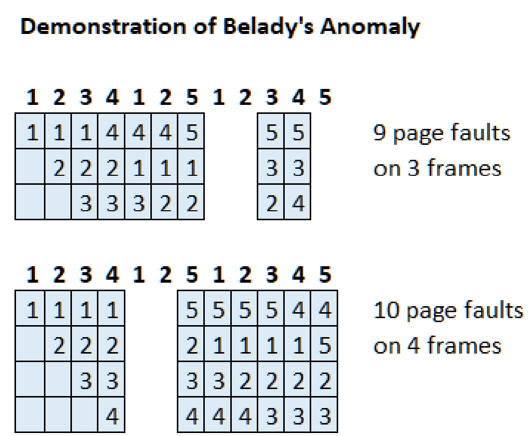
First-In-First-Out (FIFO) Algorithm

- 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)



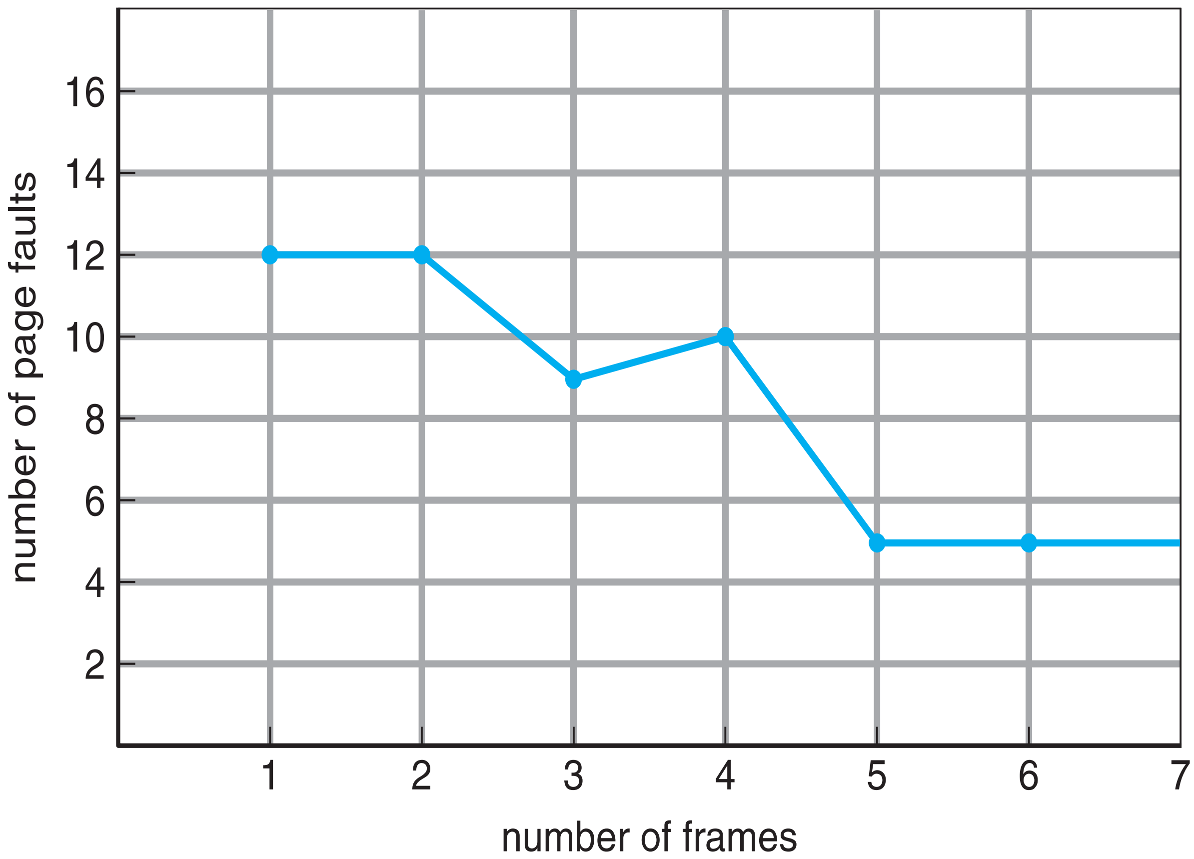
15 page faults

- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!
    - **Belady's Anomaly**



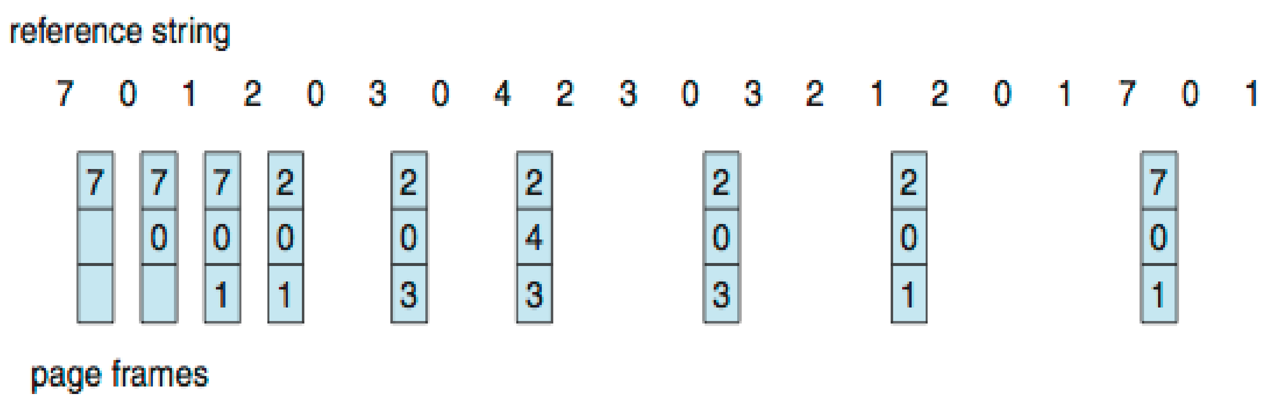
- How to track ages of pages?
  - Just use a FIFO queue

FIFO Illustrating Belady's Anomaly



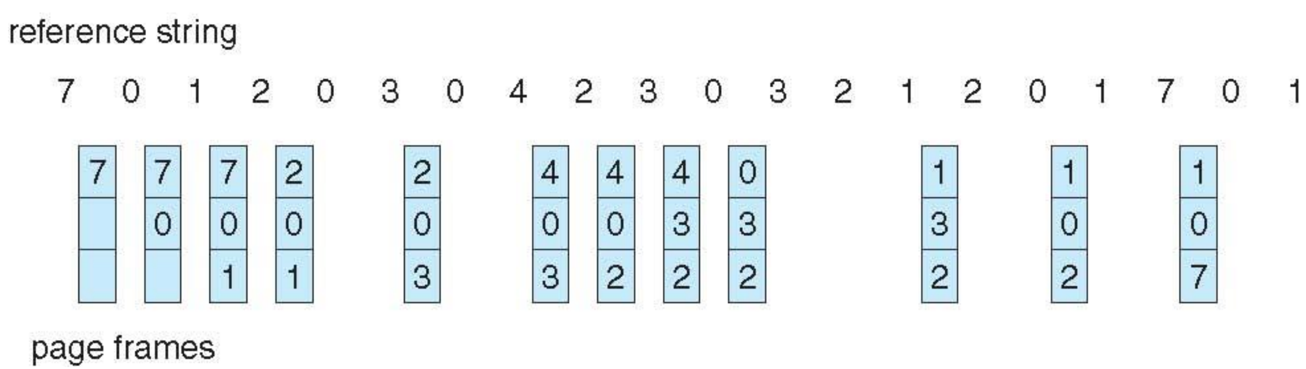
Optimal Algorithm

- Replace page that will not be used for longest period of time
  - 9 is optimal for the example
- How do you know this?
  - Can't read the future
- Used as a measuring stick to judge how well your algorithm performs



Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page



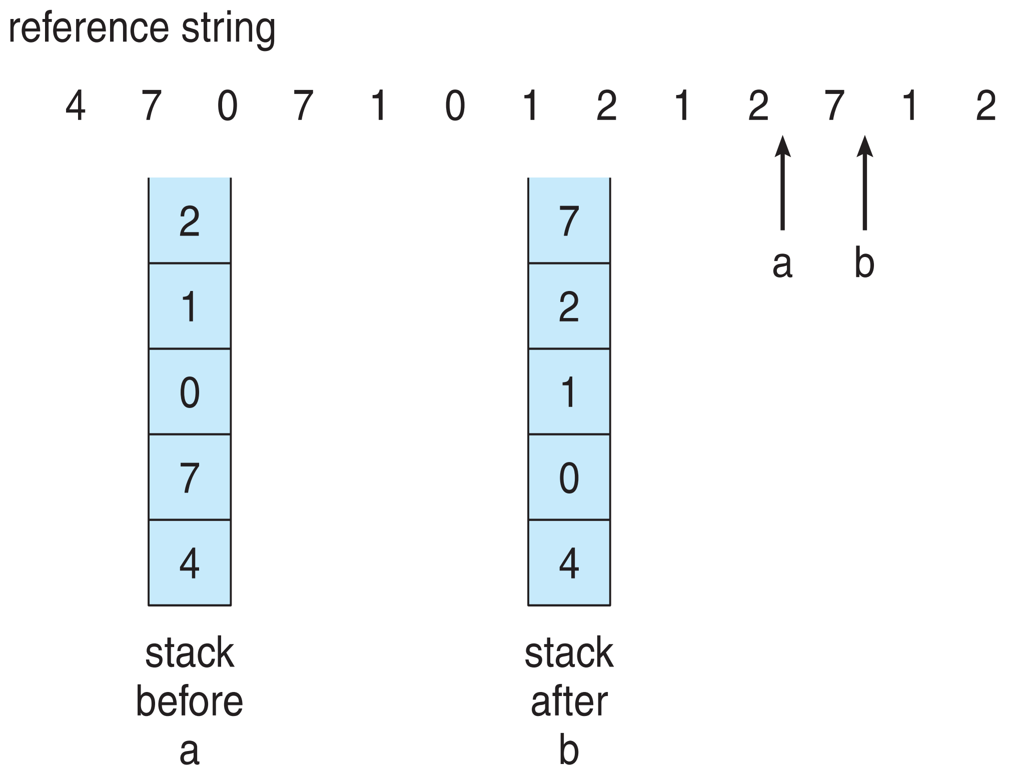
- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?

LRU Algorithm (Cont.)

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock (logical clock) into the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - Search through table needed
- Stack implementation (note this is not the data structure stack)
  - Keep a stack of page numbers in a double link form
    - Top is the most recently used page
    - Tail (bottom) is the least recently used page
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed in worst case
  - But each update more expensive
  - No search for replacement

LRU Algorithm (Cont.)

- 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



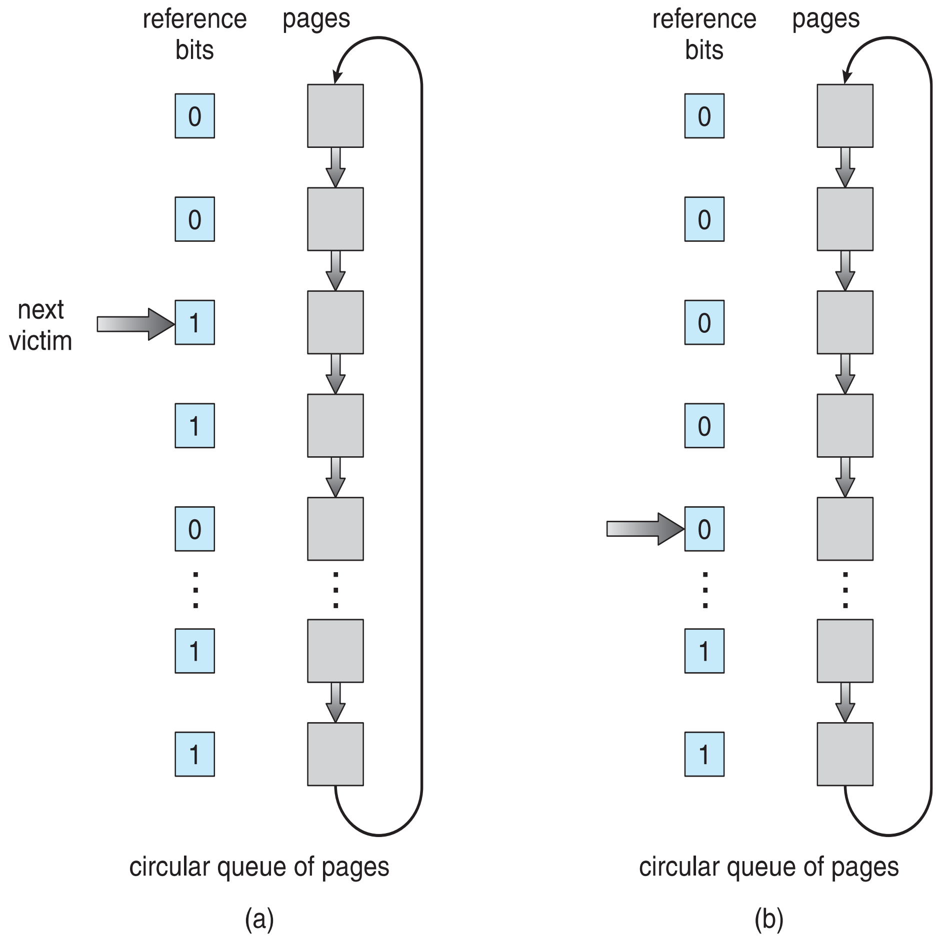
LRU Approximation Algorithms

- LRU needs special hardware (clock fields or stack) and still slow
- Few systems could tolerate this level of overhead
- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced, bit set to 1
    - We do not know the order, however
- **Additional-Reference-Bits Algorithm**
  - Keep an 8-bit byte for each page in a table in memory
  - A timer interrupt regularly (say every 100ms) allows the OS to right-shift the reference bits for each page
  - So a page has one use followed by no use will have the bit pattern of (10000000, then 01000000)
  - If the 8 bits are interpreted as an integer, then the lowest number is the LRU page and can be replaced

LRU Approximation Algorithms (cont.)

- **Second-chance algorithm**
  - Generally FIFO, plus hardware-provided reference bit
  - Main idea is to give a page with reference bit of 1 a second chance – clearing the bit but not replacing it
  - **Clock** replacement
    - Note here clock is merely the semblance to a clock hand of a pointer going through a circular queue (see next slide)
  - 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

Second-chance Algorithm



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, similar to the clock scheme but use the four classes to replace first page in lowest non-empty class
  - Might need to search circular queue several times

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
  - Not common
- **Lease Frequently Used** (**LFU**) **Algorithm**:
  - Replaces page with smallest count
  - But it’s possible a page used heavily in the beginning may never be used again -> use some decaying
- **Most Frequently Used** (**MFU**) **Algorithm**:
  - Based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Page-Buffering Algorithms

- Keep a pool of free frames, always
  - Read page into free frame and select victim to evict but don’t evict it (write it out) immediately, this speeds up the restart process
  - When convenient, evict victim and add it to the free-frame pool
- Possibly, keep list of modified pages
  - When backing store is otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected

Allocation of Frames

- Each process needs ***minimum*** number of frames
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
- ***Maximum*** of course is total frames in the system
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- 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 $p_i = \frac{s_i}{S} \times m$

$$m = 64$$
$$s_1 = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 62 \approx 4$$
$$a_2 = \frac{127}{137} \times 62 \approx 57$$

2 left-over frames for free frame buffer pool

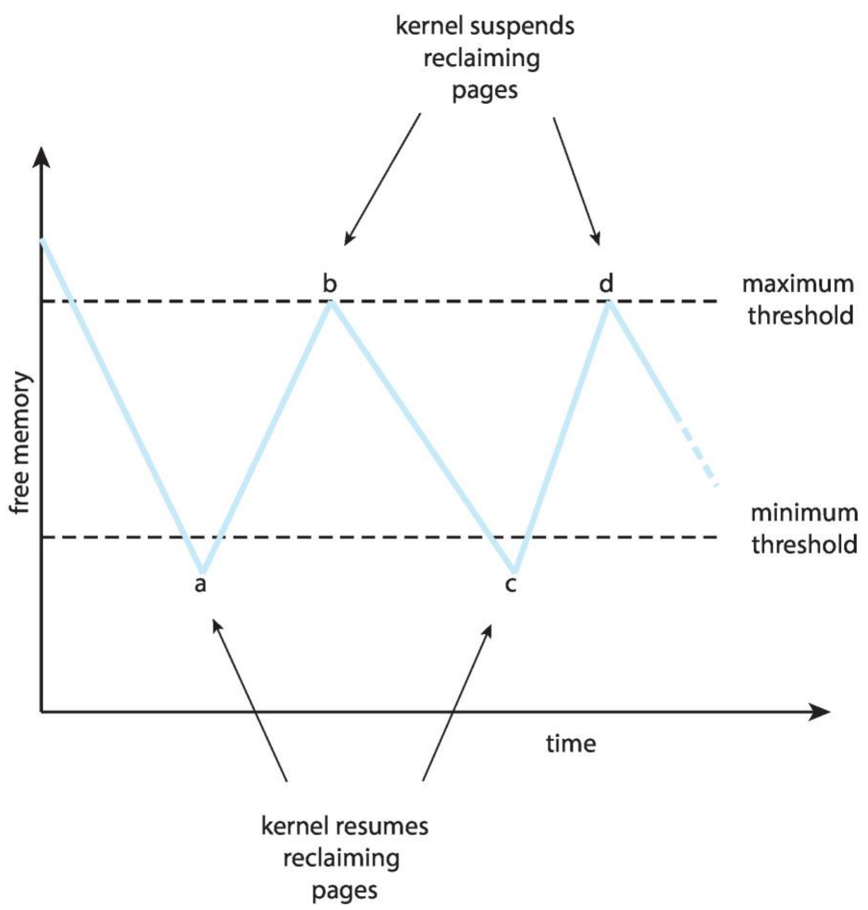
Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another (e.g., a high-priority process takes frames from lower-priority ones)
  - But then process execution time can vary greatly
  - But greater throughput so more common
- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance, as paging behavior is local to each process
  - But possibly underutilized memory

Reclaiming Pages

- A strategy to implement global page-replacement policy
- All memory requests are satisfied from the free-frame list
- Rather than waiting for the list to drop to zero before we begin selecting pages for replacement, Page replacement is triggered when the list falls below a certain threshold.
- This strategy attempts to ensure there is always sufficient free memory to satisfy new requests.

Reclaiming Pages Example

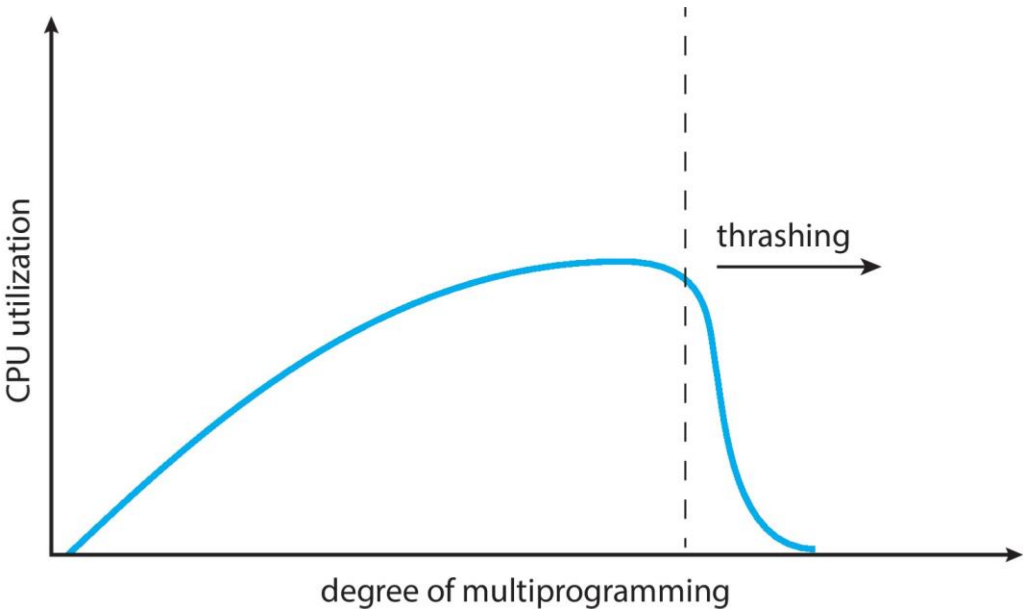


Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame,
  - Which is quickly needed again – causing more page faults
  - This leads to:
    - Low CPU utilization (more paging than executing, and ready queue becomes empty)
    - Operating system thinking that it needs to increase the degree of multiprogramming because CPU is not busy
    - Another process added to the system and exaggerates the situation even more

Thrashing (Cont.)

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



Demand Paging and Thrashing

- Why does demand paging work?

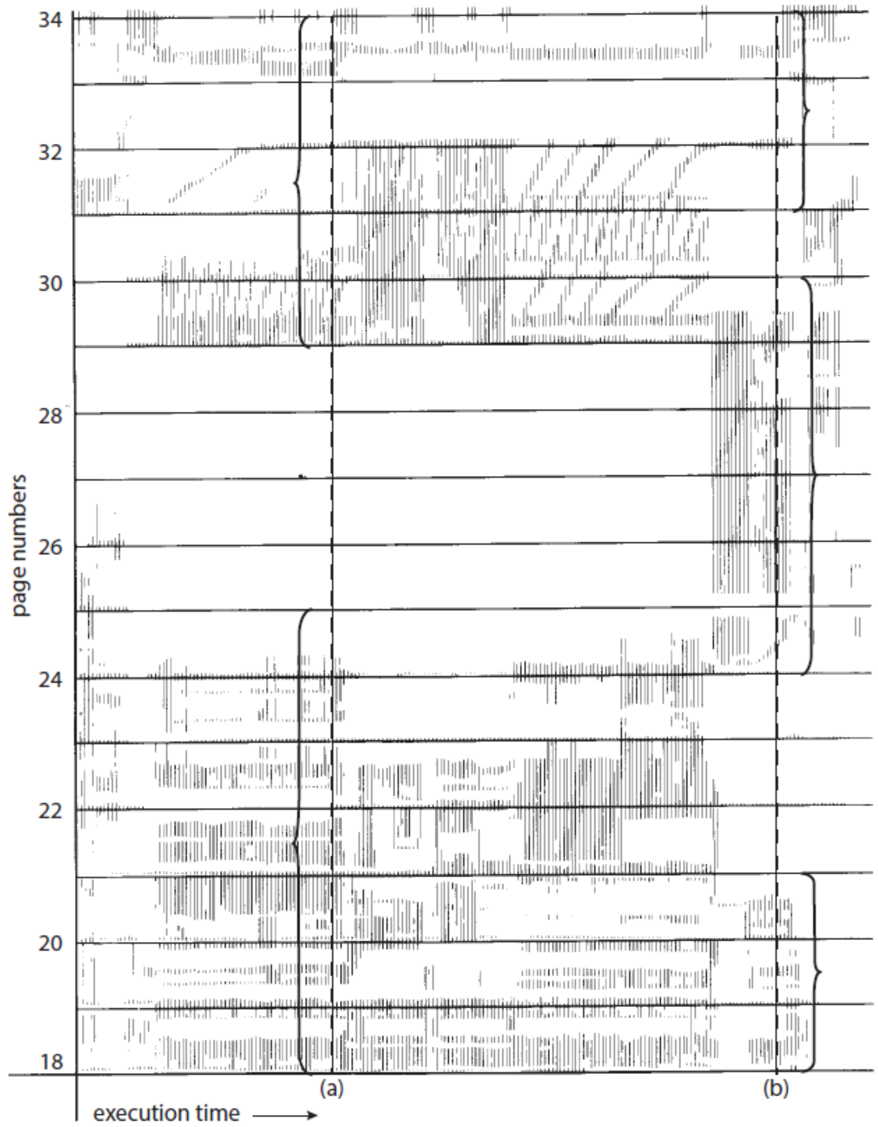
  Locality model

  - A locality is a set of pages actively used together
  - Process migrates from one locality to another (like calling a function)
  - Localities may overlap (global variables)
- Why does thrashing occur?

  Σ size of locality > total memory size

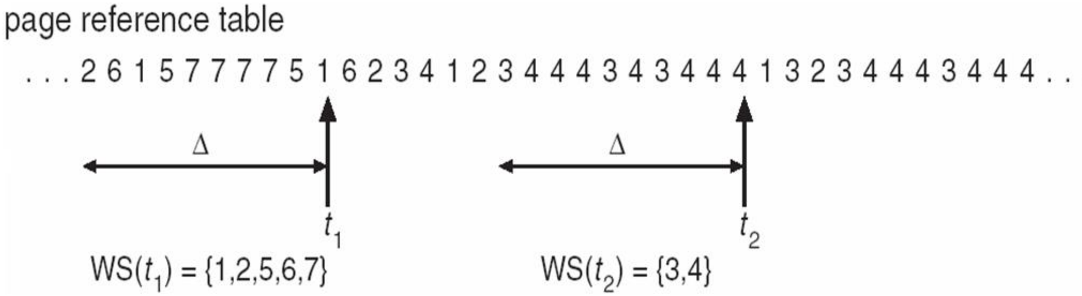
- Limit effects by using local page replacement
  - Each process selects from own set of allocated frames
  - One process starting thrashing can't steal frames from another process and cause the latter to thrash as well.

Locality In A Memory-Reference Pattern



Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10 references



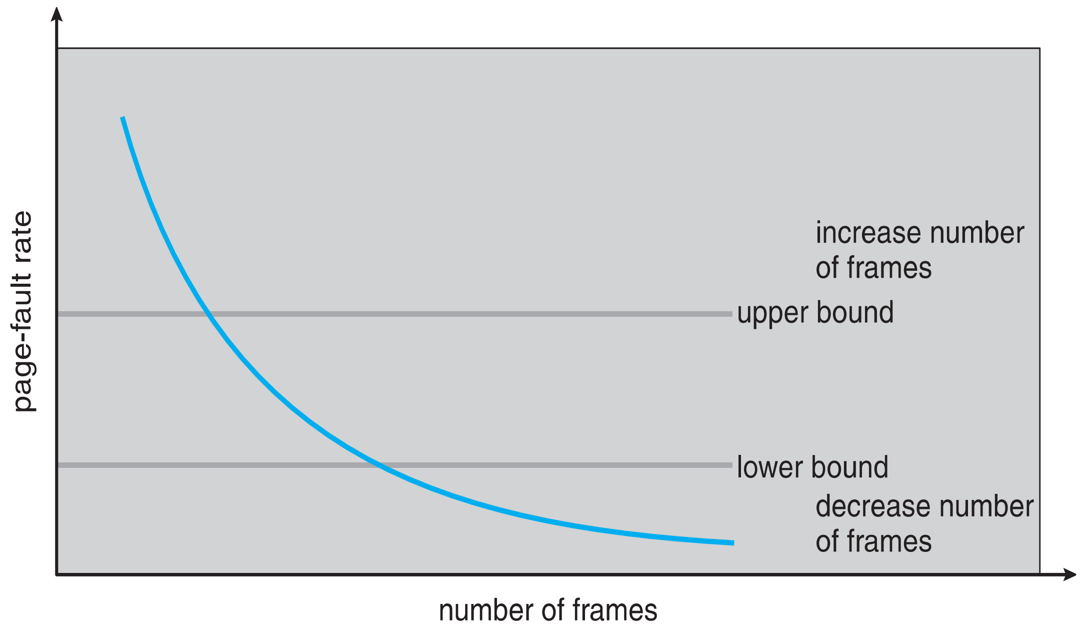
- WSS_i (Working Set Size of Process P_i) = total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \Sigma\, WSS_i \equiv$ total demand frames
- if $D > m$ *(total # of available frames)* $\Rightarrow$ Thrashing happens
- Policy: if $D > m$, then suspend or swap out one of the processes

Keeping Track of the Working Set

- Working-set window is a moving window (new reference appears at one end, and the oldest drops off the other end)
- Approximate with interval timer + a reference bit
- Example: Δ = 10,000 (time units, approximated as # of references)
  - Timer interrupts after every 5000 time units (references)
  - Keep 2 in-memory bits for each page
  - Whenever a timer interrupts, for each page copy the value of the reference bit to one of the in-memory bits and reset the reference bit to 0
    - In between the timer interrupts, a page's reference bit may be set
    - But in two interrupts, pages with at least one in-memory bit set is considered to be in the working set
- Why is this not completely accurate?
  - Uncertain exactly when the reference takes place
- Improvement = 10 bits and interrupt every 1000 time units, but the cost will be much higher

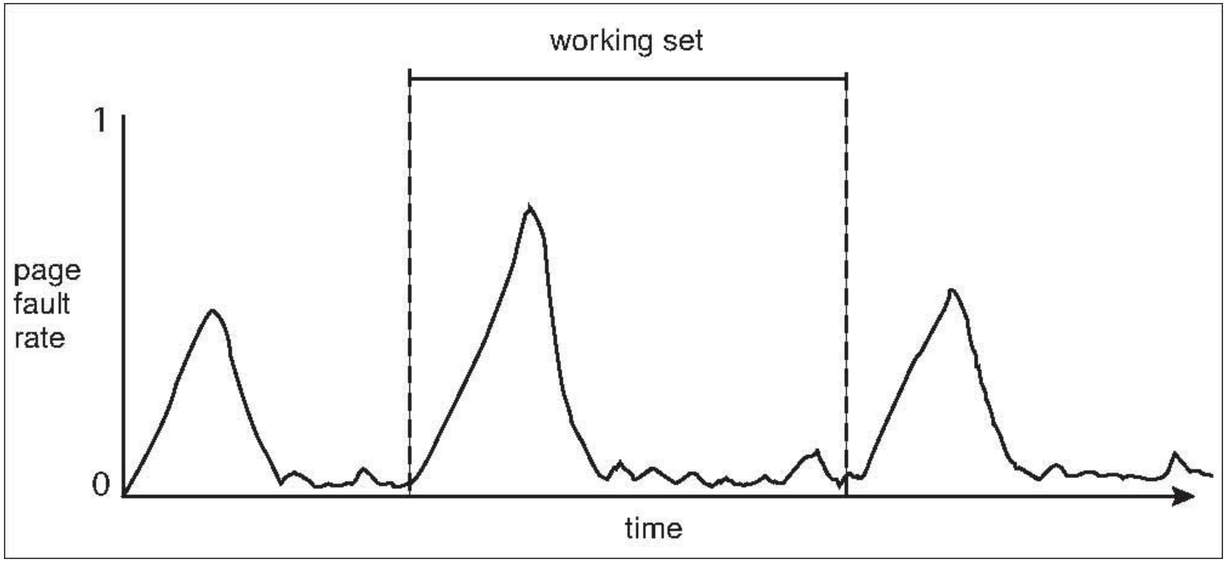
Page-Fault Frequency

- More direct approach than WSS in preventing thrashing
- Establish "acceptable" **page-fault frequency** (**PFF**) rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



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



Other Considerations

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

Prepaging

- To reduce the large number of page faults that occurs at process startup
- Prepage all or some of the pages a process will need, before they are referenced
  - e.g., prepaging a working-set of a process suspended due to lack of free frames
- But if prepaged pages are unused, I/O and memory were wasted
  - The question is whether the cost is less than that of page faults
- Assume s pages are prepaged and a fraction α of the pages is used
  - Is cost of $s * \alpha$ saved pages faults > or < than the cost of prepaging $s * (1- \alpha)$ unnecessary pages?
  - α near zero $\Rightarrow$ prepaging loses
  - α near one $\Rightarrow$ prepaging wins
  - Prepaging a data file is more predictable than prepaging an executable program

Page Size

- Sometimes OS designers have a choice
  - Especially if running on custom-built CPU
- Page size selection must take into consideration:
  - Fragmentation (internal, small pages favored)
  - Page table size (large pages favored -> smaller page table)
  - **Resolution** (better isolate needed memory, favors small pages)
  - I/O overhead (favors large pages, seek/latency time dominates)
  - Number of page faults (favors large pages)
  - Locality (favors small pages which match locality more accurately)
  - TLB size and effectiveness (can make small pages tolerable)
- Always power of 2, usually in the range 2^{12} (4,096 bytes) to 2^{22} (4,194,304 bytes)
- On average, growing over time

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 (internal) 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
  - Then OS may have to manage the TLB instead of the hardware (TLB entry may need field indicating page size), increasing cost
  - Used when increased hit ratio and TLB reach offset the OS cost

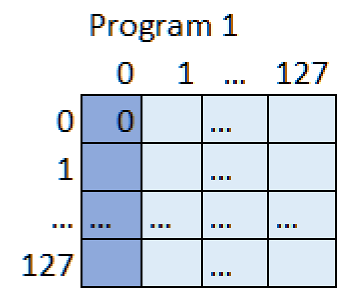
Program Structure

- Programmers can help improve system performance
- Example to initialize to 0 each element of a 128x128 array
  - `int[128][128] data; // stored as row major`
  - Each row is stored in one page (page size holds 128 integers)
  - Program 1

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

If the OS allocates fewer than 128 frames to entire program

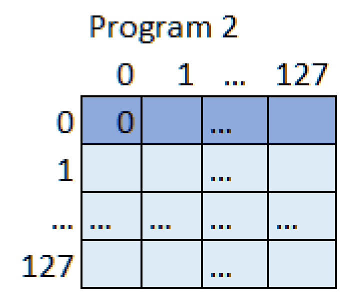
There will be 128 x 128 = 16,384 page faults



  - Program 2

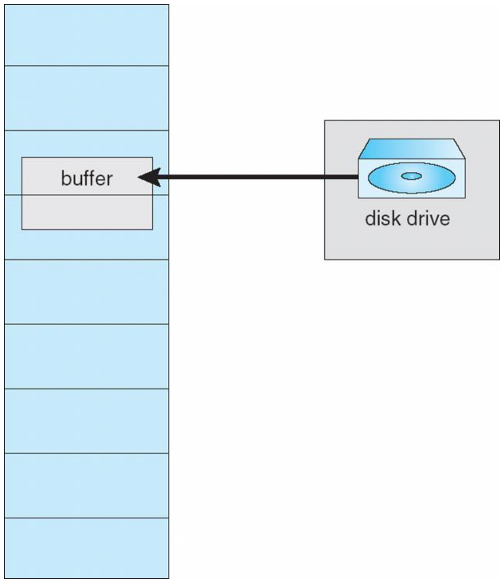
```
for (i = 0; i < 128; i++)
     for (j = 0; j < 128; j++)
          data[i][j] = 0;
```

Only 128 page faults



I/O interlock

- **I/O Interlock** – Pages must sometimes be locked into memory
  - Can't allow I/O processor to access memory already evicted
- **Pinning** of pages to lock into memory
  - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm
  - A lock bit is associated with each frame

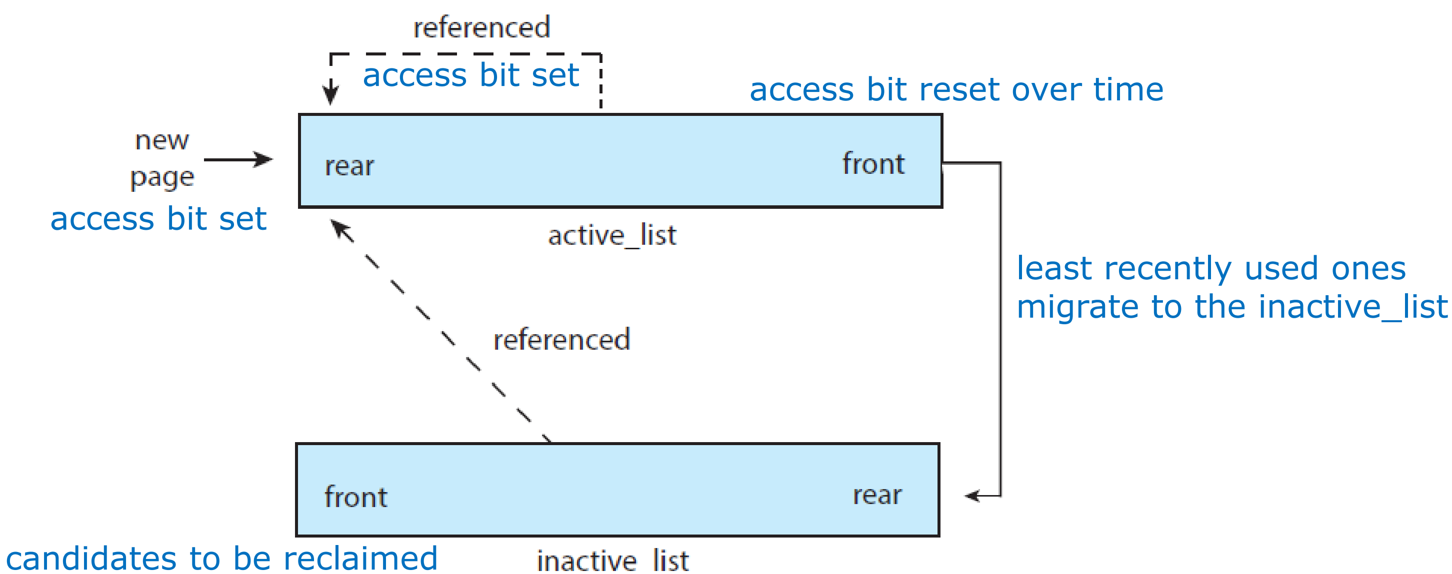


Operating System Examples

- Linux
- Windows
- Solaris

Linux

- Uses demand paging with a global page-replacement policy
  - Similar to the LRU-approximation clock algorithm
- Maintains two types of page lists: `active_list` (in use) and `inactive_list` (not recently referenced, eligible to be reclaimed)
- The two lists are kept in relative balance



Windows

- Uses demand paging with **clustering**. Clustering brings in pages surrounding the faulting page
- Processes are assigned **working set minimum** and **working set maximum**
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory
- A process may be assigned as many pages as possible up to its working set maximum
- When the amount of free memory in the system falls below a threshold**, automatic working set trimming** is performed to restore the amount of free memory
- Working set trimming removes pages from processes that have pages in excess of their working set minimum

End of Chapter 10