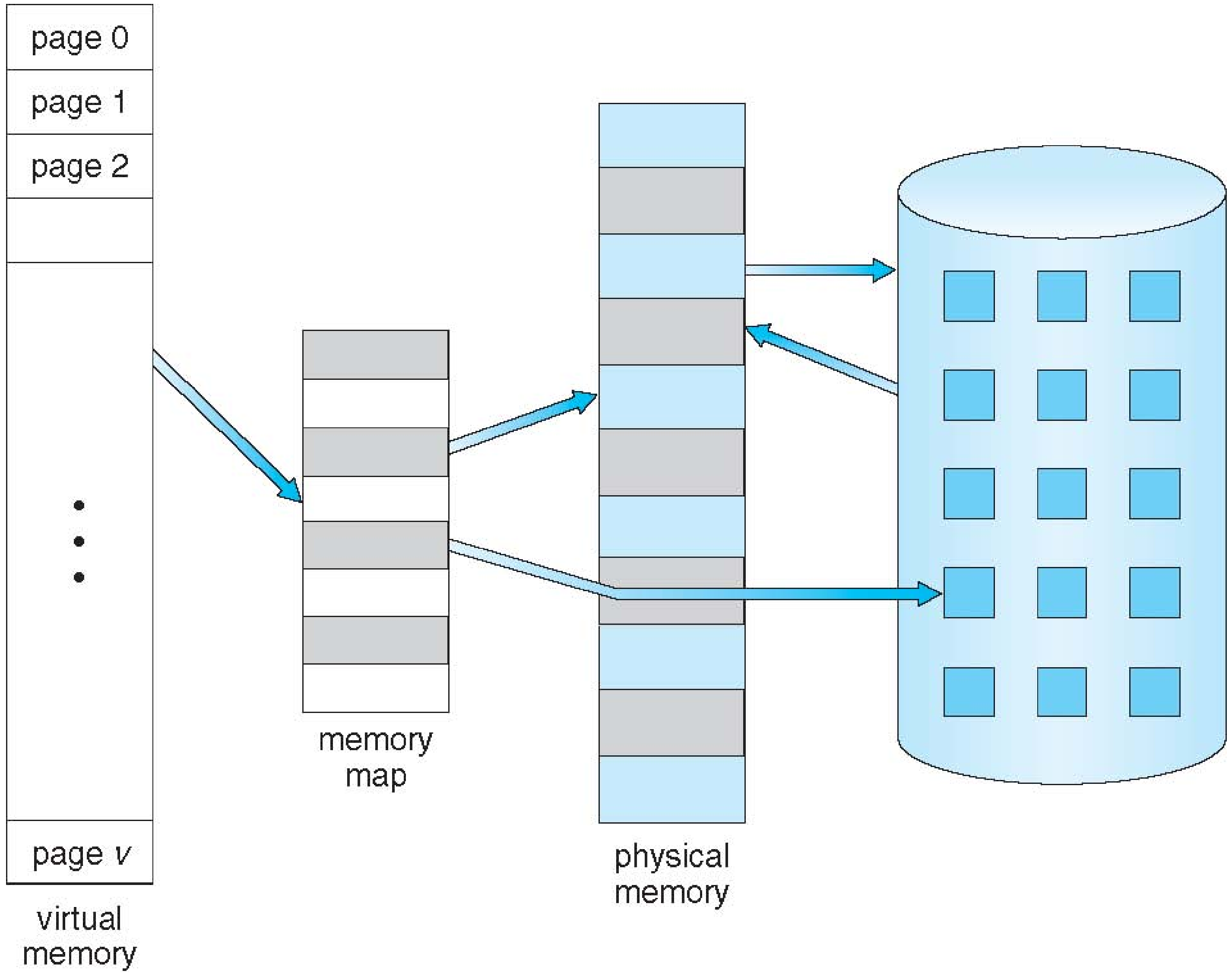
Amirkabir University of Technology
(Tehran Polytechnic)
Department of Computer Engineering

Virtual Memory Management

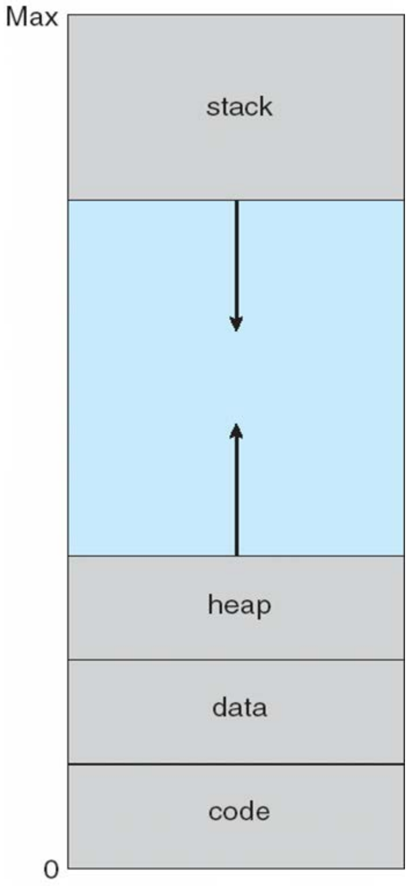
Hamid R. Zarandi
h_zarandi@aut.ac.ir

Virtual memory > physical memory

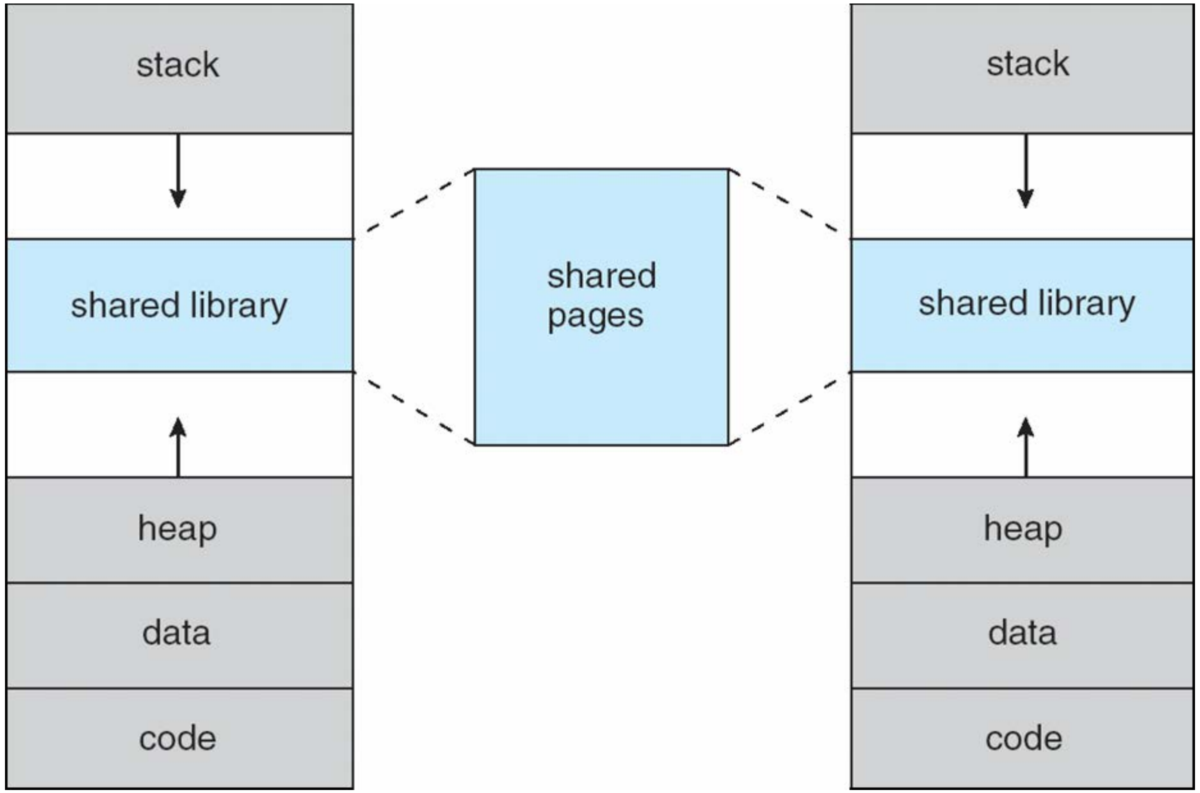


Virtual address space

- Code, date, heap, stack
  - Possible of Sparse addresses



- Shared library usage

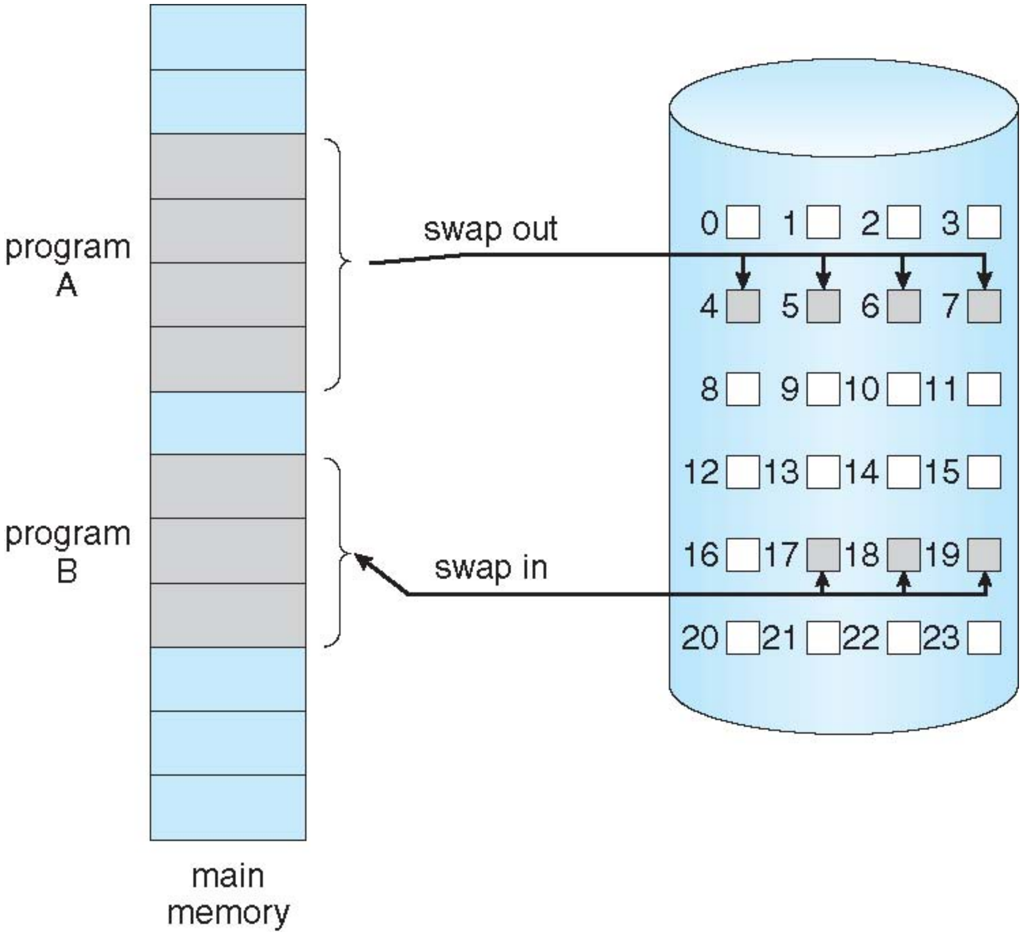


Demand paging

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

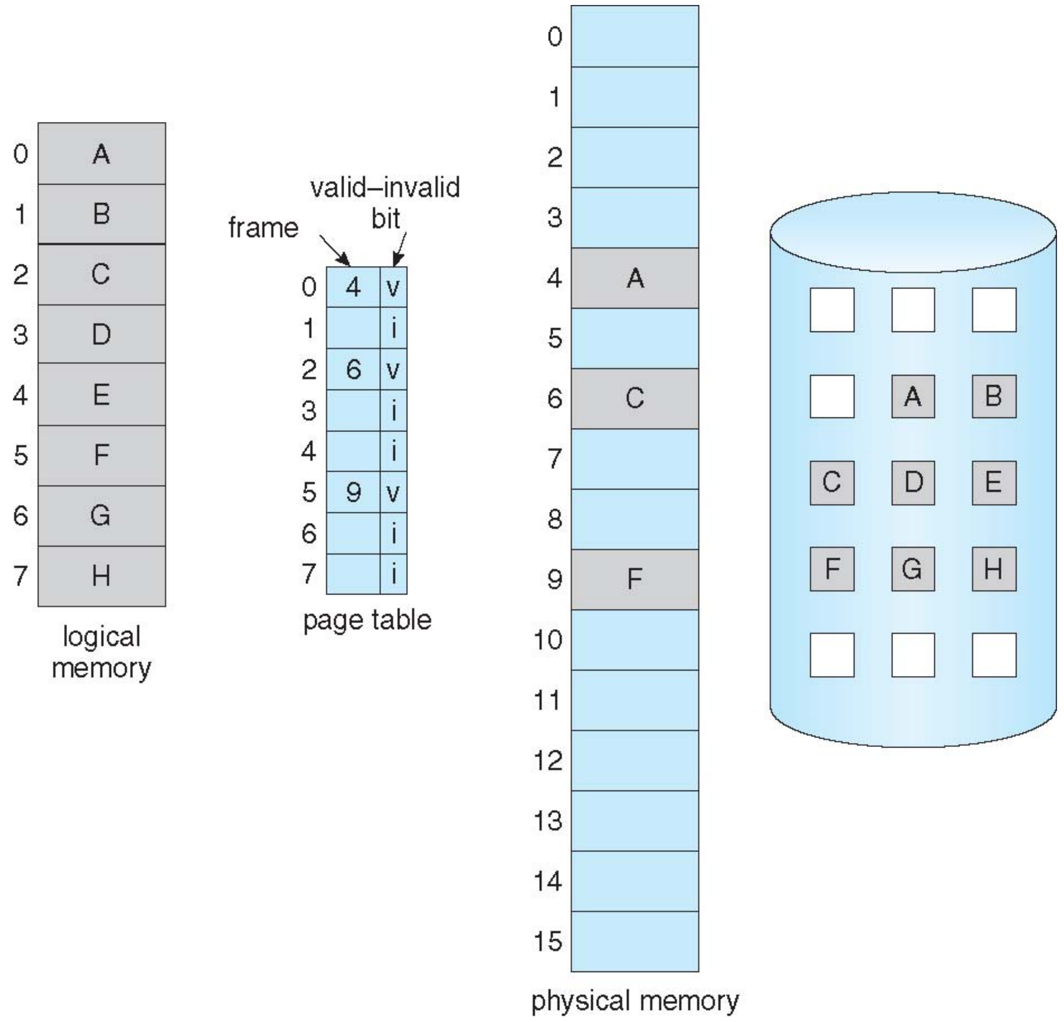
- Lazy swapper – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a pager

- Swapper is different from Pager



Demand paging implementation

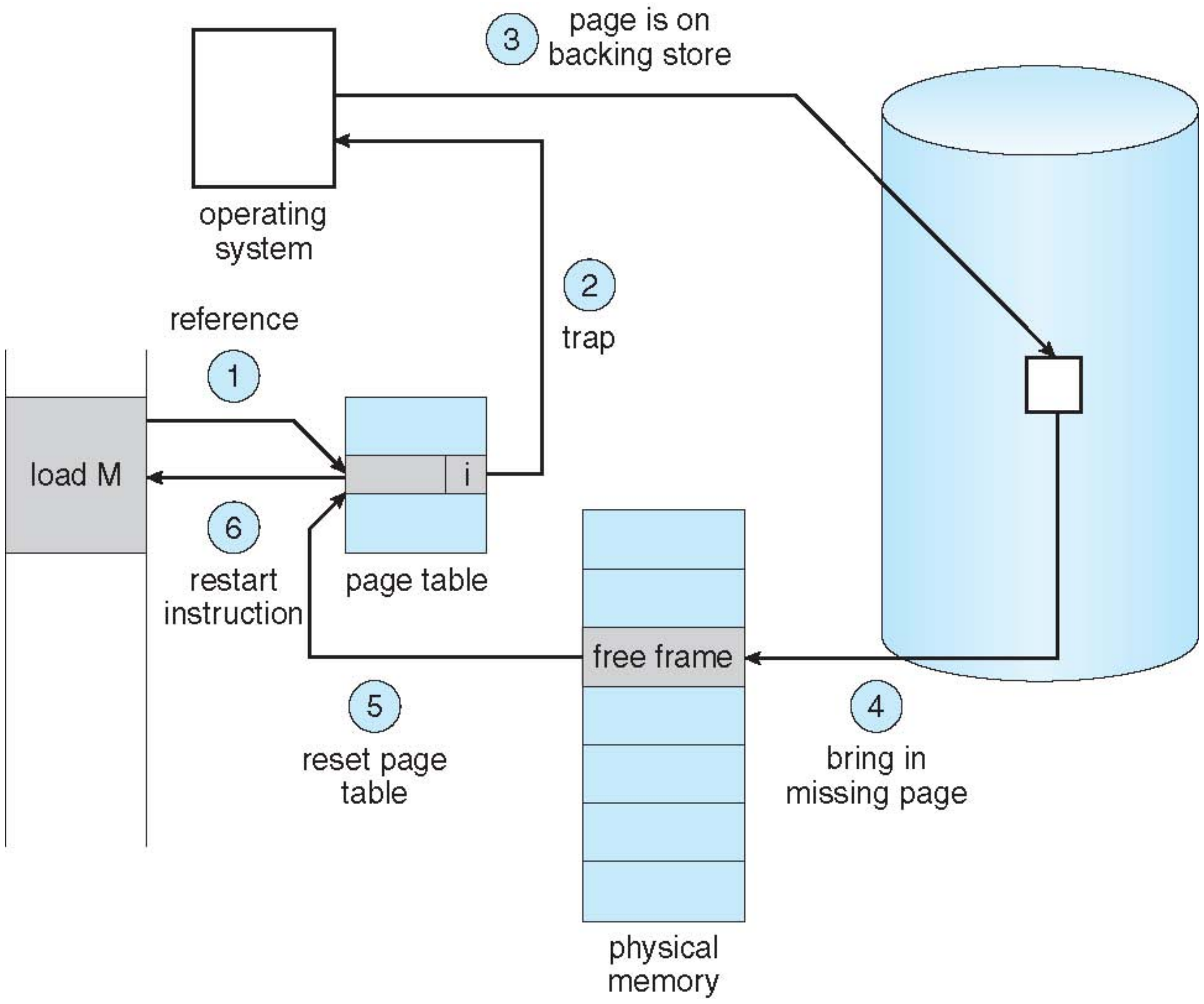
- **Help of valid-invalid bit**



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.

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:
    1. Wait in a queue for this device until the read request is serviced
    2. Wait for the device seek and/or latency time
    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)

$$
\begin{aligned}
EAT = {} & (1 - p) \times \text{memory access} \\
& + p\,(\text{page fault overhead} \\
& \quad + \text{swap page out} \\
& \quad + \text{swap page in})
\end{aligned}
$$

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

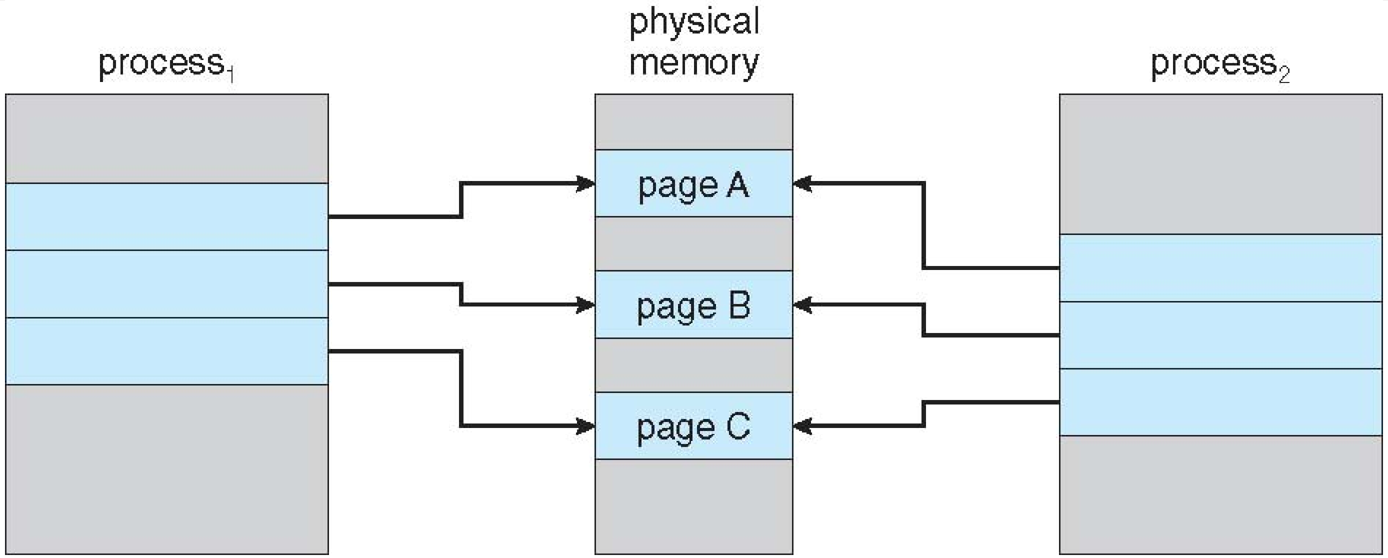
  = 200 + p x 7,999,800
- If one access out of 1,000 causes a page fault, then

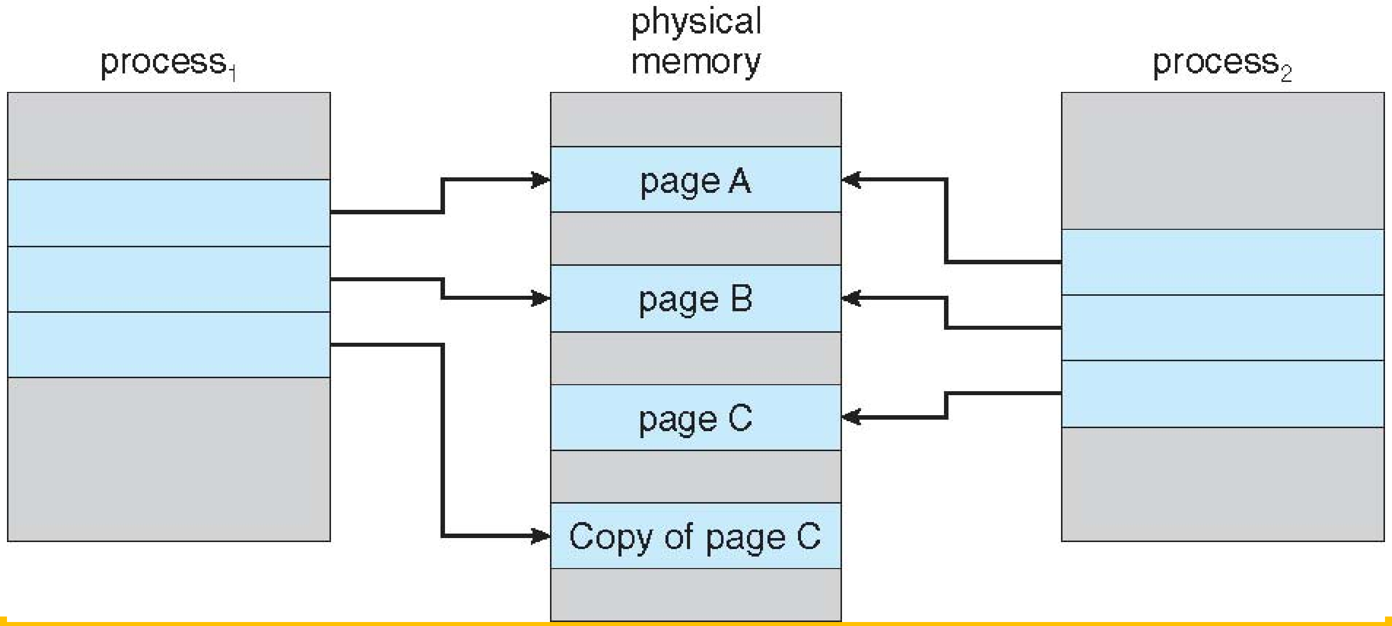
  EAT = 8.2 microseconds.

  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

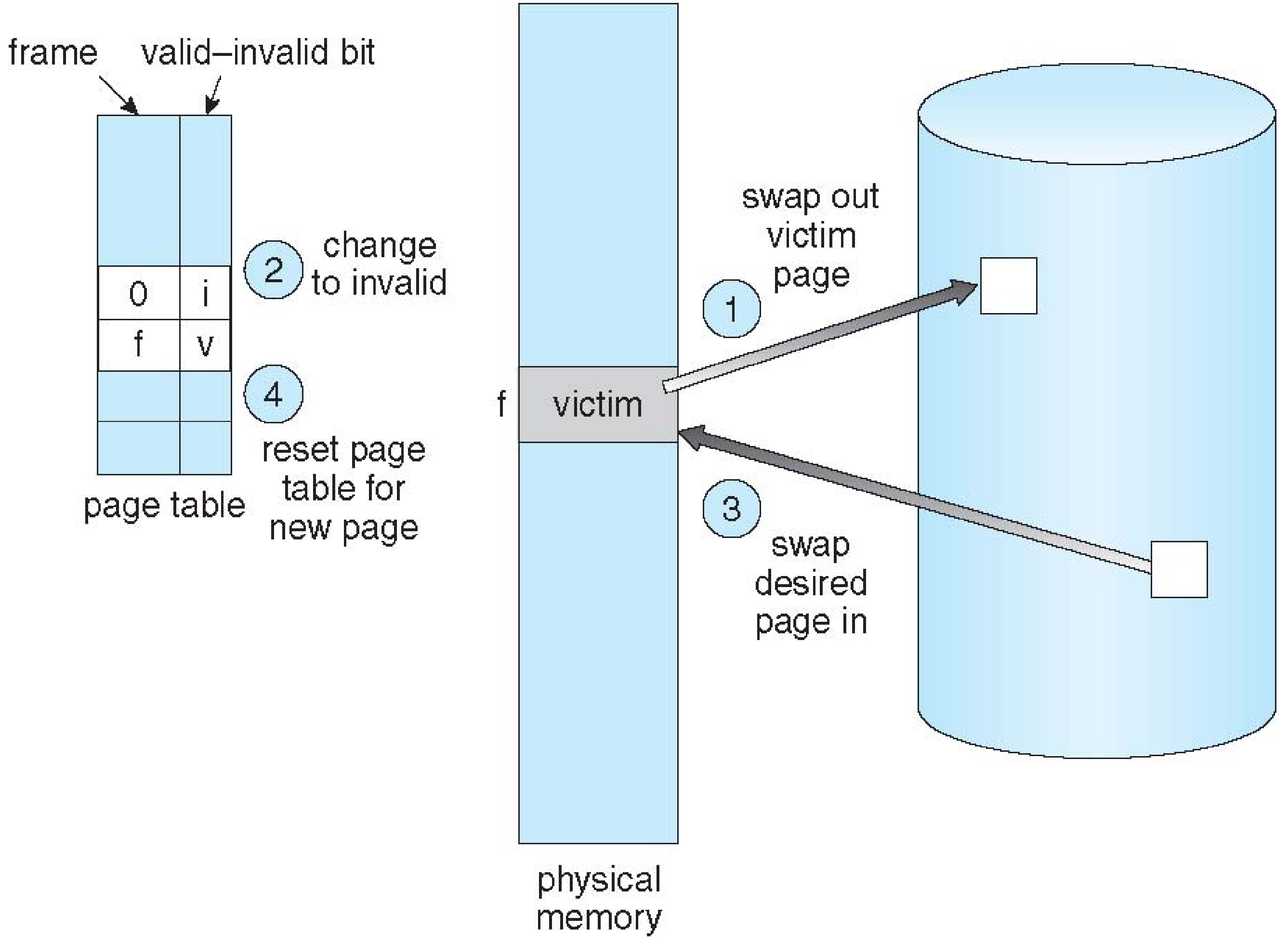
- **Copy-on-Write** (COW) allows both parent and child processes to initially *share* the same pages in memory



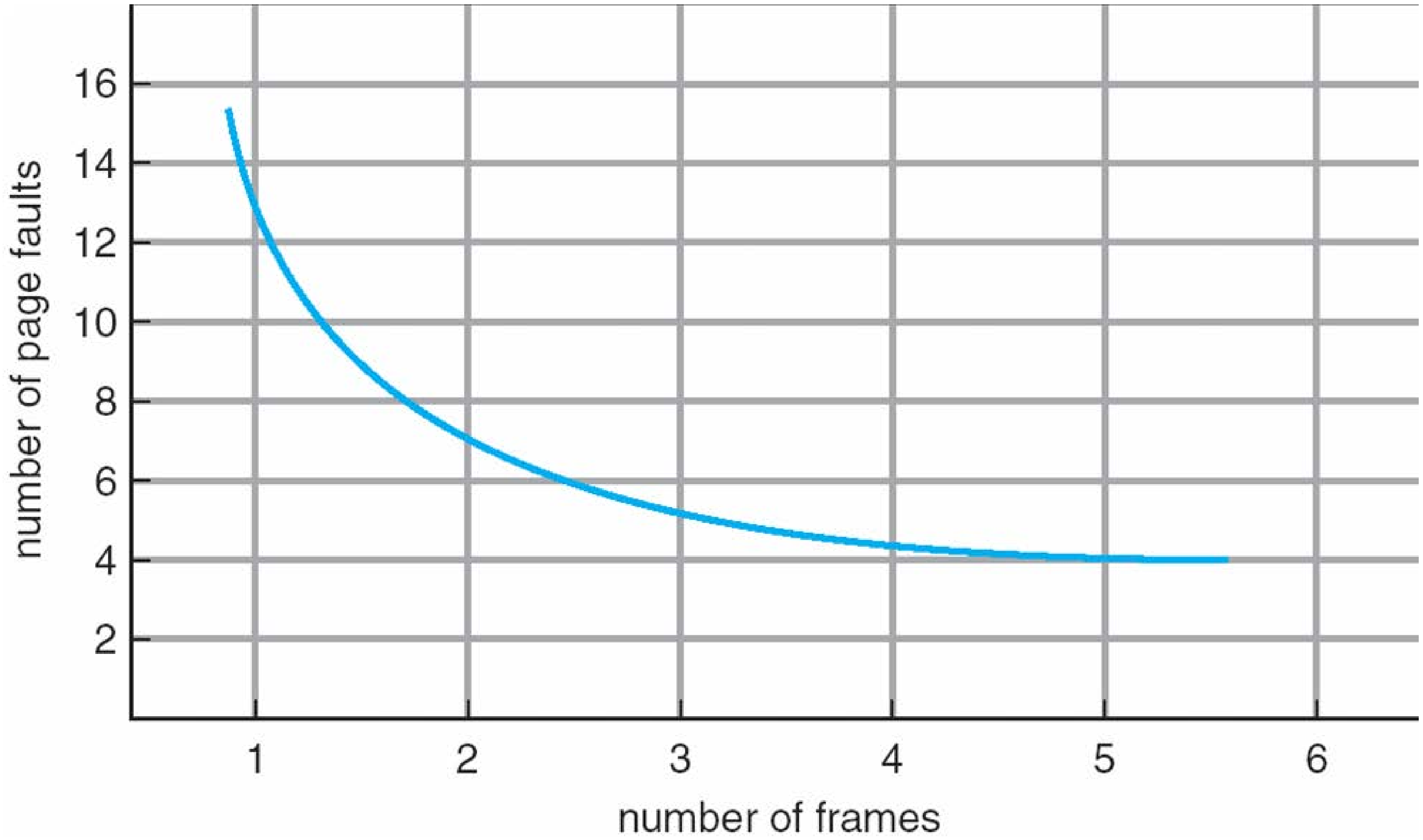


Page replacement

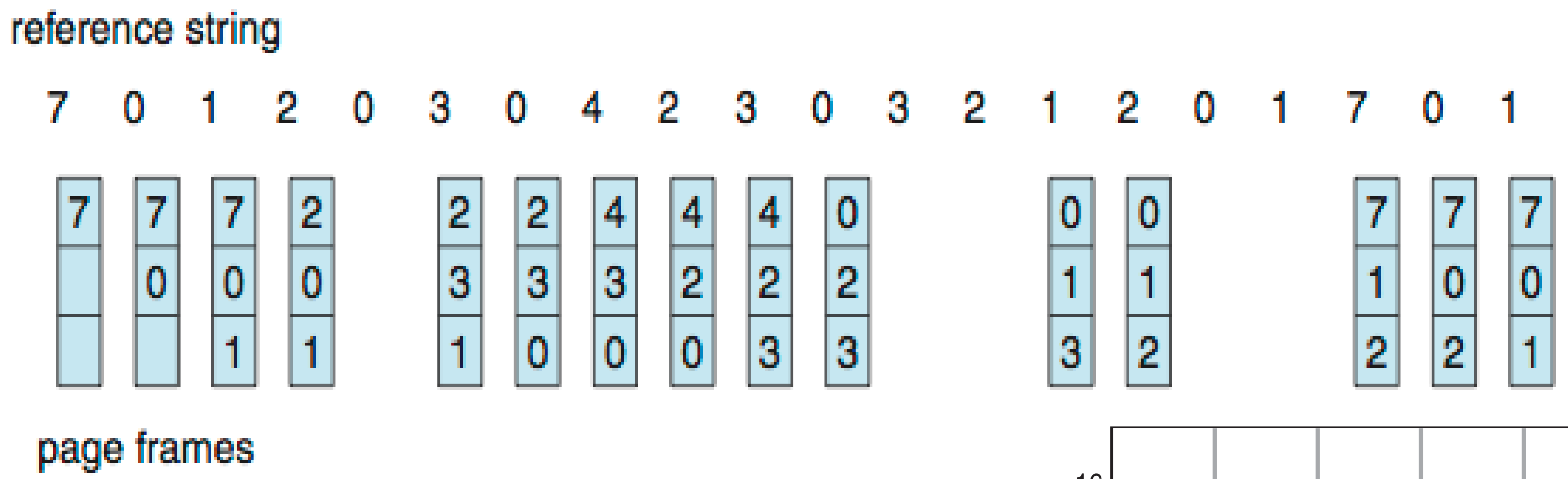
Page replacement



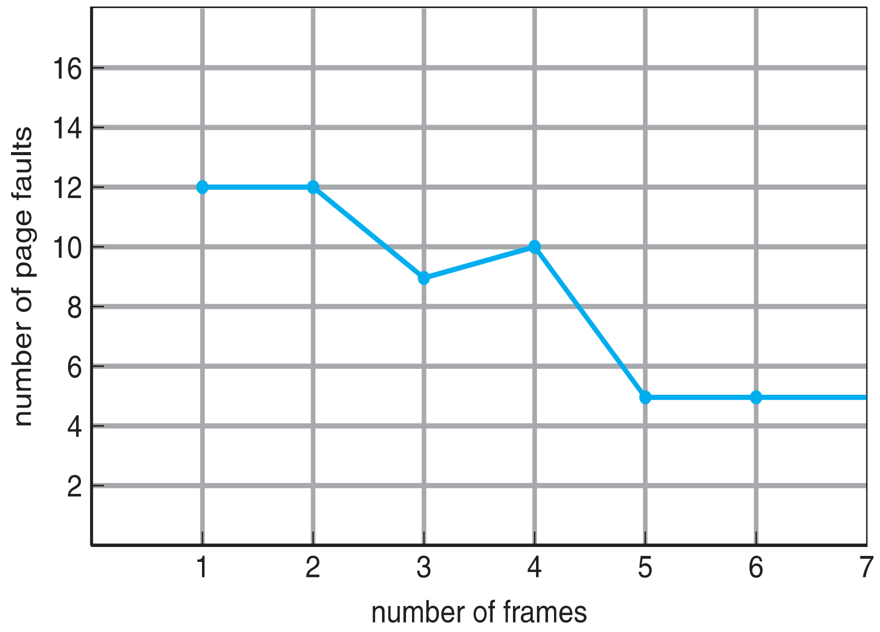
Page faults vs. # of frames



First-In-First-Out (FIFO) algorithm

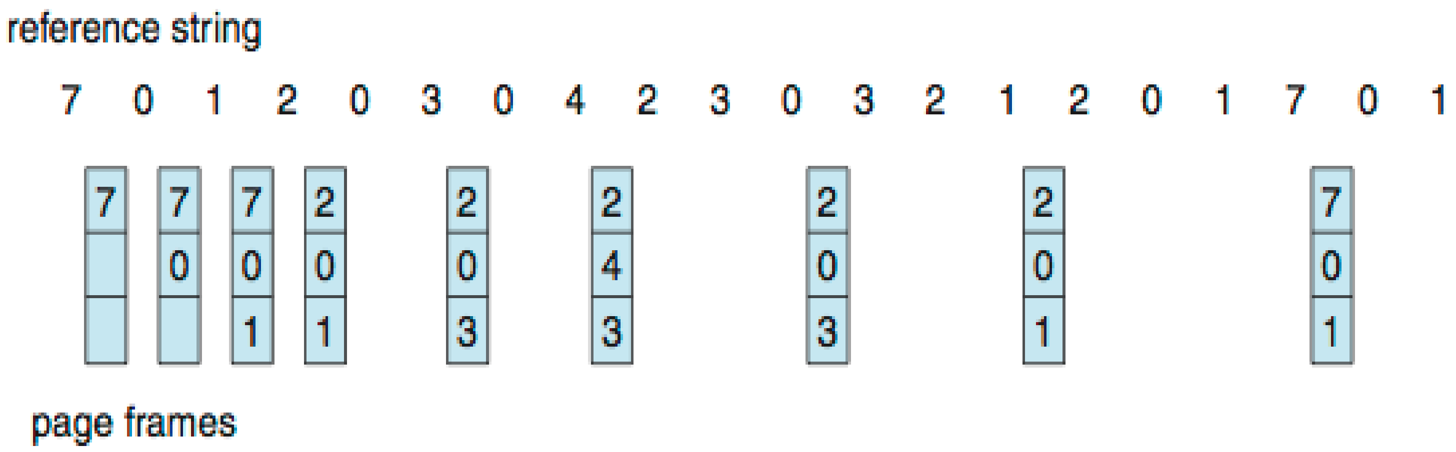


- **Problems**
  - Much miss rate (high page fault)
  - FIFO anomaly (Belady's anomaly)
    - Check 1,2,3,4,1,2,5,1,2,3,4,5
      - ✓ 8 page faults for 3-page mem!
      - ✓ 10 page faults for 4-page mem!



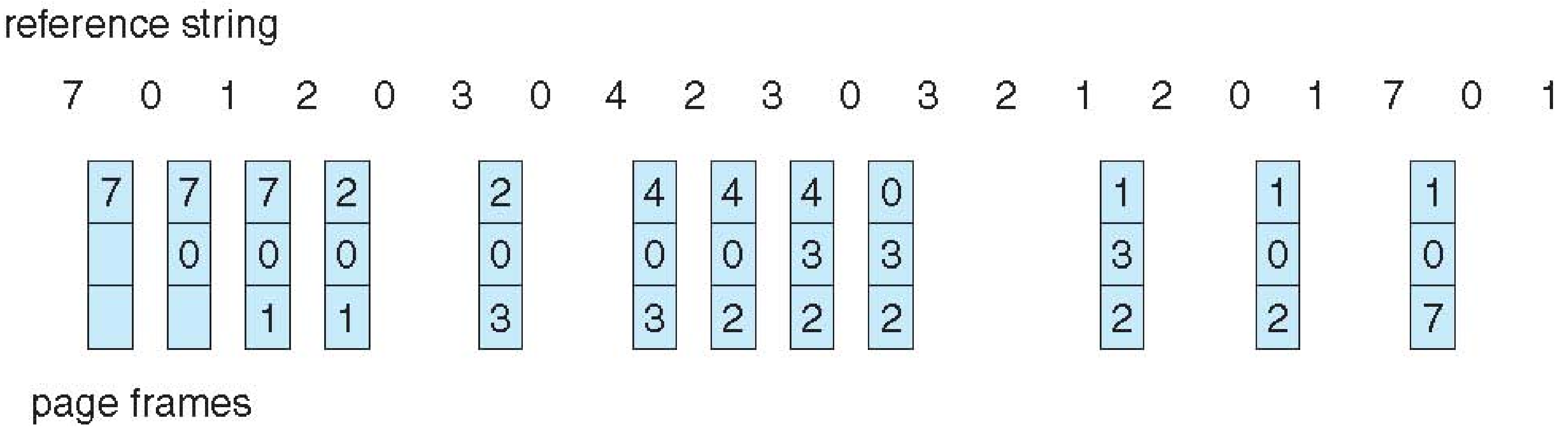
Optimal algorithm

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



- **Problems**
  - Who is aware of future?
- Used only for comparison other algorithms

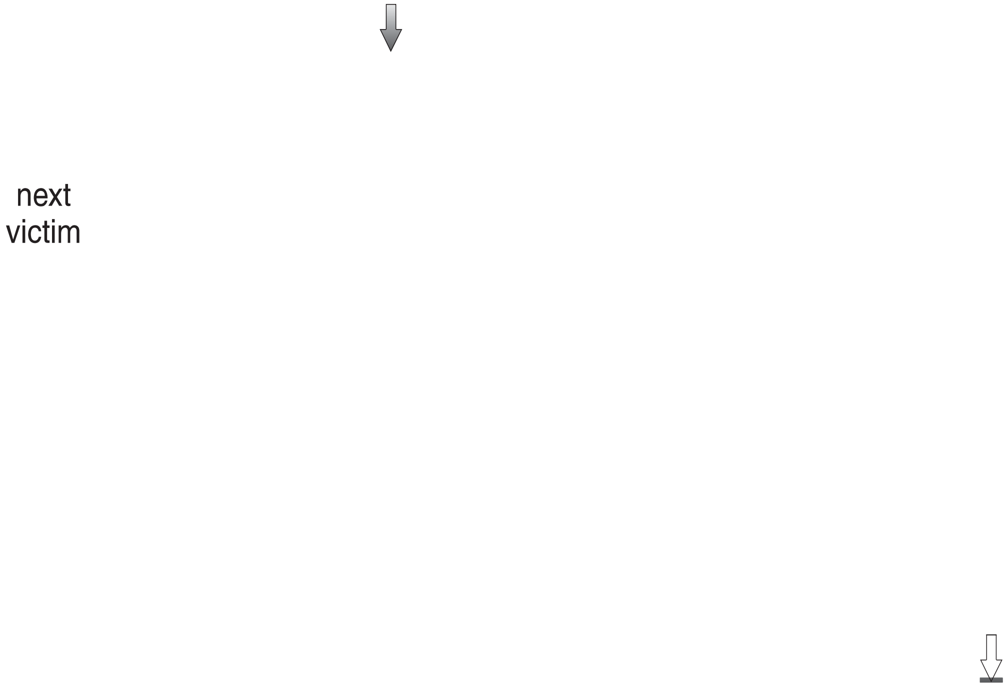
Least Recently Used (LRU)



- **Better** than **FIFO** but **worse** than **OPT**
- **Two possible implementations**
  - o **Counter-based**
  - o **Stack-based**

- **LRU and OPT are cases of stack algorithms that don't have Belady's Anomaly**

Second-Chance (clock) algorithm



circular queue of pages

(a)

LFU, MFU

- 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

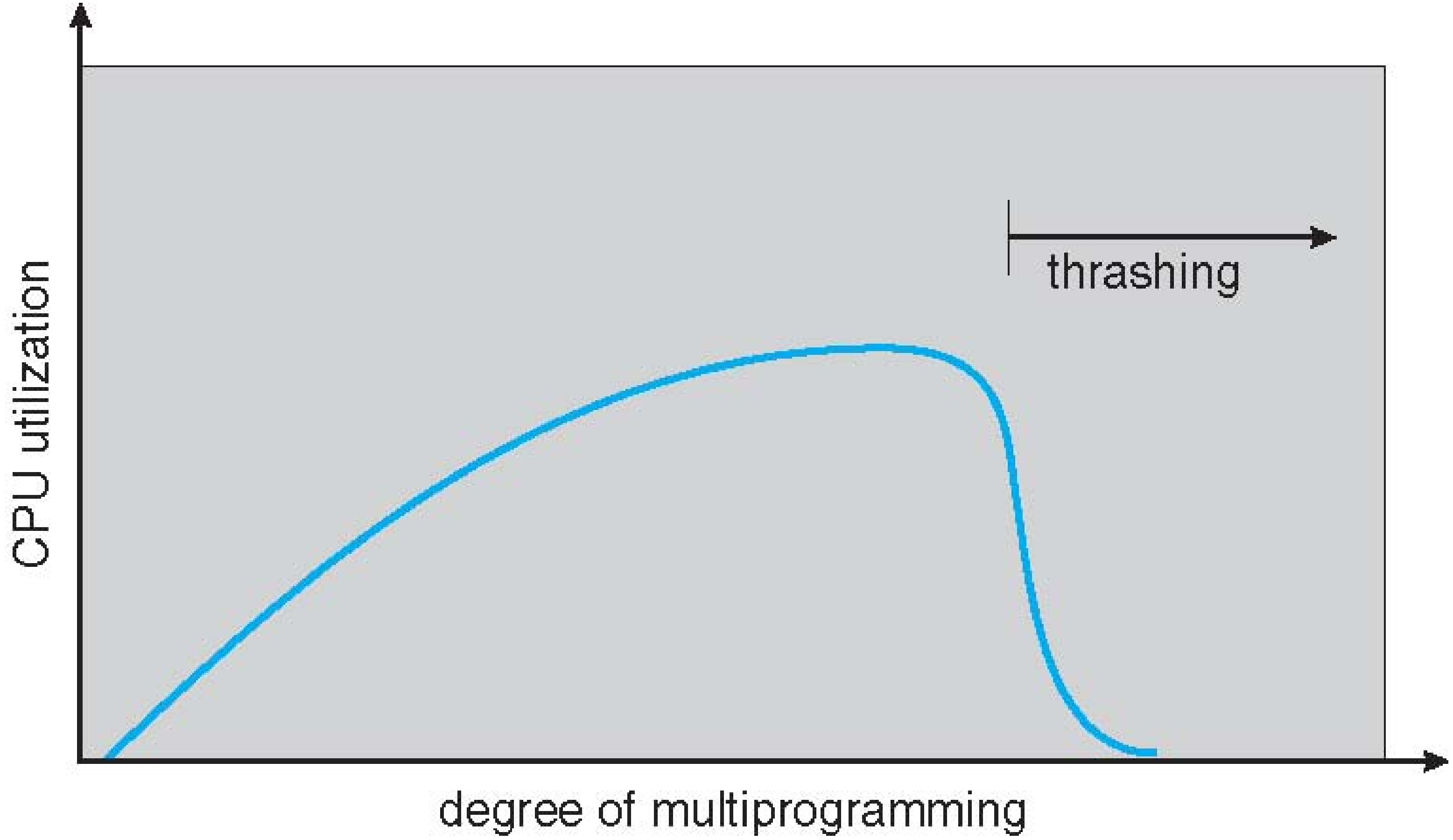
Global vs. Local allocation

- 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 (نامتعادل، کوبیدگی)



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

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?**

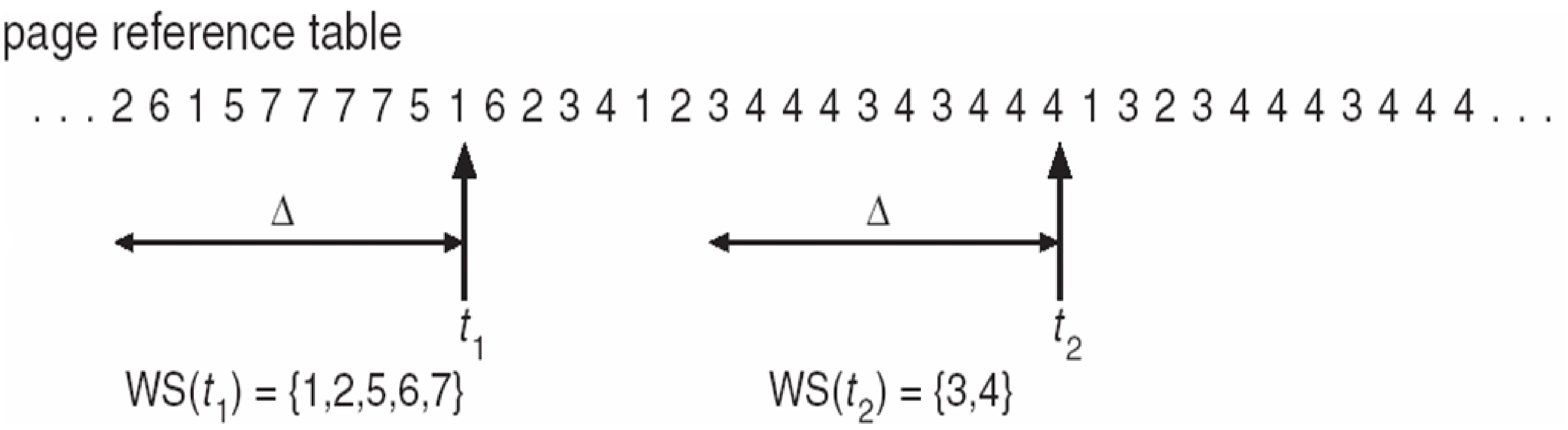
Σ size of locality > total memory size

- Limit effects by using local or priority page replacement

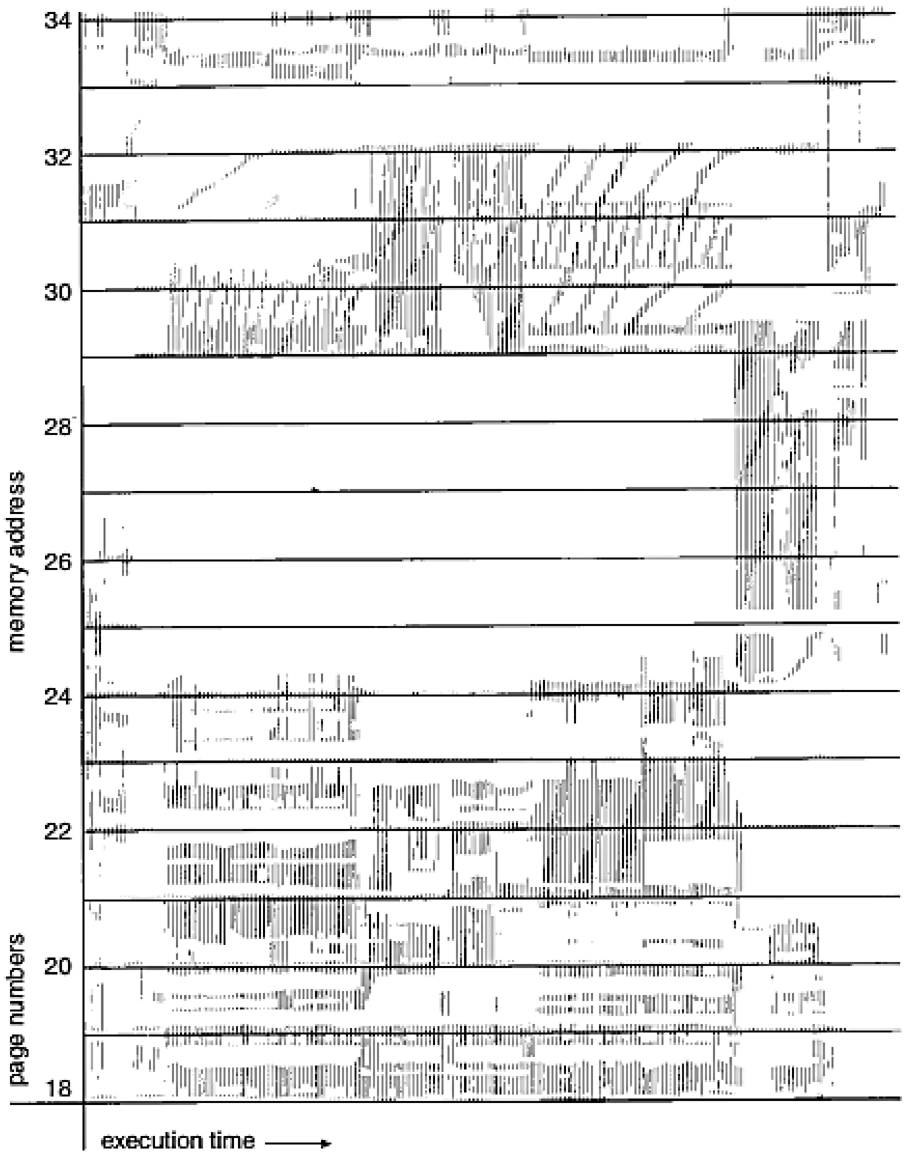
Solutions for Thrashing

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 of a program



Keeping track of the working set

- Approximate with interval timer + a reference bit

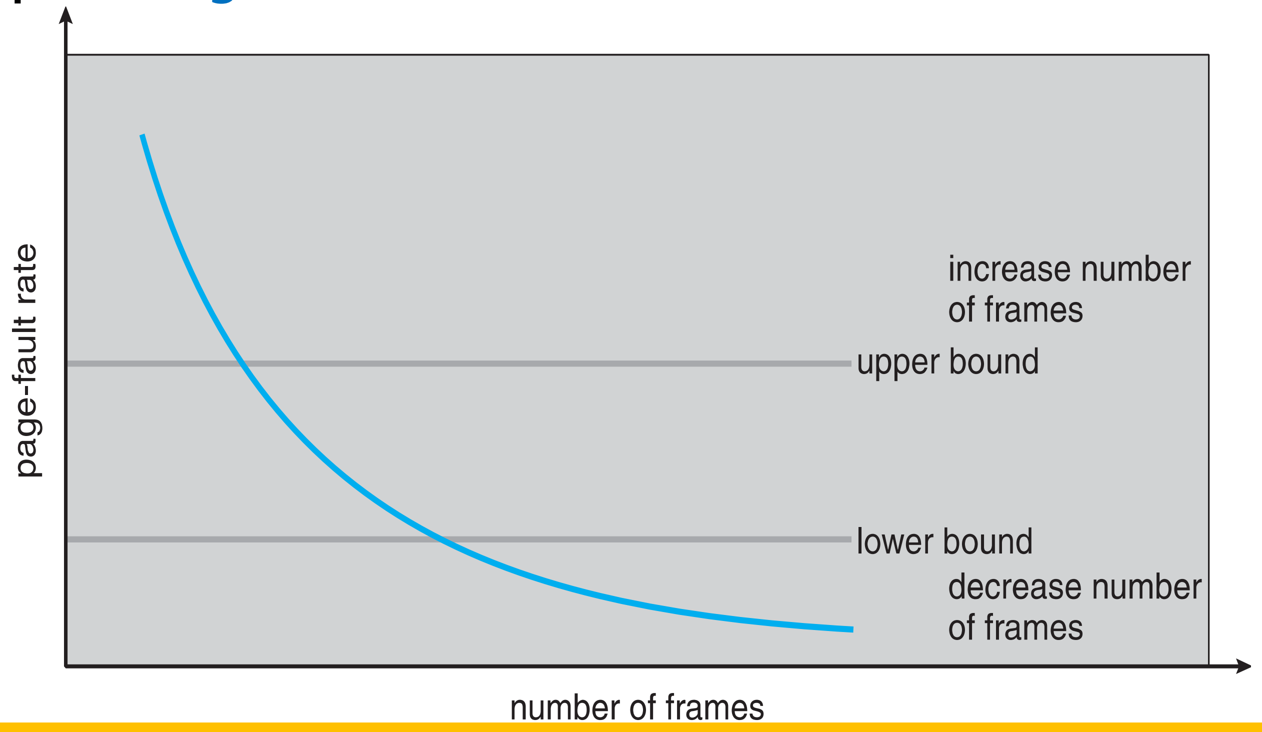
- Example: $\Delta = 10{,}000$
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set

- Why is this not completely accurate?

- Improvement = 10 bits and interrupt every 1000 time units

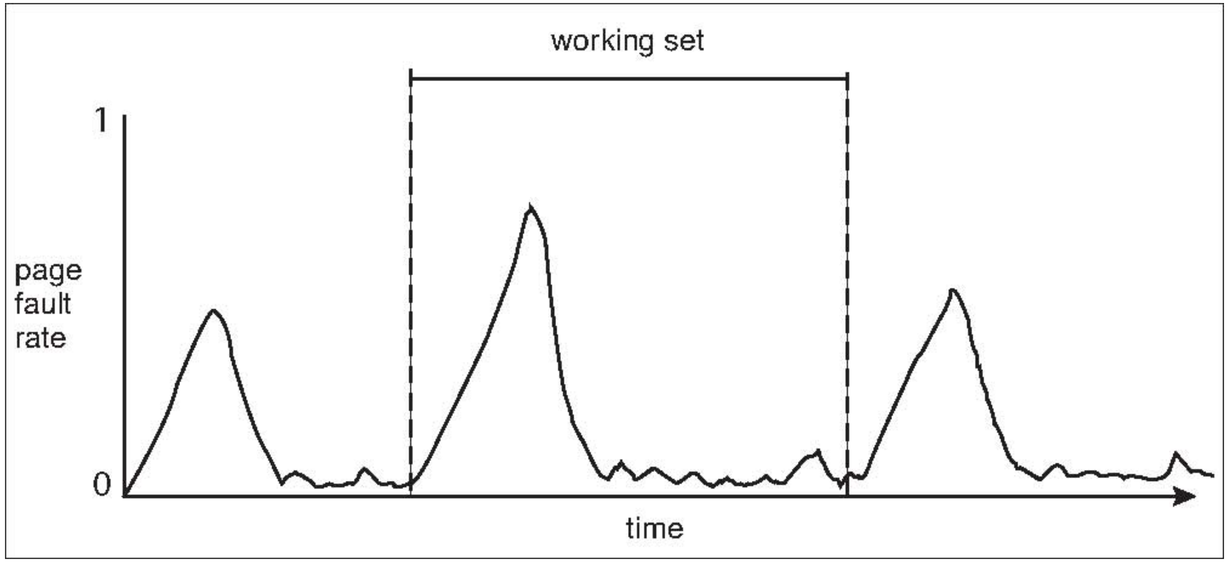
2) Page-fault frequency

- 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



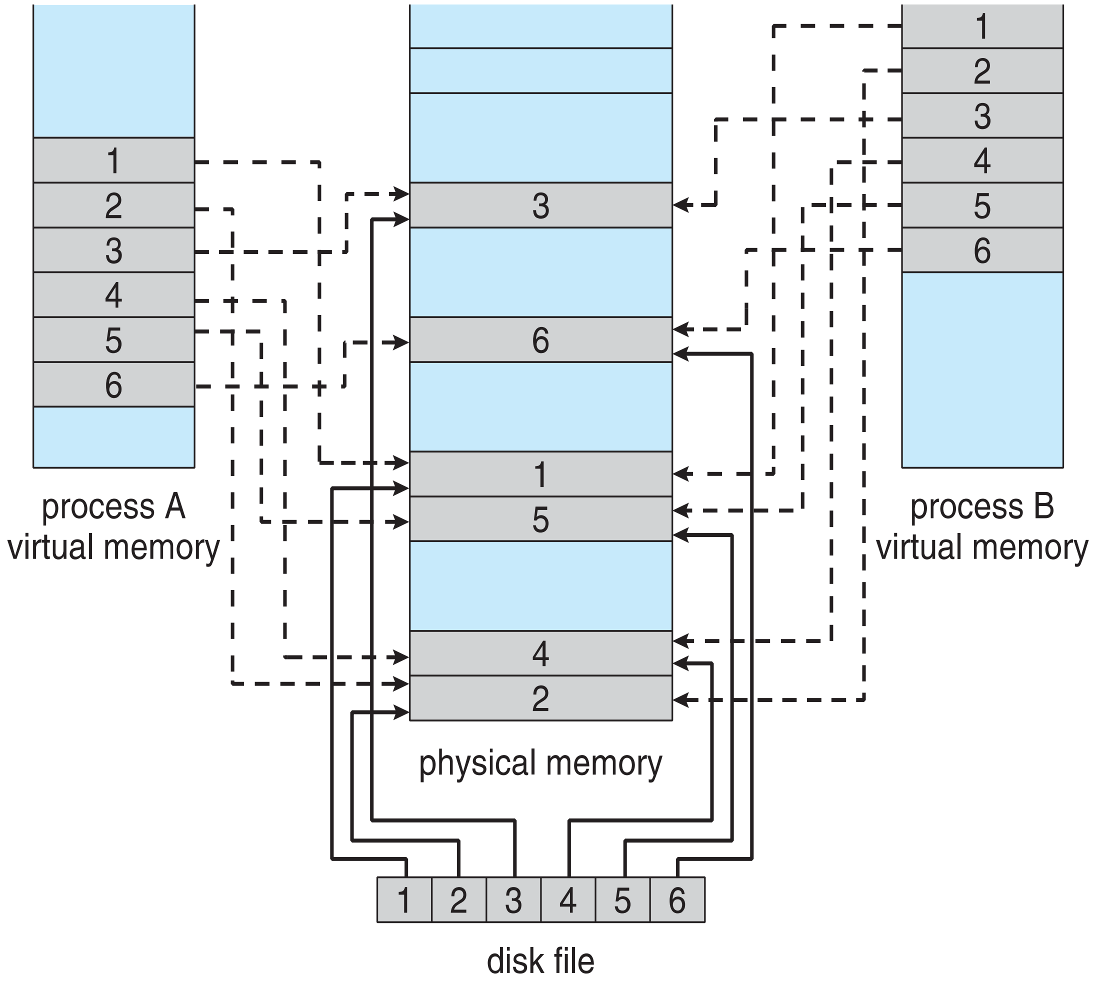
Memory mapped files, IOs

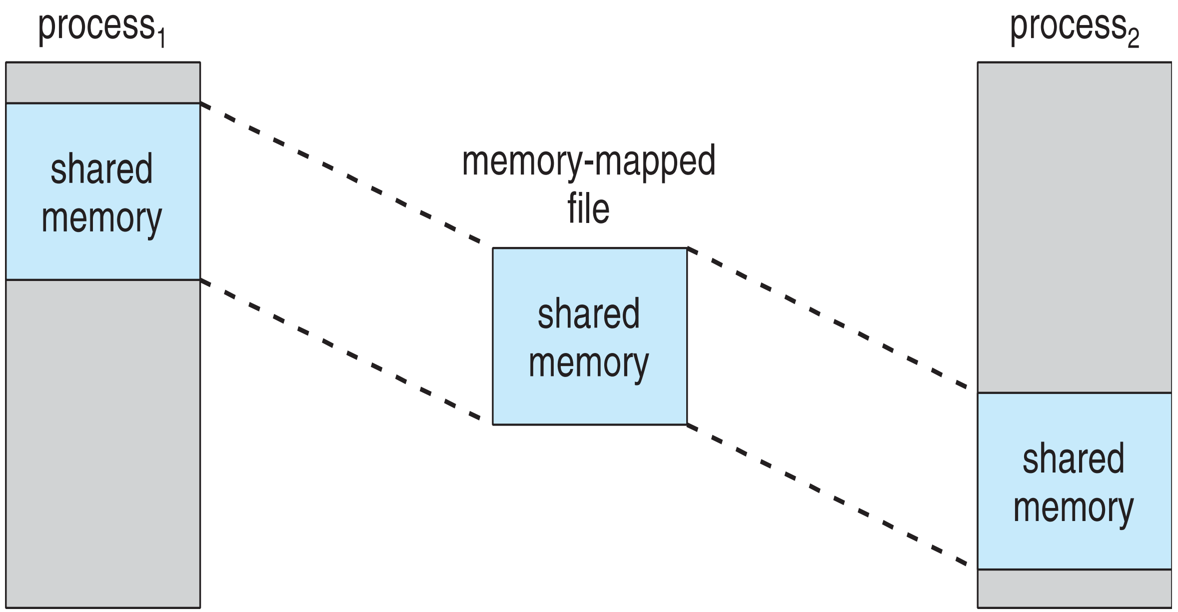
Memory-mapped files

- 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

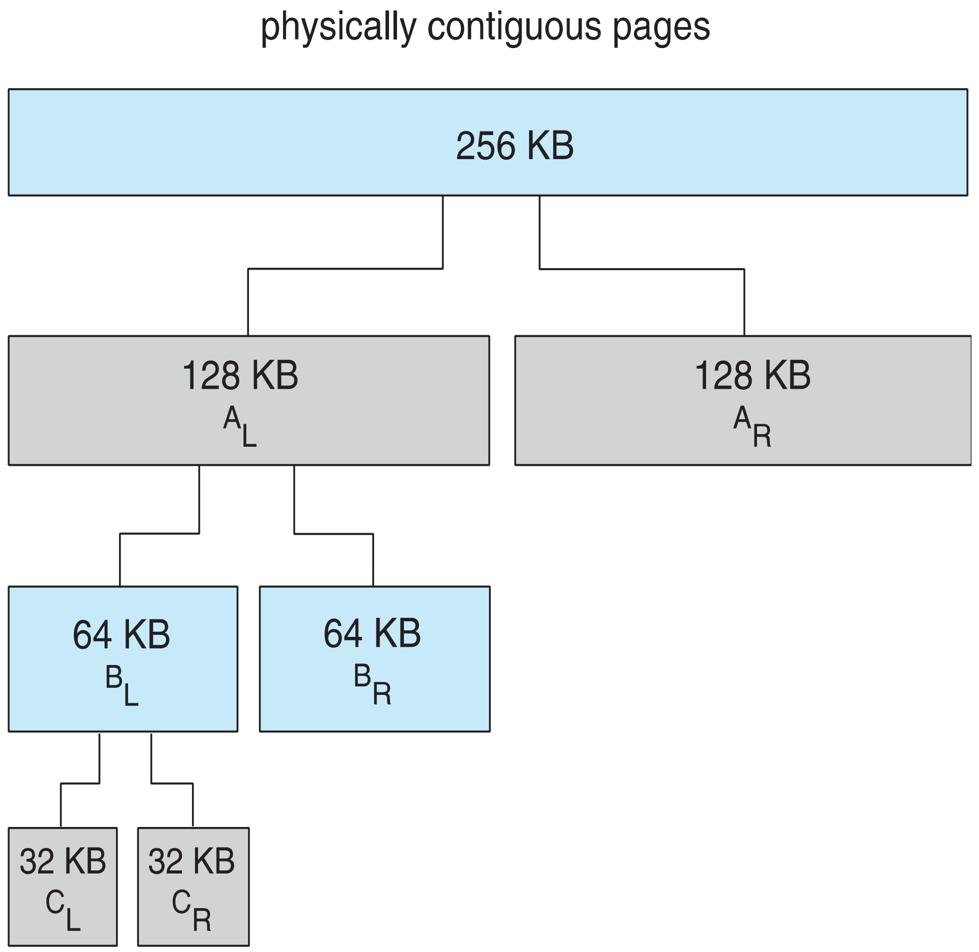
Some OSes uses memory mapped files for standard I/O

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
- For example, assume 256KB chunk available, kernel requests 21KB
  - Split into A_L and A_R of 128KB each
    - One further divided into B_L and B_R of 64KB
      - ✓ One further into C_L and C_R of 32KB each – one used to satisfy request
- Advantage – quickly coalesce unused chunks into larger chunk
- Disadvantage - fragmentation

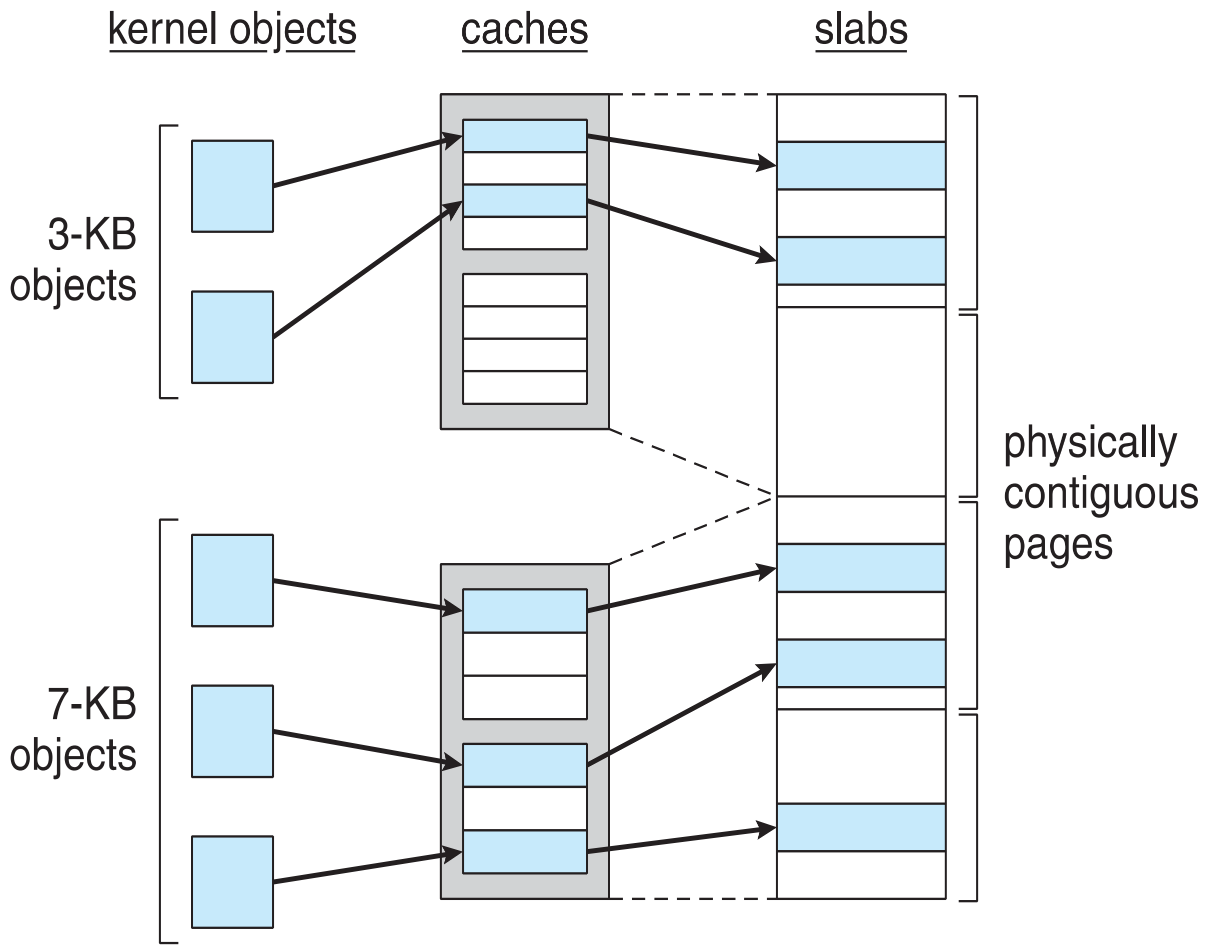
Buddy system allocator scheme



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
- Benefits include no fragmentation, fast memory request satisfaction

Slab allocation scheme



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 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
  - Resolution
  - 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

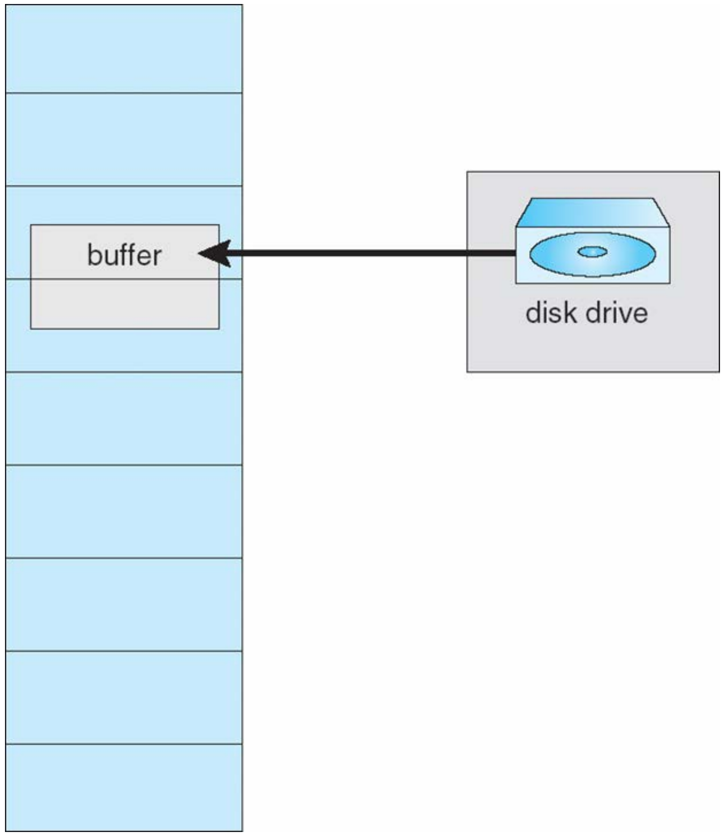
- Program 2

```
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



Questions?