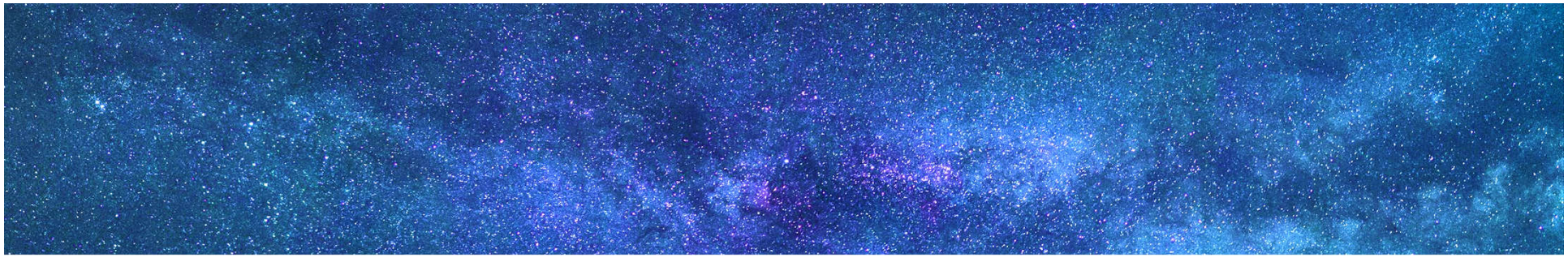
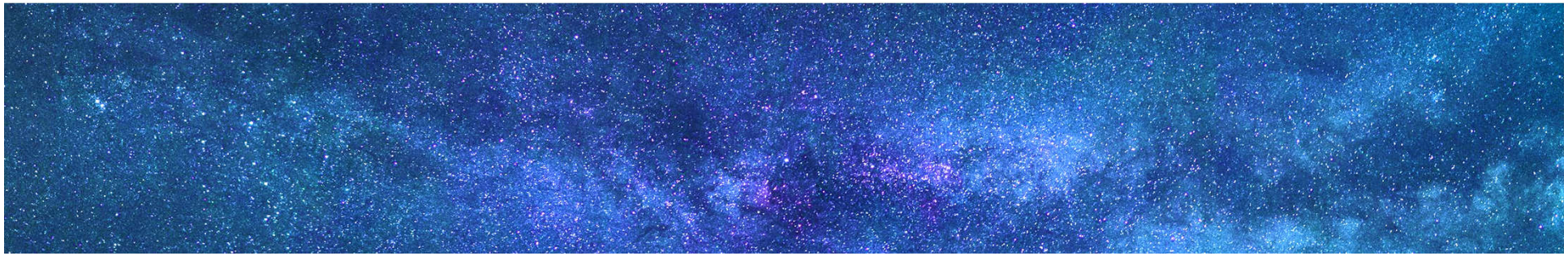


Operating System

Chapter 9: Virtual memory



Objective

- 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 principles of the working-set model.
- To examine the relationship between shared memory and memory-mapped files.

Background

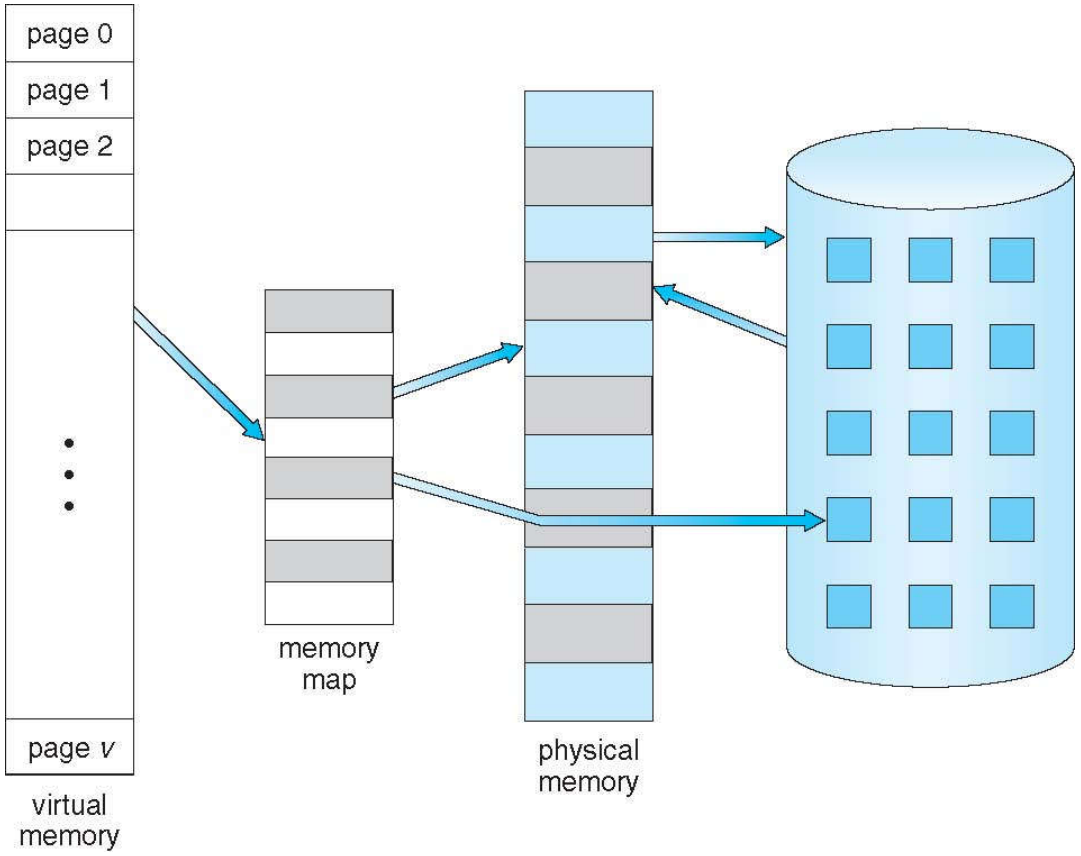
- Code needs to be in memory to execute, but entire program rarely needed or used at the same time
  - error handling code, unusual routines, large data structures
- Consider ability to execute partially-loaded program
  - program no longer constrained by limits of physical memory
  - programs could be larger than physical memory

Background

- Virtual memory: separation of logical memory from physical memory
  - ➢ only part of the program needs to be in memory for execution
    - – logical address space can be much larger than physical address space
    - – more programs can run concurrently
    - – less I/O needed to load or swap processes (part of it)
  - ➢ allows memory (e.g., shared library) to be shared by several processes: better IPC performance
  - ➢ allows for more efficient process forking (copy-on-write)
- Virtual memory can be implemented via:
  - ➢ demand paging

Virtual memory > physical memory

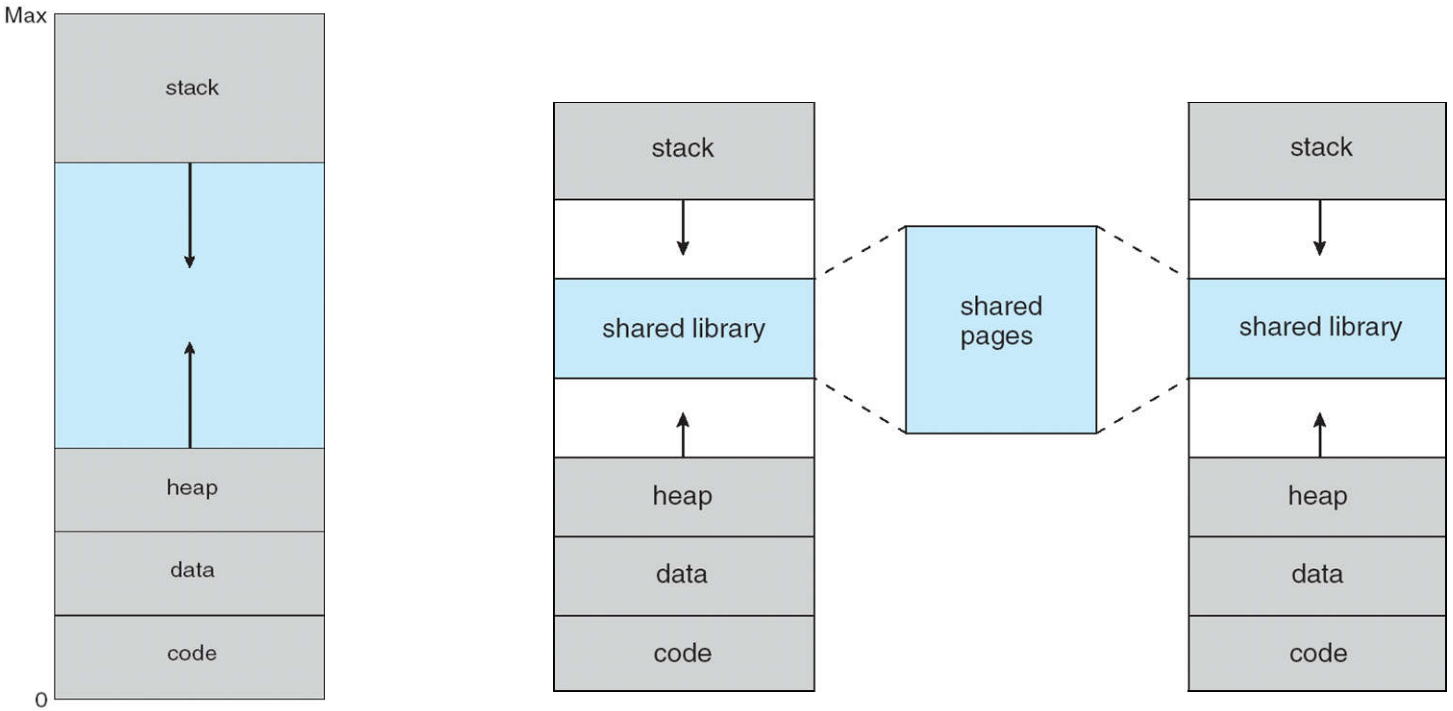
- **Virtual memory** involves the separation of logical memory as perceived by users from physical memory.



Virtual address space

- The virtual address space of a process refers to the logical (or virtual) view of how a process is stored in memory.
- Code, date, heap, stack
  - Possible of Sparse addresses

➢ **Shared library usage**

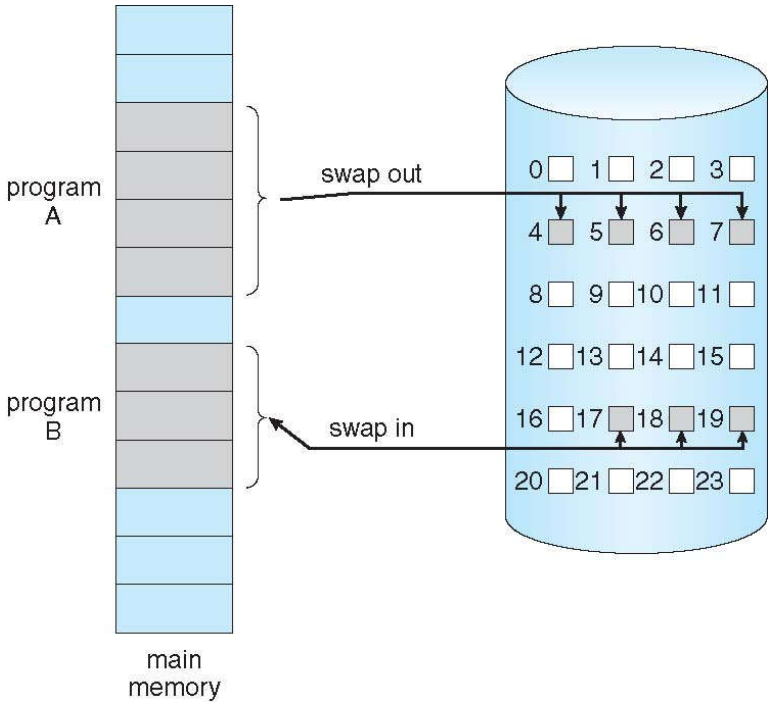


Demand paging

- **Why demand paging?**
  - Less memory needed
  - Faster response
  - Better CPU utilization

- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**

- **Pre-Paging**: pre-page all or some of pages a process will need, before they are referenced
  - it can reduce the number of page faults during execution



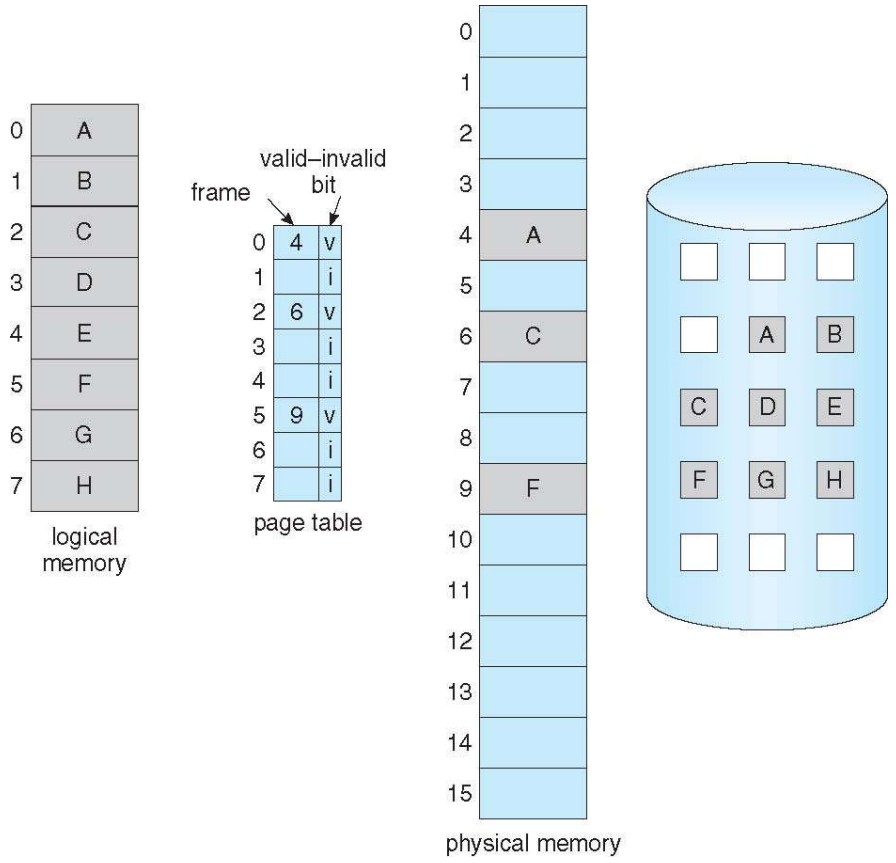
Demand paging implementation

- Help of valid-invalid bit

Each page table entry has a valid–invalid (present) bit

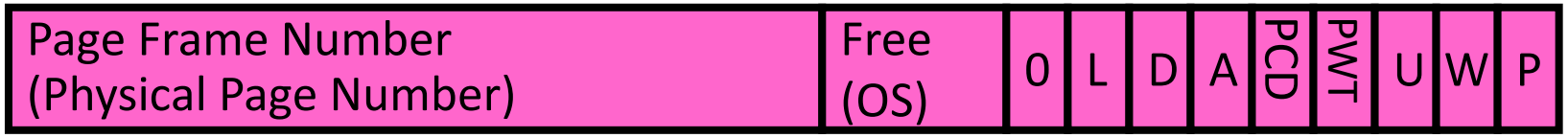
- V ➠ in memory (memory is resident)
- I ➠ not-in-memory
- initially, valid–invalid bit is set to i on all entries
- during address translation, if the entry is invalid, it will trigger a page fault

- Example of a page table snapshot



Recall: What is in a Page Table Entry

- What is in a Page Table Entry (or PTE)?
  - Pointer to next-level page table or to actual page
  - Permission bits: valid, read-only, read-write, write-only
- Example: Intel x86 architecture PTE:
  - Address same format previous slide (10, 10, 12-bit offset)
  - Intermediate page tables called “Directories”



P: Present (same as “valid” bit in other architectures)
W: Writeable
U: User accessible
PWT: Page write transparent: external cache write-through
PCD: Page cache disabled (page cannot be cached)
A: Accessed: page has been accessed recently
D: Dirty (PTE only): page has been modified recently
L: L=1⇒4MB page (directory only).
Bottom 22 bits of virtual address serve as offset

Demand Paging Mechanisms

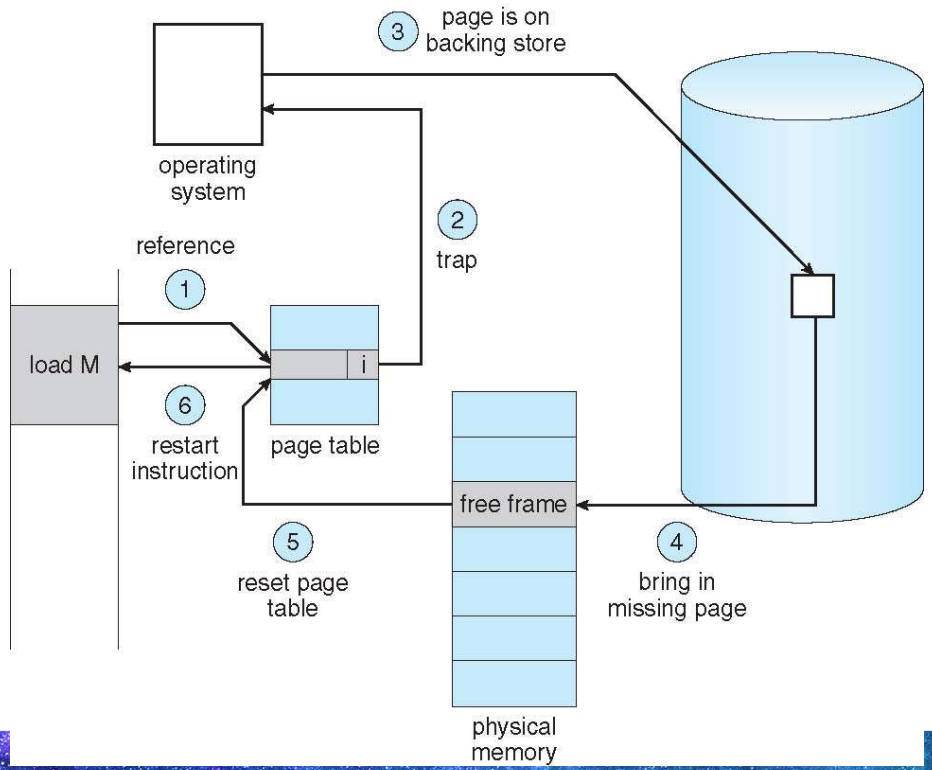
- PTE helps us implement demand paging
  - Valid ⇒ Page in memory, PTE points at physical page
  - Not Valid ⇒ Page not in memory; use info in PTE to find it on disk when necessary
- Suppose user references page with invalid PTE?
  - Memory Management Unit (MMU) traps to OS
    - Resulting trap is a "Page Fault"
  - What does OS do on a Page Fault?:
    - Choose an old page to replace
    - If old page modified ("D=1"), write contents back to disk
    - Change its PTE and any cached TLB to be invalid
    - Load new page into memory from disk
    - Update page table entry, invalidate TLB for new entry
    - Continue thread from original faulting location
  - TLB for new page will be loaded

Page fault, its handler

- If there is a reference to a page, first reference to that page will trap to operating system: **page fault**
  - ➢ Extreme case – start process with *no* pages in memory (**Pure demand paging**)
    - – OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault

What happens for an instruction with page fault?

1. Fetch and decode the instruction (ADD).
2. Fetch A.
3. Fetch B.
4. Add A and B.
5. Store the sum in C.



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.

Stages in demand paging

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:
   - 5.1 Wait in a queue for this device until the read request is serviced
   - 5.2 Wait for the device seek and/or latency time
   - 5.3 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

Performance of demand paging

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

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

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
- If one access out of 1,000 causes a page fault, then

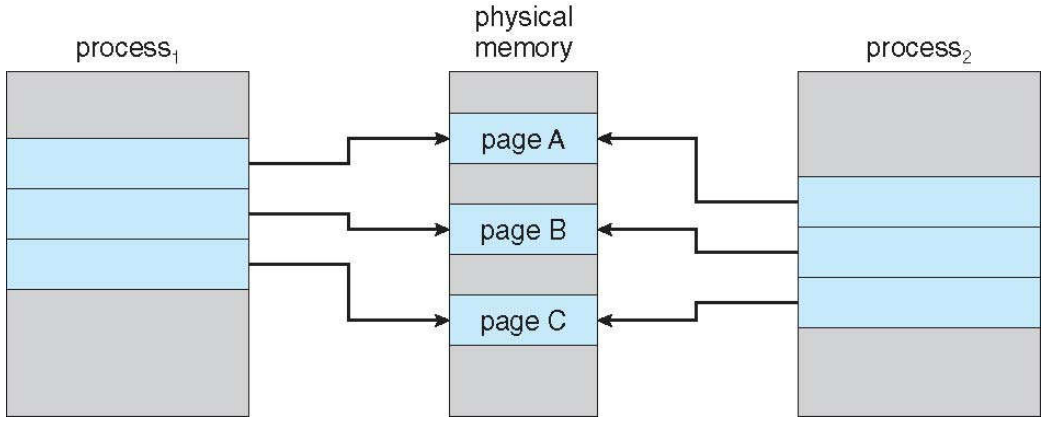
  EAT = 8.2 microseconds.

  This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p

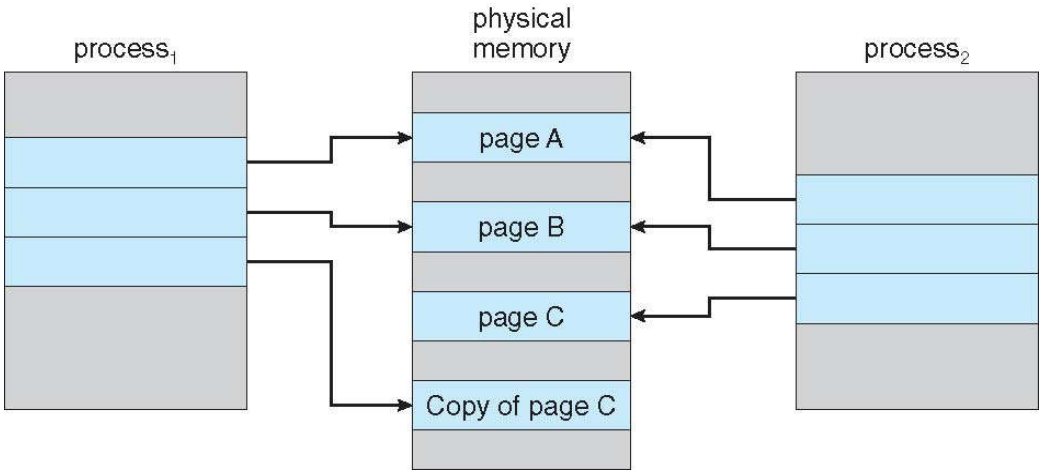
    20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 memory accesses

Copy-on-Write

- **Copy-on-Write** (COW) allows both parent and child processes to initially ***share*** the same pages in memory
- COW allows more efficient process creation
  - no need to copy the parent memory during fork
  - only changed memory will be copied later



Before Process 1 Modifies Page C



After Process 1 Modifies Page C

Page replacement

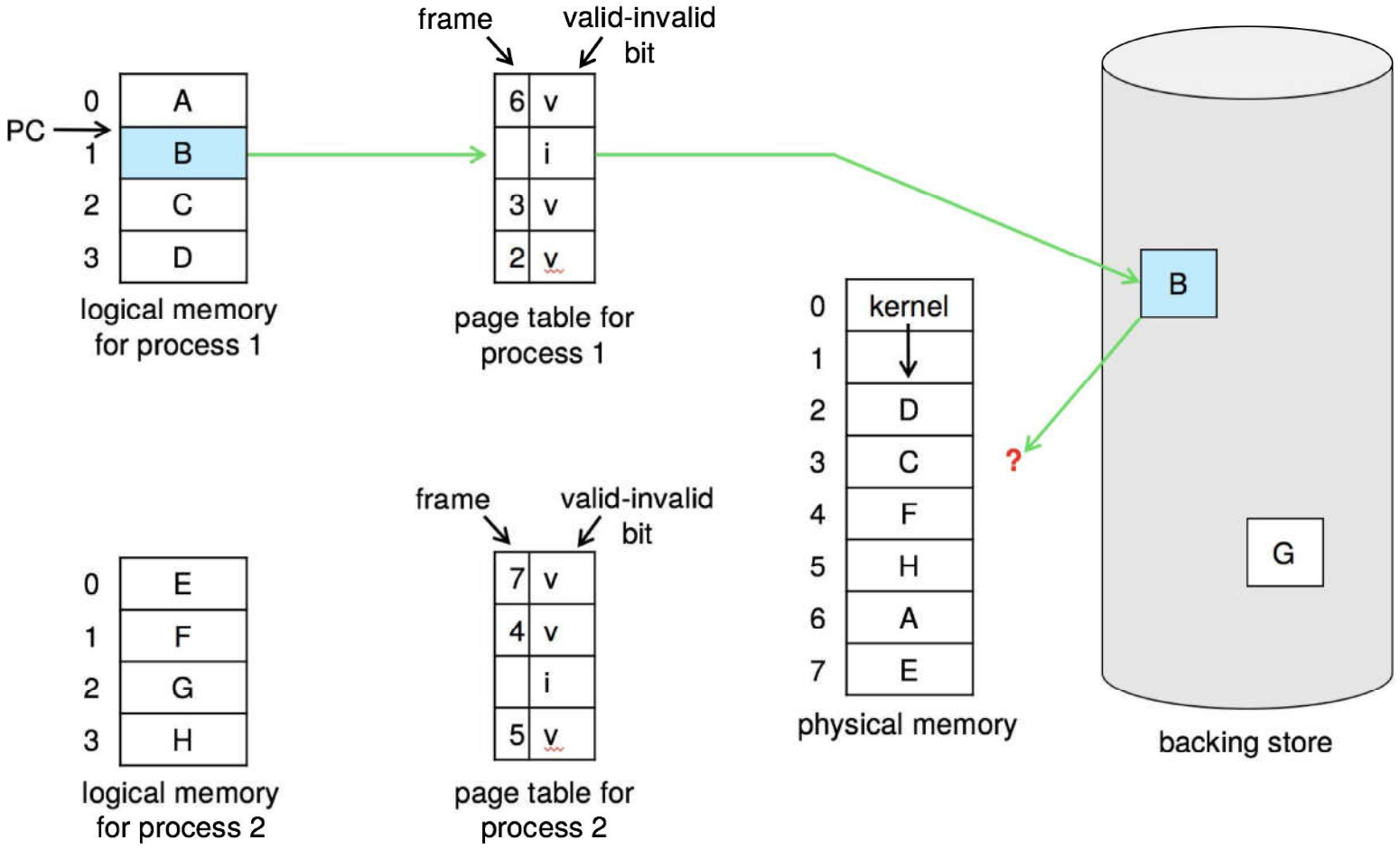
What Happens if There is no Free Frame?

- 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

- Memory is an important resource, system may run out of memory
  - ✓ To prevent out-of-memory, swap out some pages
  - ✓ page replacement usually is a part of the page fault handler
  - ✓ policies to select victim page require careful design
    - need to reduce overhead and avoid thrashing
  - ✓ use modified (dirty) bit to reduce number of pages to swap out
    - only modified pages are written to disk
  - ✓ select some processes to kill (last resort)
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

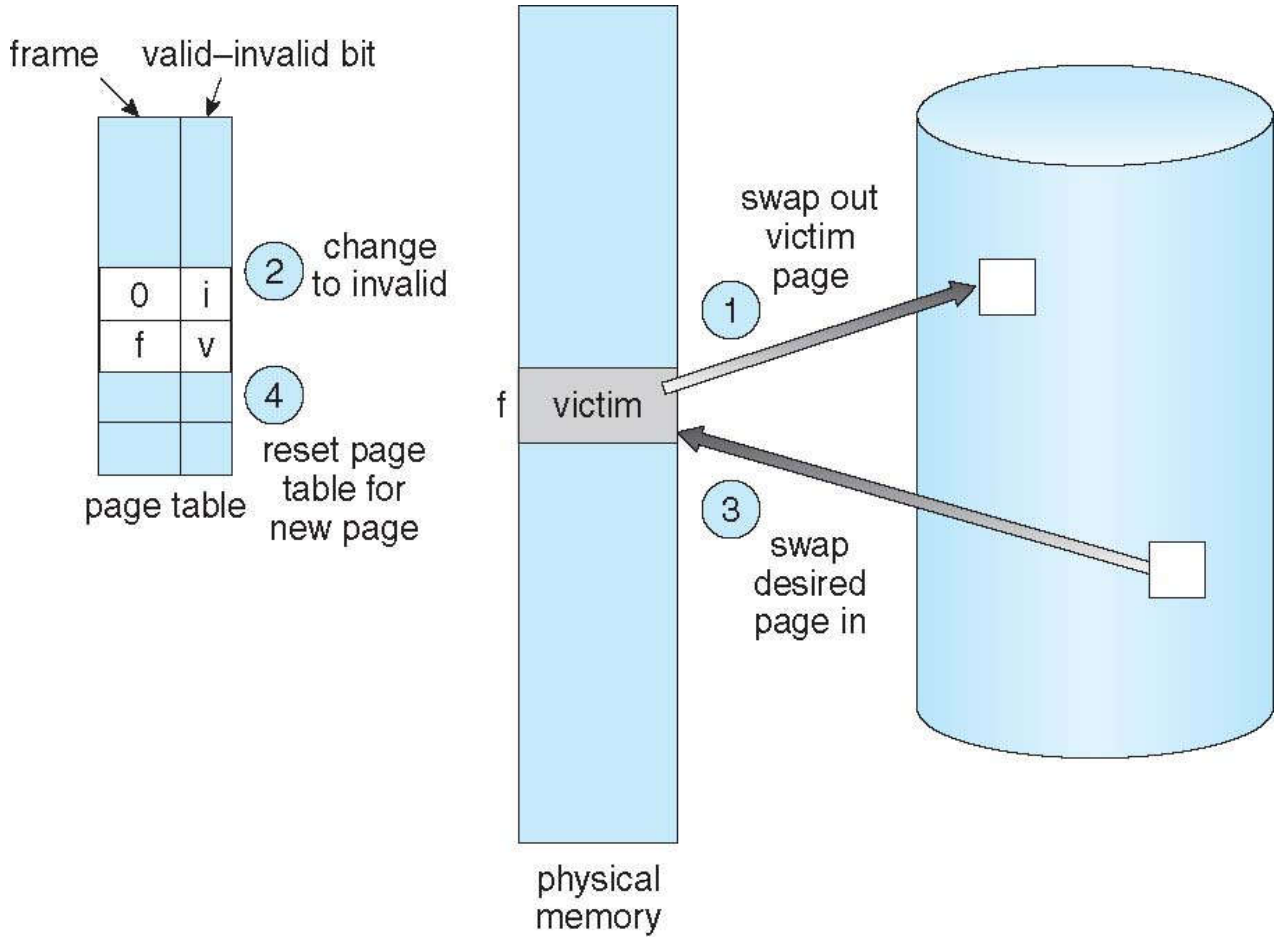
Need For Page Replacement



Page Fault Handler (with Page Replacement)

- To page in a page:
  - ✓ find the location of the desired page on disk
  - ✓ find a free frame:
    - – if there is a free frame, use it
    - – if there is none, use a page replacement policy to pick a victim frame, write victim frame to disk if dirty
  - ✓ bring the desired page into the free frame; update the page tables
  - ✓ restart the instruction that caused the trap
- Note now potentially 2 page I/O for one page fault ➠ increase EAT

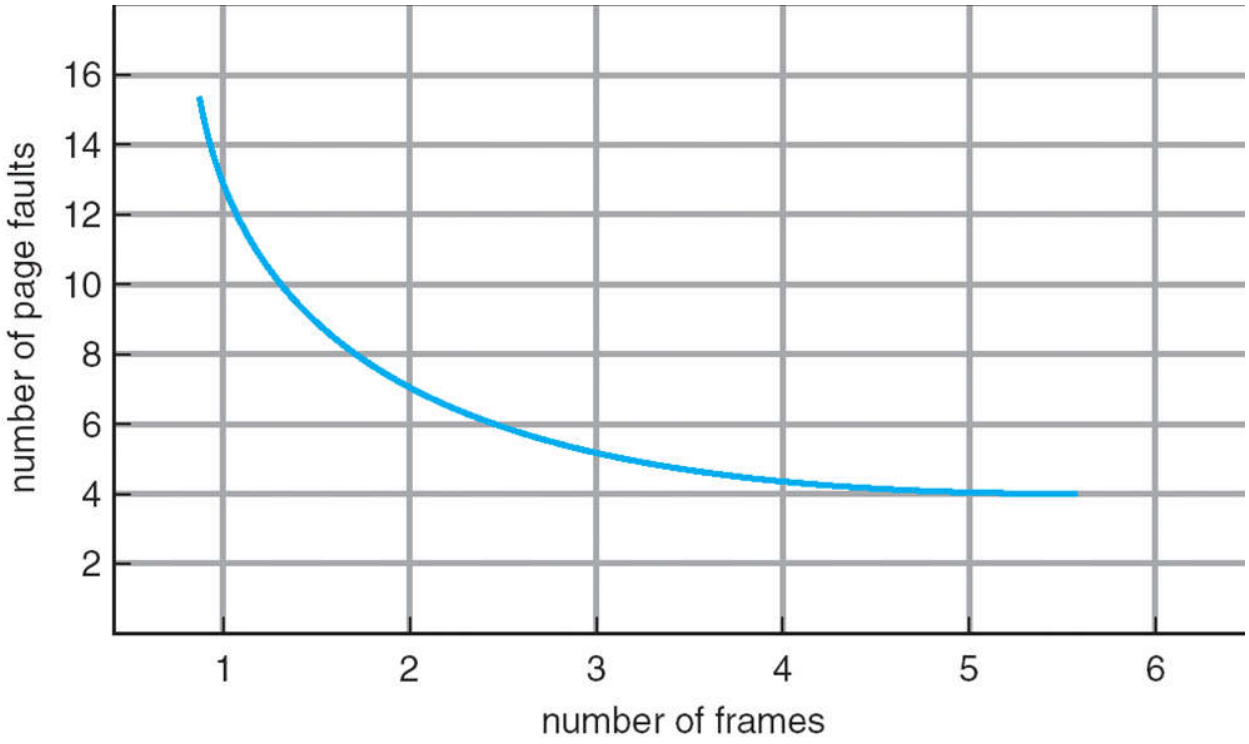
Page replacement



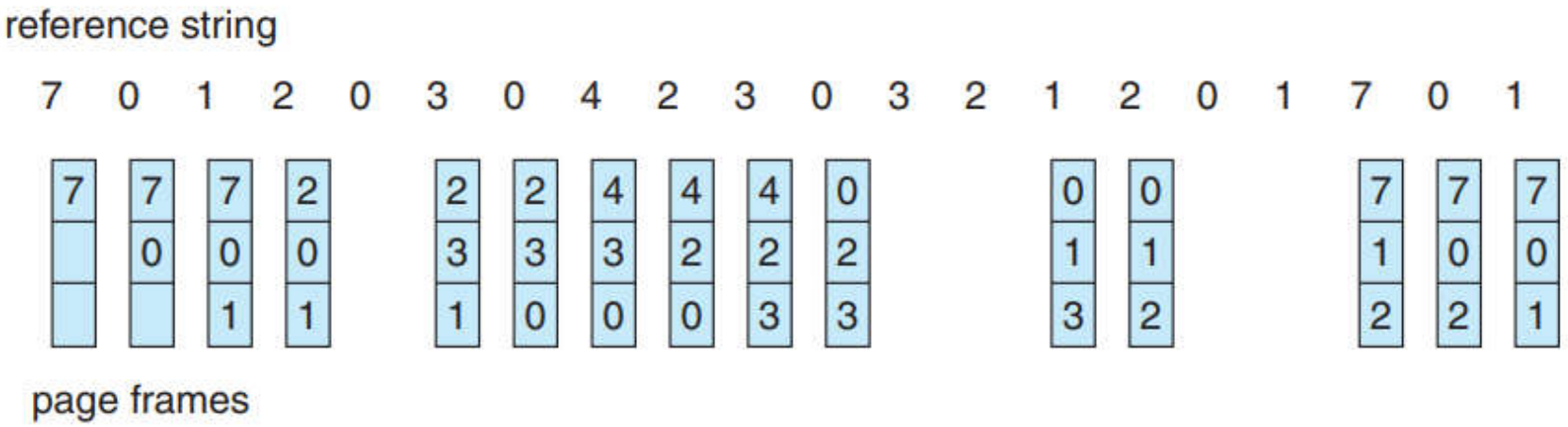
Page Replacement Algorithms

- Page-replacement algorithm should have lowest page-fault rate on both first access and re-access
  - ✓ FIFO, optimal, LRU, LFU, MFU…
- To evaluate a page replacement algorithm:
  - ✓ run it on a particular string of memory references (reference string)
    - – string is just page numbers, not full addresses
  - ✓ compute the number of page faults on that string
    - – repeated access to the same page does not cause a page fault
  - ✓ in all our examples, the reference string is
    - – 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1
    - – A,B,C,D

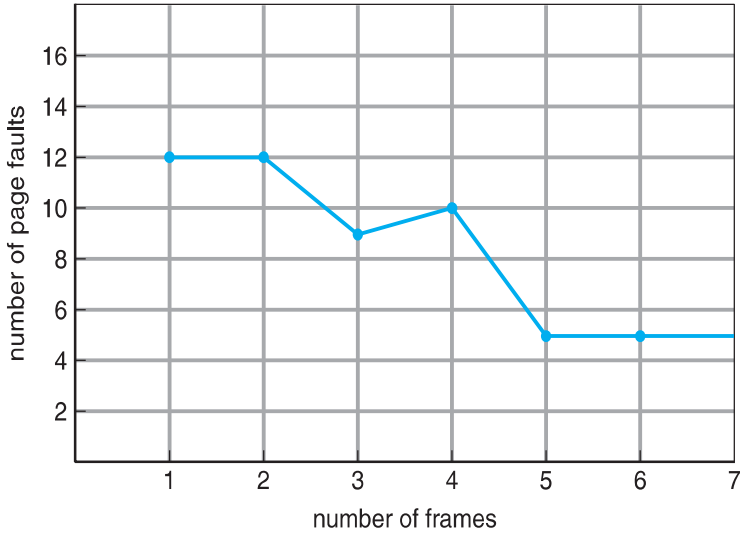
Page faults vs. # of frames



First-In-First-Out (FIFO) algorithm



- Problems
  - Much miss rate (high page fault)
  - FIFO anomaly (Belady's anomaly)
    - Check 1,2,3,4,1,2,5,1,2,3,4,5
      - 8 page faults for 3-page mem!
      - 10 page faults for 4-page mem!



Example: FIFO

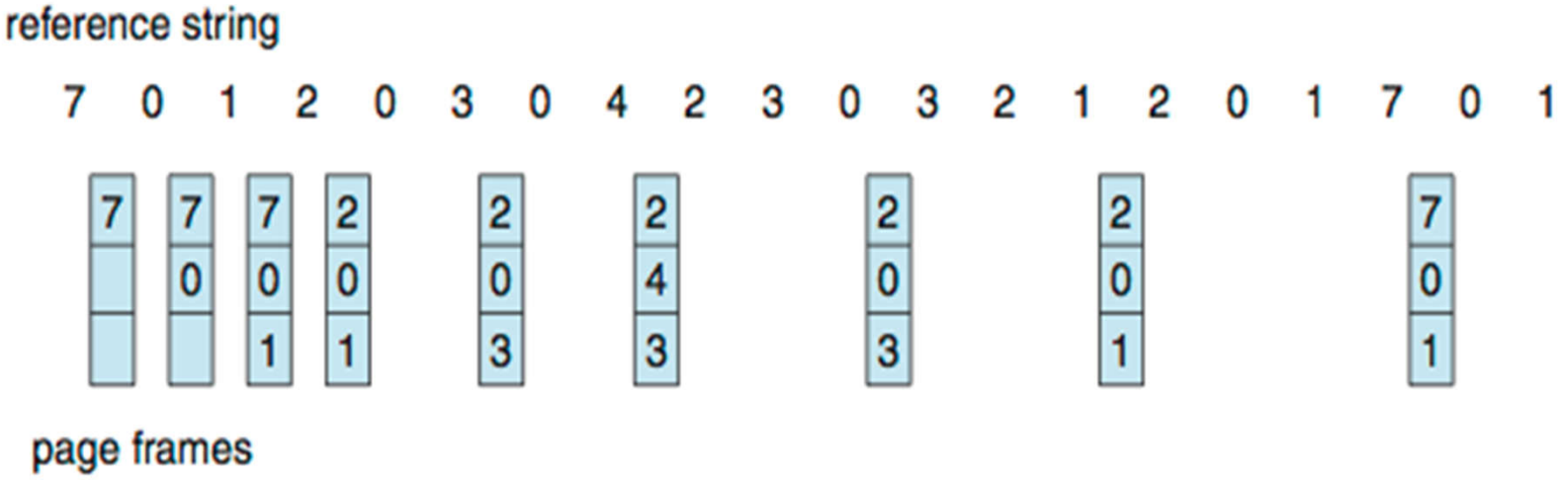
- Suppose we have 3 page frames, 4 virtual pages, and following reference stream:
  - A B C A B D A D B C B
- Consider FIFO Page replacement:

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

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

Optimal algorithm

Replace page that will not be used for longest period of time



- Problems
  - Who is aware of future?

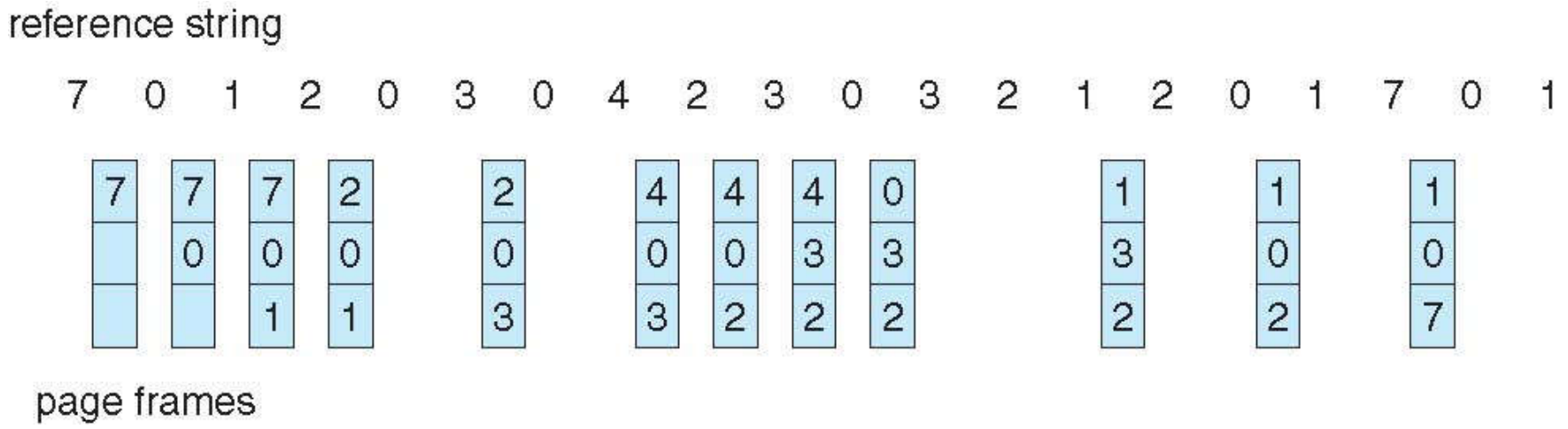
Example: Optimal

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

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

- MIN: 5 faults
  - Where will D be brought in? Look for page not referenced farthest in future

Least Recently Used (LRU)



- use the recent past as an approximation of the near future
- Better than FIFO but worse than OPT

Example: Least Recently Used (LRU)

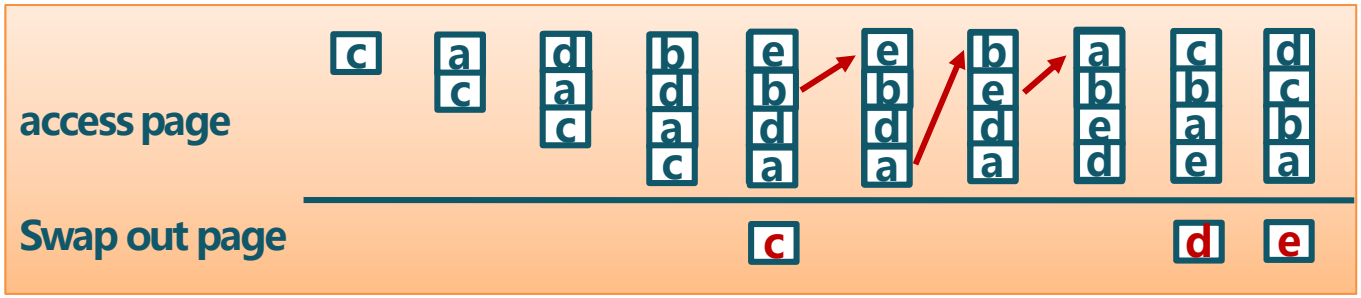
Time	0	1	2	3	4	5	6	7	8	9	10
Request order		c	a	d	b	e	b	a	b	c	d
Frame no 0	a	a	a	a	a	a	a	a	a	a	a
Frame no 1	b	b	b	b	b	b	b	b	b	b	b
Frame no 2	c	c	c	c	c → e	e	e	e	e	e → d	
Frame no 3	d	d	d	d	d	d	d	d	d	d → c	c
Page fault						●				●	●
Access time of frame						a=2 b=4 c=1 d=3				a=7 b=8 e=5 d=3	a=7 b=8 e=5 c=9

LRU Implementation

- Two possible implementations
  - Counter-based implementation
    - ✓ every page table entry has a counter
    - ✓ every time page is referenced, copy the clock into the counter
    - ✓ when a page needs to be replaced, search for page with smallest counter
      - min-heap can be used
  - Stack-based implementation
    - ✓ keep a stack of page numbers (in double linked list)
    - ✓ when a page is referenced, move it to the top of the stack
    - ✓ each update is more expensive, but no need to search for replacement

Stack-based LRU

time	0	1	2	3	4	5	6	7	8	9	10
Request order		c	a	d	b	e	b	a	b	c	d
Frame no 0	a	a	a	a	a	a	a	a	a	a	a
Frame no 1	b	b	b	b	b	b	b	b	b	b	b
Frame no 2	c	c	c	c	c	e	e	e	e	e	b
Frame no 3	d	d	d	d	d	d	d	d	d	c	c
Page fault						●				●	●



LRU Approximation Implementation

- Counter-based and stack-based LRU have high performance overhead
- Hardware provides a reference bit
- LRU approximation with a reference bit
  - ✓ associate with each page a reference bit, initially set to 0
  - ✓ when page is referenced, set the bit to 1 (done by the hardware)
  - ✓ replace any page with reference bit = 0 (if one exists)
    - – We do not know the order, however

Additional-Reference-Bits Algorithm

- Reordering the bits at regular intervals
  - ✓ Suppose we have 8-bits byte for each page
  - ✓ During a time interval (100ms), sets the high bit and shifts bit rights by 1 bit, and then discards the low-order bits

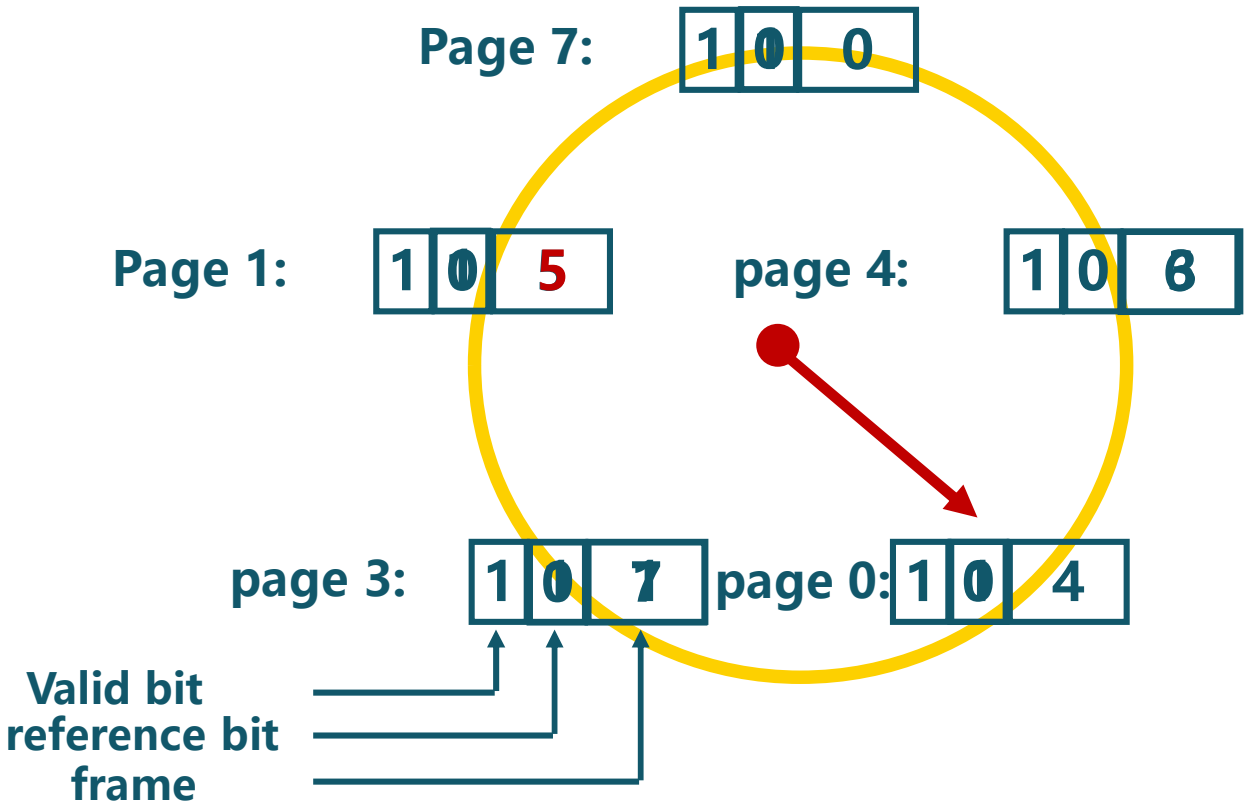
  For example:

  - ✓ 00000000 => has not been used in 8 time intervals
  - ✓ 11111111 => has been used in all time intervals
  - ✓ 11000100 vs 01110111 : which one is used more recently?

LRU Implementation

- Second-chance algorithm
  - ✓ Generally FIFO, plus hardware-provided reference bit
  - ✓ Clock replacement
  - ✓ 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

Clock replacement



Example :Clock replacement

time		0	1	2	3	4	5	6	7	8	9	10
Request			c	a	d	b	e	b	a	b	c	d
Frame no	0	a	a	a	a	a	→e	e	e	e	e	→d
	1	b	b	b	b	b	b	b	b	b	b	b
	2	c	c	c	c	c	c	c	→a	a	a	a
	3	d	d	d	d	d	d	d	d	d	→c	c
Page fault							●		●		●	●

page table

0	a	0	a	1	a	1	a	1	a	1	e	1	e	1	e	1	e	1	d
0	b	0	b	0	b	0	b	1	b	0	b	1	b	0	b	1	b	0	b
0	c	1	c	1	c	1	c	1	c	0	c	0	c	1	a	1	a	0	a
0	d	0	d	0	d	1	d	1	d	0	d	0	d	0	d	1	c	0	c

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 replace page in lowest non-empty class
  - ✓ Might need to search circular queue several times

Enhanced Second-Chance Algorithm

time	0	1	2	3	4	5	6	7	8	9	10
request		c	a^w	d	b^w	(e)	b	a^w	b	(c)	(d)
Frame no 0	a	a	a	a	a	a	a	a	a	a	a
Frame no 1	b	b	b	b	b	b	b	b	b	b	→d
Frame no 2	c	c	c	c	c	→e	e	e	e	e	e
Frame no 3	d	d	d	d	d	d	d	d	d	→c	c
Page fault						●				●	●

Page table

00 a	00 a	11 a	11 a	11 a	00 a	00 a	11 a	11 a	11 a	00 a*
00 b	00 b	00 b	00 b	11 b	00 b	10 b	10 b	10 b	10 b	10 d
00 c	10 c	10 c	10 c	10 c	10 e	10 e	10 e	10 e	10 e	00 e
00 d	00 d	00 d	10 d	10 d	00 d	00 d	00 d	00 d	10 c	00 c

Counting-based Page Replacement

- Keep a counter of the number of references that have been made to each page
  - ✓ Not common
- **Least Frequently Used (LFU) Algorithm**
  - ✓ replaces page with smallest count
- **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

Example: LFU

- Suppose accessing order :a- > 8 b- > 5 c- > 6 d- > 2

time		0	1	2	3	4	5	6	7	8	9	10
rquest			c^{7}	a^{1}	d^{14}	b^{5}	e^{18}	b^{1}	a^{19}	b^{20}	c^{20}	d^{17}
Frame no	0	a^{8}	a^{8}	a^{9}	a^{9}	a^{9}	e^{18}	e^{18}	e^{18}	e^{18}	e^{18}	d^{17}
	1	b^{5}	b^{5}	b^{5}	b^{5}	b^{10}	b^{10}	b^{11}	a^{19}	a^{19}	a^{19}	a^{19}
	2	c^{6}	c^{13}	c^{13}	c^{13}	c^{13}	c^{13}	c^{13}	c^{13}	b^{20}	b^{20}	b^{20}
	3	d^{2}	d^{2}	d^{2}	d^{16}	d^{16}	d^{16}	d^{16}	d^{16}	d^{16}	c^{20}	c^{20}
Page fault							●		●	●	●	●

Page-Buffering Algorithms

- Keep a pool of free frames, always
  - ✓ Then frame available when needed, not found at fault time
  - ✓ Read page into free frames without waiting for victims to write out
    - Restart as soon as possible
  - ✓ When convenient, evict victim
- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty: this page can be replaced without writing pages to backing store
- Possibly, keep free frame contents intact and note what is in them - a kind of cache
  - If referenced again before reused, no need to load contents again from disk
    - cache hit

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
  - 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
  - But possibly underutilized memory

- Allocation algorithms
  - Equal allocation
  - Proportional allocation
    - Proportional to size of program

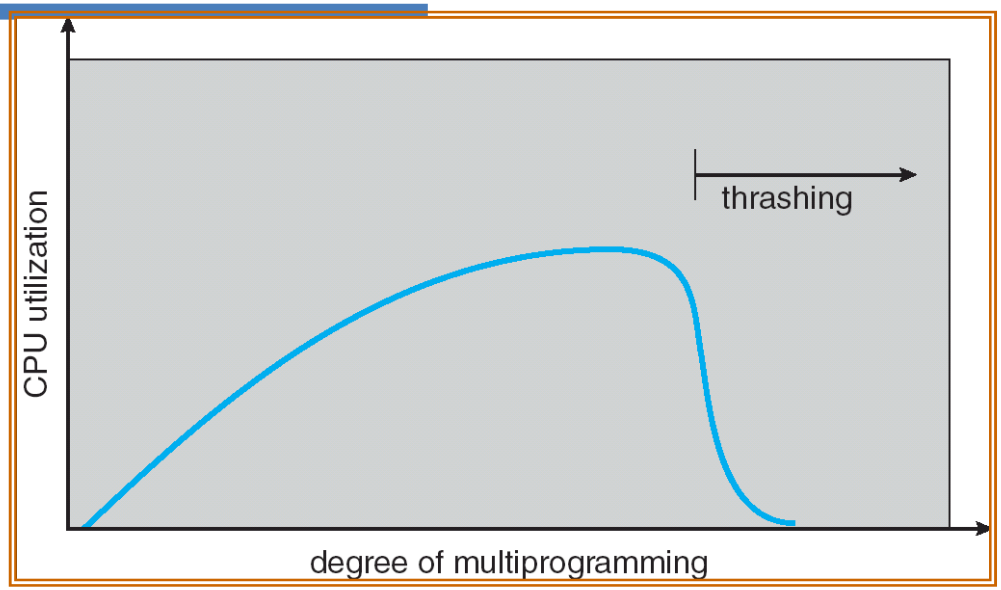
Thrashing

Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system

- **Thrashing** ≡ a process is busy swapping pages in and out

Thrashing



- Thrashing leads to:
  - low CPU utilization
  - operating system spends most of its time swapping to disk
- Thrashing ≡ a process is busy swapping pages in and out
- Questions:
  - How do we detect Thrashing?
  - What is best response to Thrashing?

Demand paging vs. thrashing

- Why does demand paging work?

Locality model

  - Process migrates from one locality to another
  - Localities may overlap

- Why does thrashing occur?

Σ size of locality > total memory size

  - Limit effects by using local or priority page replacement

solutions

Option 1

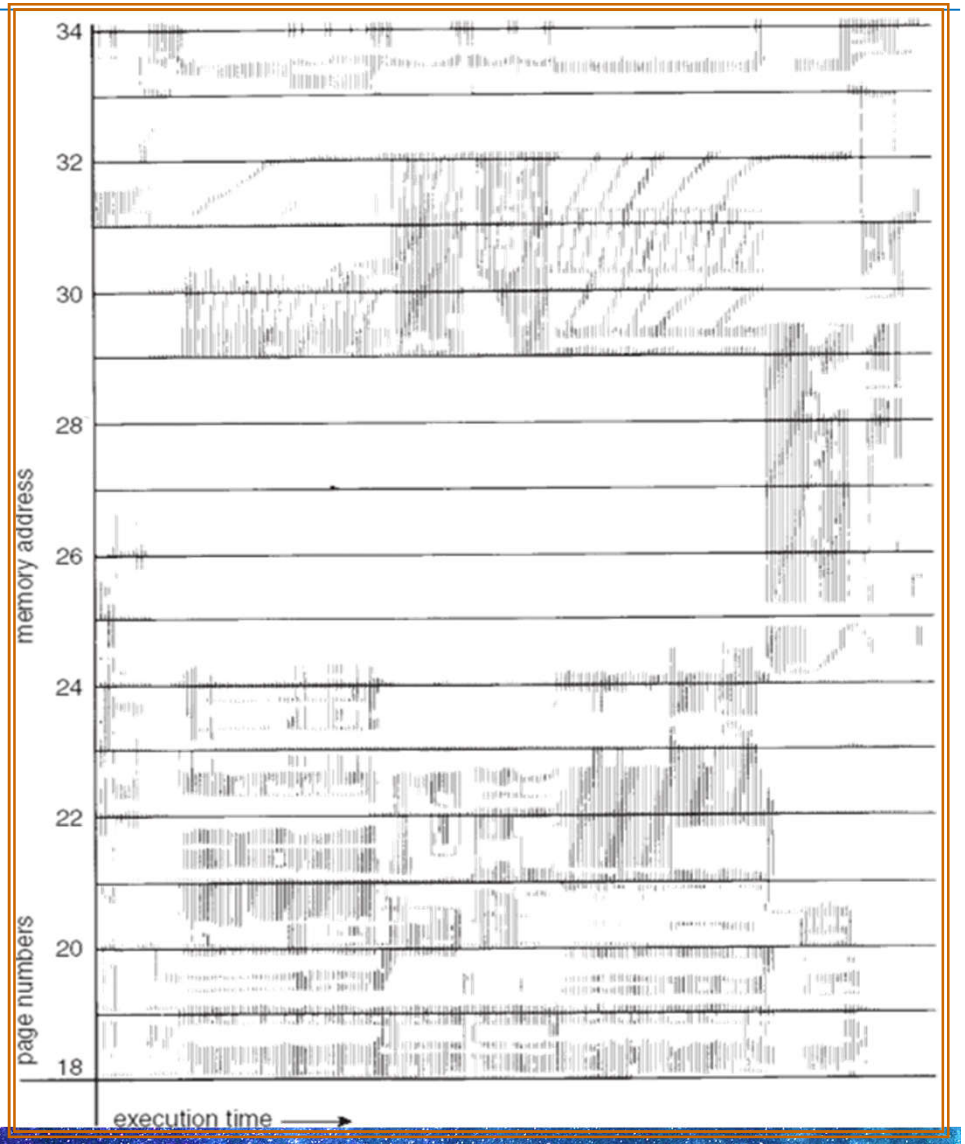
- Limit thrashing effects by using local or priority page replacement
  - One process starts thrashing does not affect others -> it cannot cause other processes thrashing

Option 2

- Provide a process with as many frames as it needs.

Locality In A Memory-Reference Pattern

- Program Memory Access Patterns have temporal and spatial locality
  - Group of Pages accessed along a given time slice called the “Working Set”
  - Working Set defines minimum number of pages needed for process to behave well
- Not enough memory for Working Set ⇒ Thrashing
  - Better to swap out process?

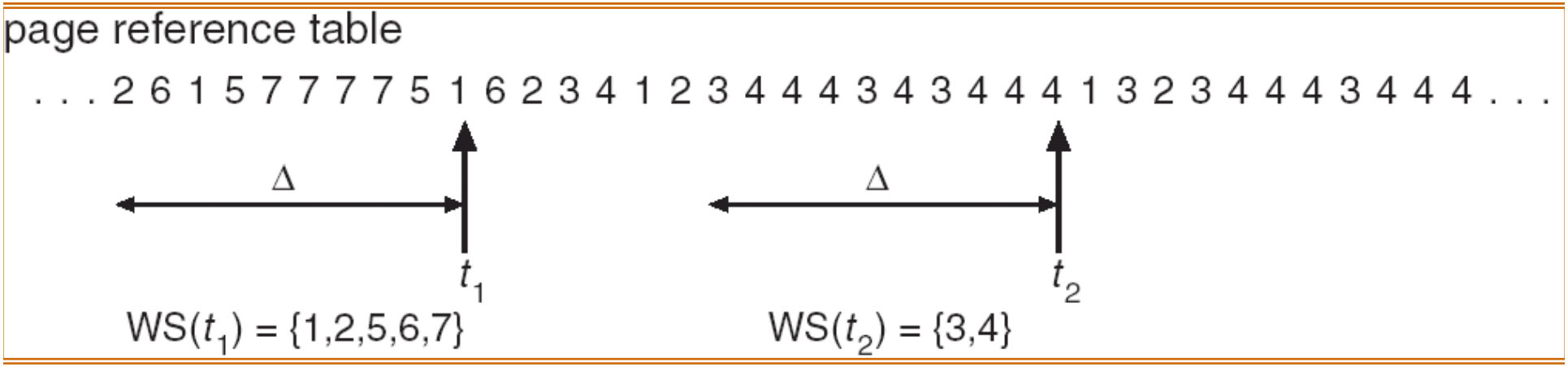


1) Working-set model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instructions
- WSS_i (working set 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
  - Approximation of locality
- if $D > m \Rightarrow$ Thrashing
- Policy: if $D >$ m then
  - suspend or
  - swap out one of the processes

Working-Set Model

- if Δ= 10 memory references, then the working set at time *t*1 is *{*1, 2, 5, 6, 7*}*.
- At time *t*2, the working set has changed to *{*3, 4*}*

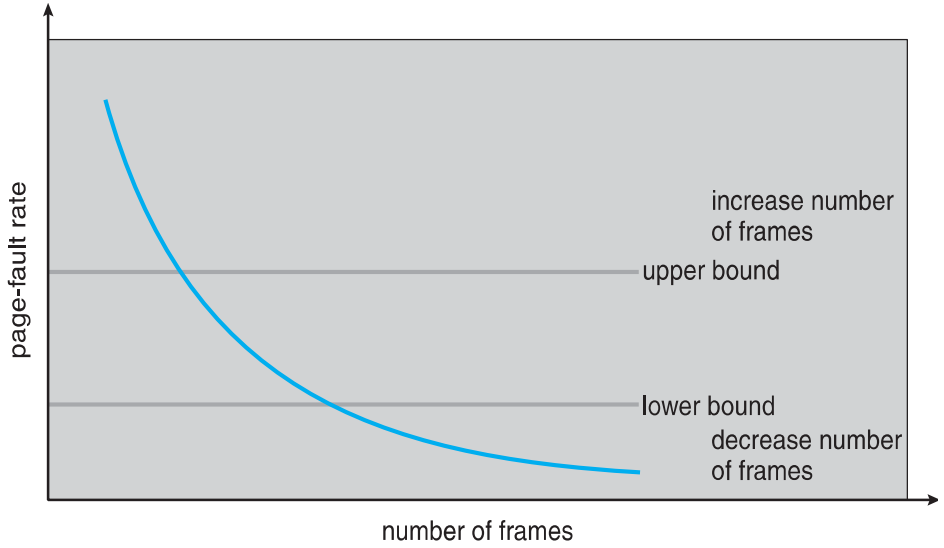


Keeping track of the working set

- Approximate with interval timer + a reference bit
- Example: Δ = 10,000
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

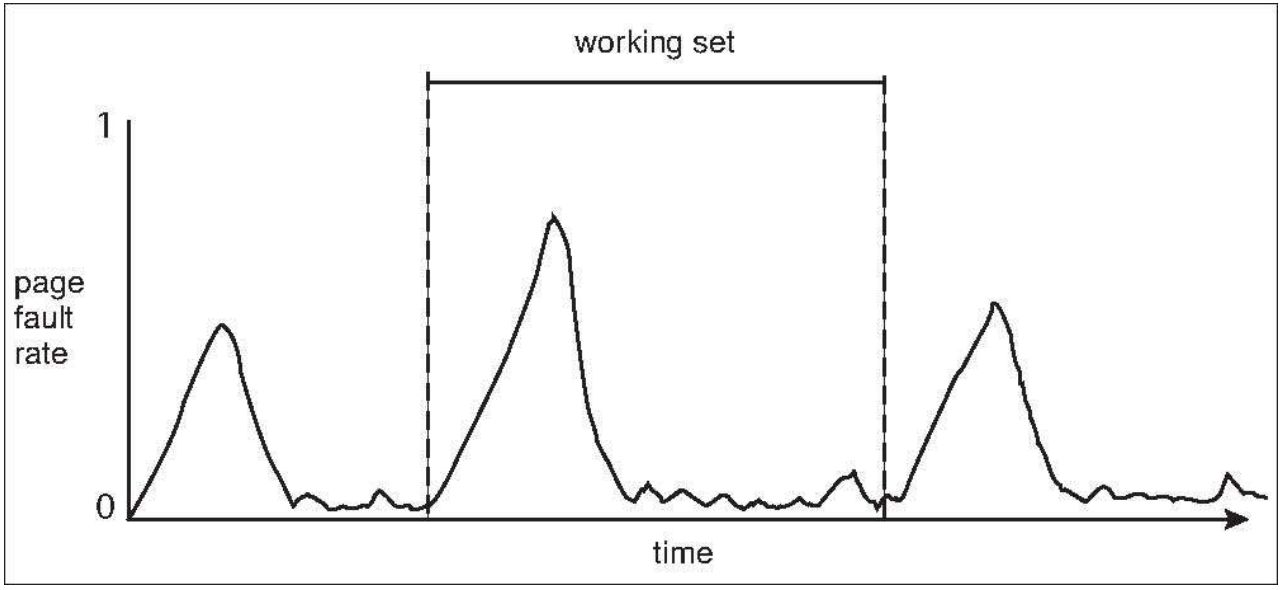
2) Page-fault frequency

- More direct approach than WSS
- 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



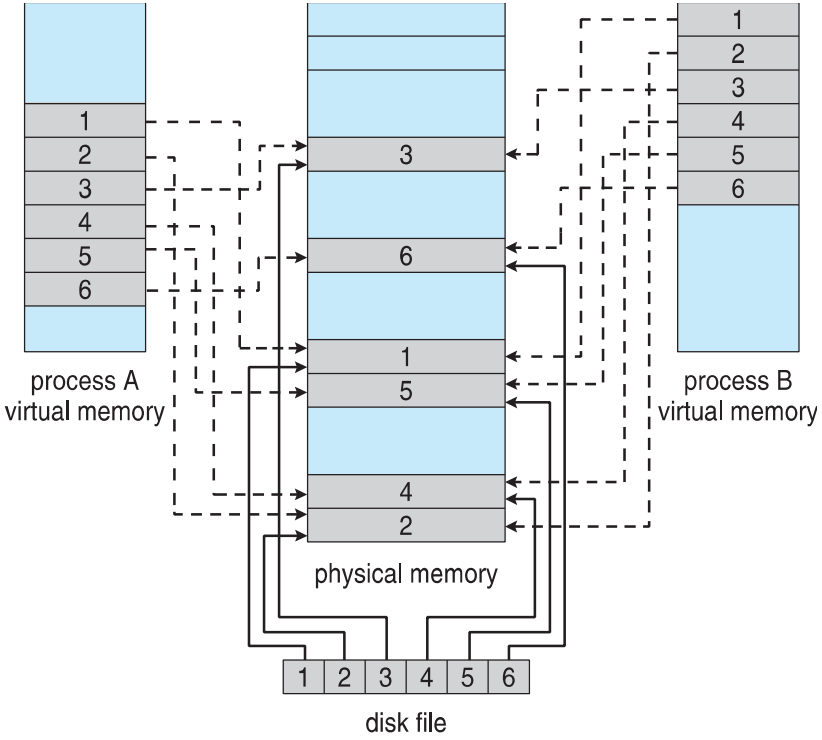
Memory mapped files, IOs

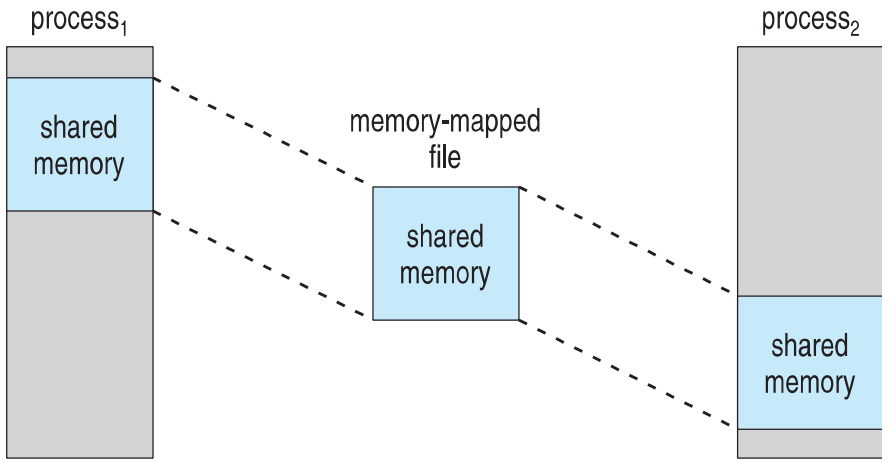
Memory-mapped files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by **mapping** a disk block to a page in memory

- A file is initially read using demand paging
  - A page-sized portion of the file is read from the file system into a physical page
  - Subsequent reads/writes to/from the file are treated as ordinary memory accesses

Memory mapped files





Shared Memory via Memory-Mapped I/O

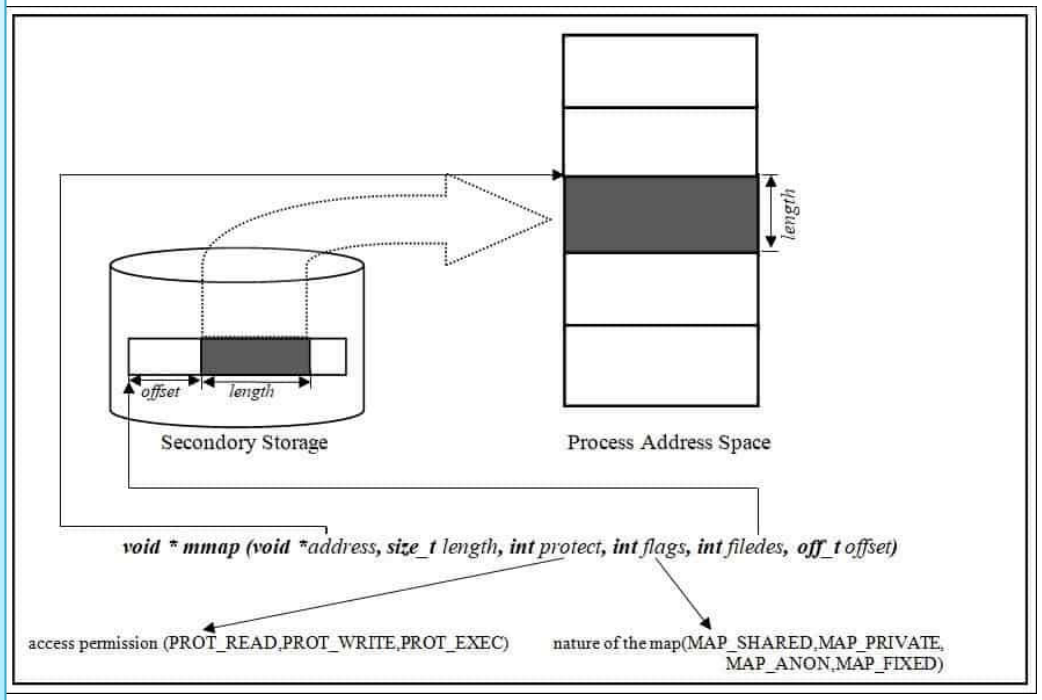
Some OSes uses memory mapped files for standard I/O

mmap() function in C

```
#include <stdio.h>
#include <sys/mman.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
int main(int argc, char *argv[]){
   const char *filepath = ”example.txt”;
   int fd = open(filepath, O_RDONLY);
   struct stat statbuf;
   int err = fstat(fd, &statbuf);

   char *ptr = mmap(NULL,statbuf.st_size,
         PROT_READ|PROT_WRITE,MAP_SHARED,
         fd,0);
   if(ptr == MAP_FAILED){
      printf("Mapping Failed\n");
      return 1;
   }
   close(fd);

   ssize_t n = write(1,ptr,statbuf.st_size);
   err = munmap(ptr, statbuf.st_size);
   return 0;
}
```



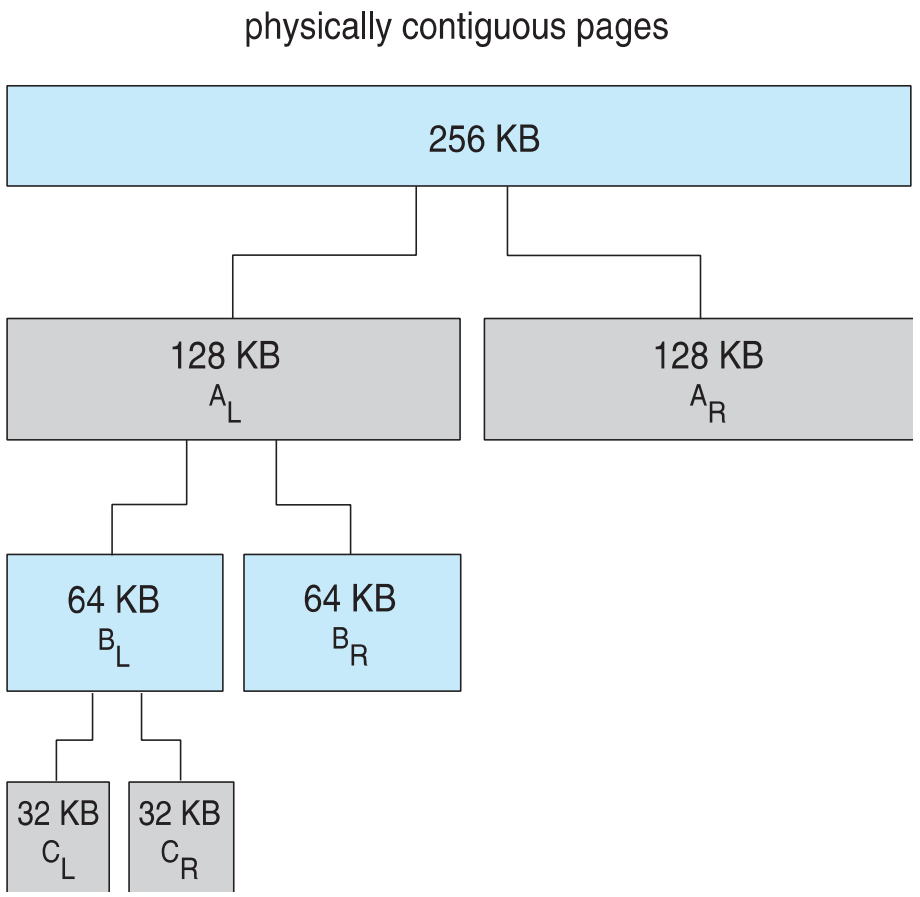
Kernel memory allocation
&
Virtual memory allocation

1) Buddy system allocator

- 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 - fragmentation

Buddy system allocator scheme

- 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

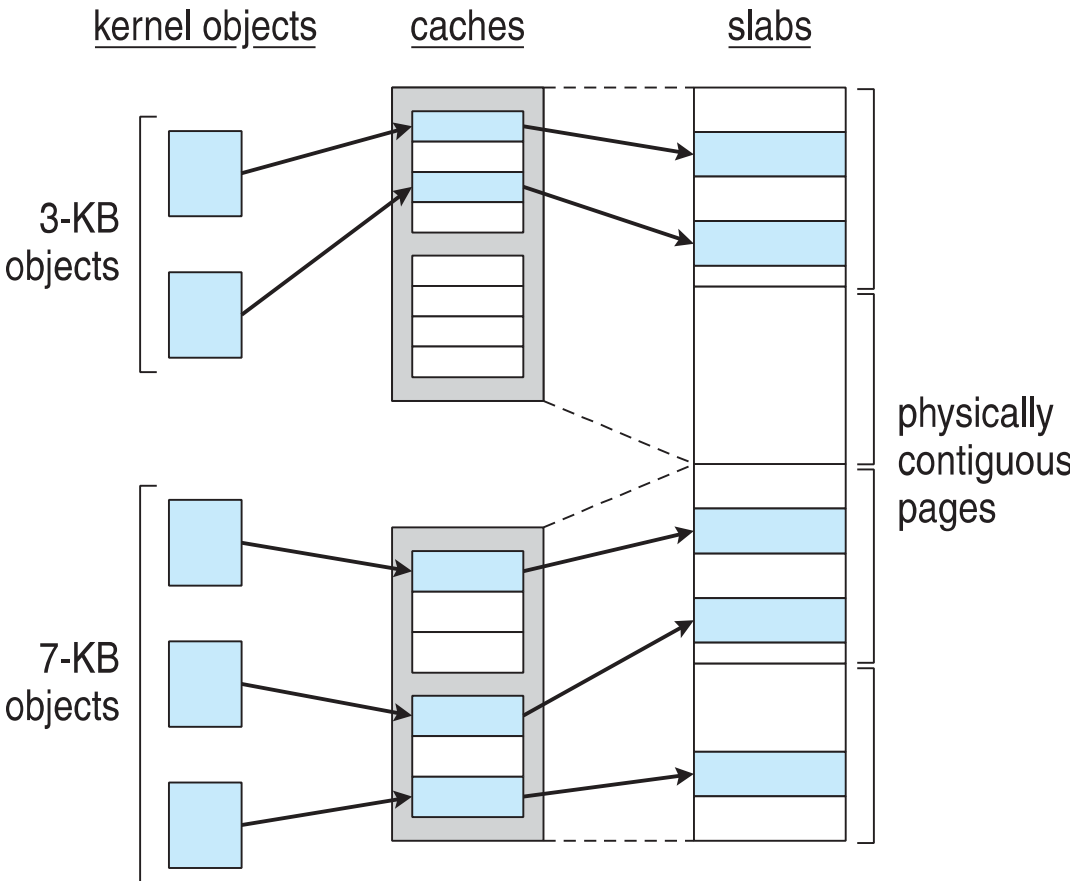


2) Slab allocator

- Alternate strategy
- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache filled with **objects** – instantiations of the data structure
- When cache created, filled with objects marked as **`free`**
- When structures stored, objects marked as **`used`**
- If slab is full of used objects, next object allocated from empty slab
  - If no empty slabs, new slab allocated

Slab allocation scheme

- Three possible states:
  - Full
  - Empty
  - Partial
- Pros：
  - No memory is wasted due to fragmentation
  - Memory requests can be satisfied quickly



Some important points

I) Prepaging

- 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
  - But if prepaged pages are unused, I/O and memory was wasted
  - Assume s pages are prepaged and α ($0 \leq \alpha \leq 1$)of the pages is used
    - Is cost of $s * \alpha$ save pages faults > or < than the cost of prepaging
      $s * (1- \alpha)$ unnecessary pages?
    - α near zero $\Rightarrow$ prepaging loses

II) Page Size

- Sometimes OS designers have a choice
  - Especially if running on custom-built CPU

- Page size selection must take into consideration:
  - Fragmentation
  - Page table size
  - I/O overhead
  - Number of page faults
  - Locality
  - TLB size and effectiveness

- 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

III) 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

IV) 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

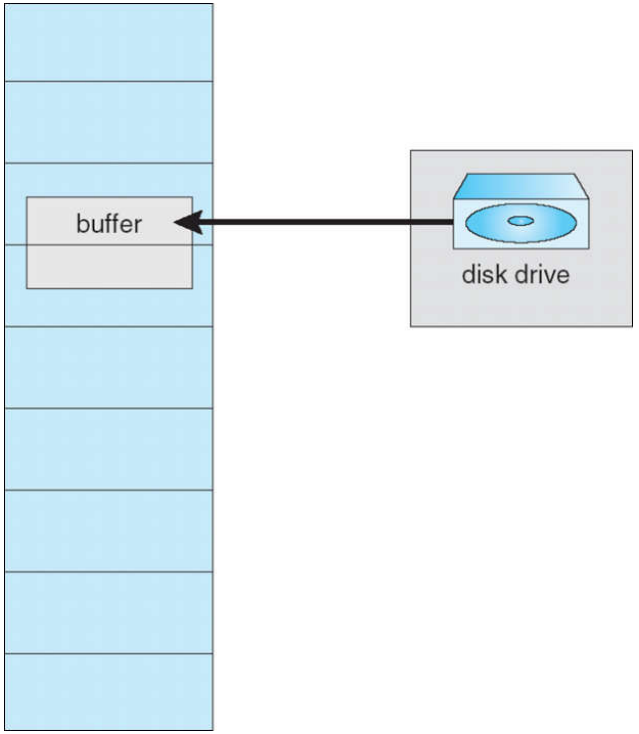
- Program 2

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

128 page faults

V) IO lock

- I/O Interlock – Pages must sometimes be locked into memory
- Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm
- Pinning of pages to lock into memory



Summary

- Replacement policies
  - FIFO: Place pages on queue, replace page at end
  - MIN: Replace page that will be used farthest in future
  - LRU: Replace page used farthest in past
- Clock Algorithm: Approximation to LRU
  - Arrange all pages in circular list
  - Sweep through them, marking as not “in use”
  - If page not “in use” for one pass, than can replace
- N^{th}-chance clock algorithm: Another approximate LRU
  - Give pages multiple passes of clock hand before replacing
- Second-Chance List algorithm: Yet another approximate LRU
  - Divide pages into two groups, one of which is truly LRU and managed on page faults.
- Working Set:
  - Set of pages touched by a process recently
- Thrashing: a process is busy swapping pages in and out
  - Process will thrash if working set doesn’t fit in memory
  - Need to swap out a process

Questions?