Chapter: Virtual Memory

- Background
- Demand Paging
- Process Creation
- Page Replacement
- Allocation of Frames
- Thrashing
- Operating System Examples

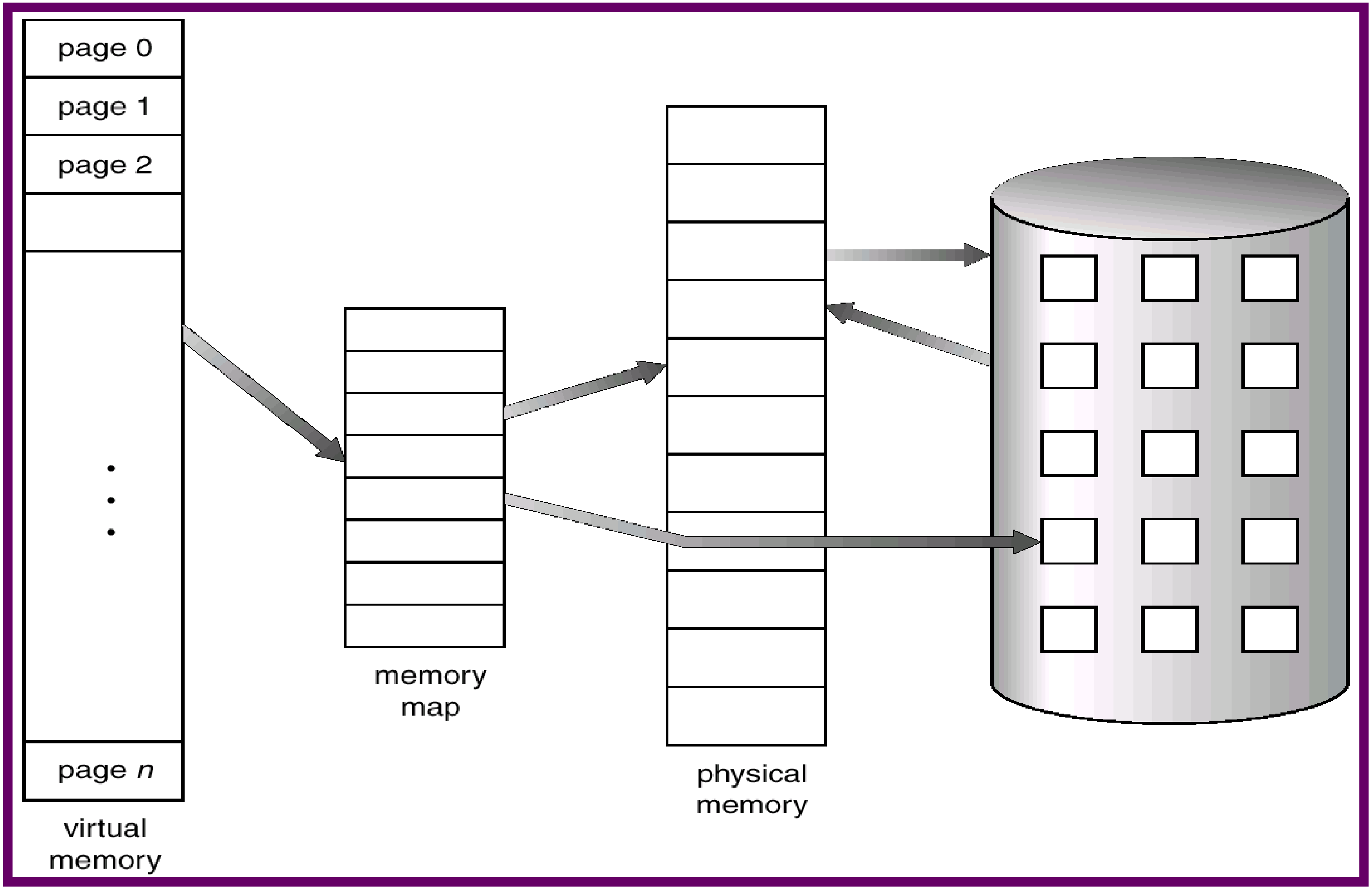
Background

- First requirement for execution: Instructions must be in physical memory
  - **One Approach**: Place entire logical address in main memory.
  - Overlays and dynamic loading may relax this criteria.
  - But the size of **the program is limited to size of main memory.**
- Normally entire program may not be needed in main memory.
  - Programs have error conditions.
  - Arrays, lists, and tables may be declared by 100 by 100 elements, but seldom larger than 10 by 10 elements.
  - Assembler program may have room for 3000 symbols, although average program may contain less than 200 symbols.
  - Certain portions or features of the program are used rarely.
- Benefits of the ability to execute program that is partially in memory:
  - User can write programs and software for entirely large virtual address space.
  - More programs can run at the same time.
  - Less I/O would be needed to load or swap each user program into memory.

Background

- Virtual memory is a technique that allows the execution of processes that may not be completely in memory.
  - Programs are larger than main memory.
  - VM abstract main memory into an extremely large, uniform array of storage.
- 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.
  - Frees the programmer from memory constraints.
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation
- We only cover demand paging.
- For demand segmentation refer research papers.
  - IBM OS/2, Burroughs' computer systems

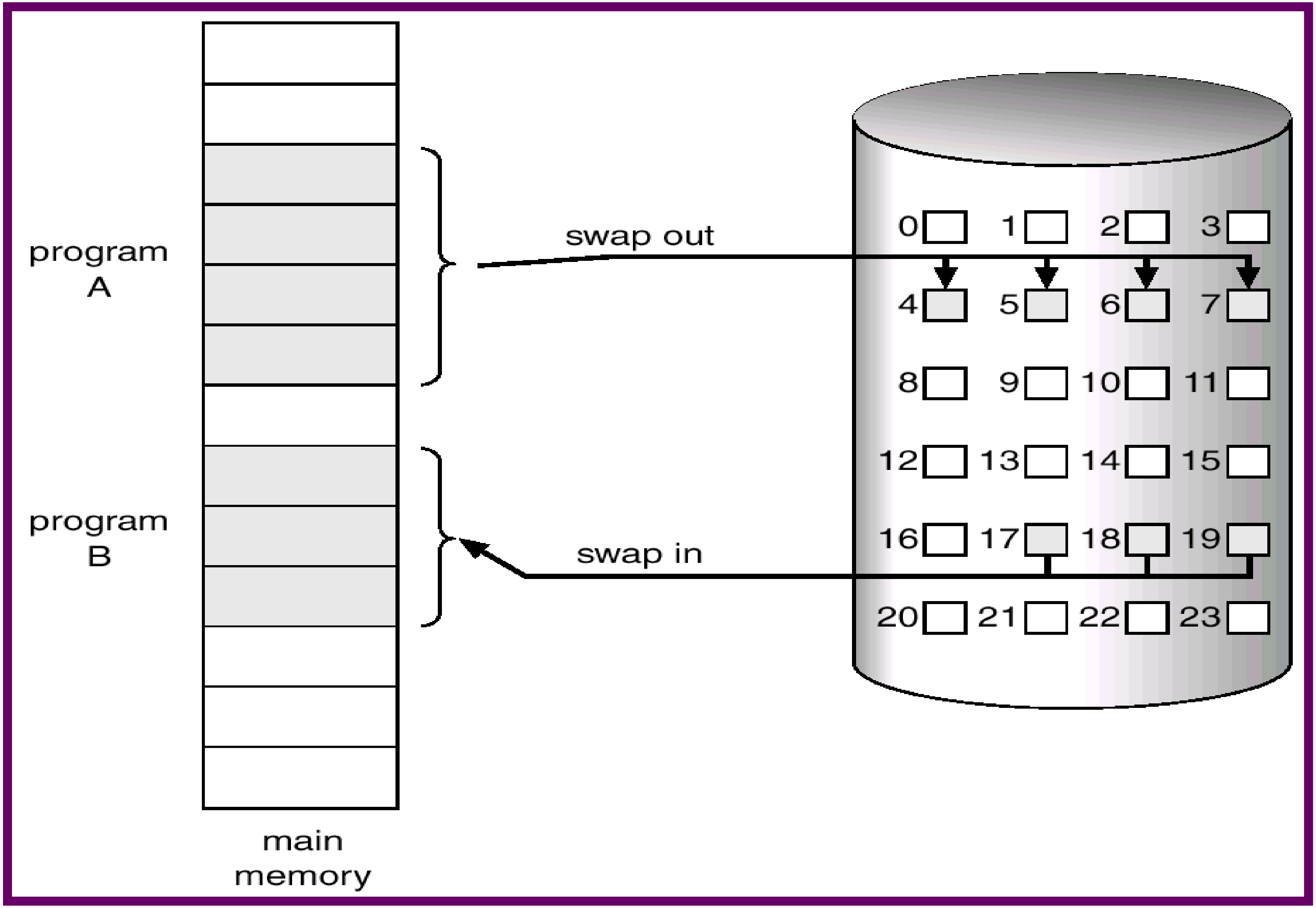
Virtual Memory That is Larger Than Physical Memory



Demand Paging

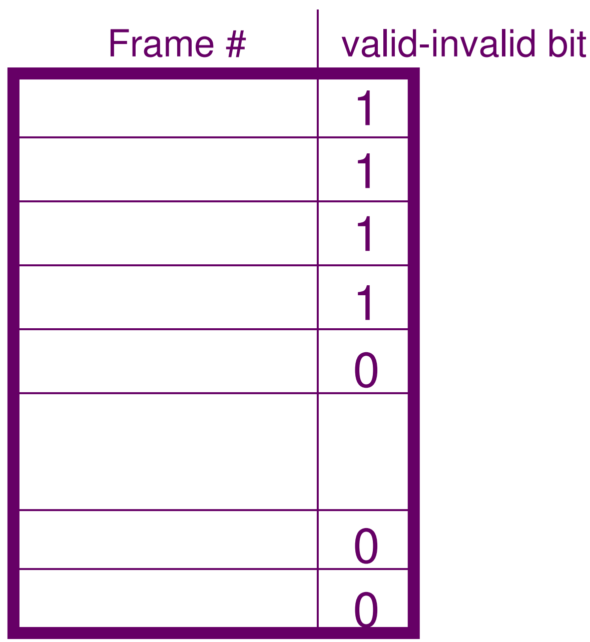
- Paging system with swapping.
  - When we execute a process we swap into memory (next fig).
- For demand paging, we use lazy swapper or pager.
  - Never swaps a page into memory unless required.
  - Bring a page into memory only when it is needed.
    - Less I/O needed
    - Less memory needed
    - Faster response
    - More users
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory

Transfer of a Paged Memory to Contiguous Disk Space



Valid-Invalid Bit

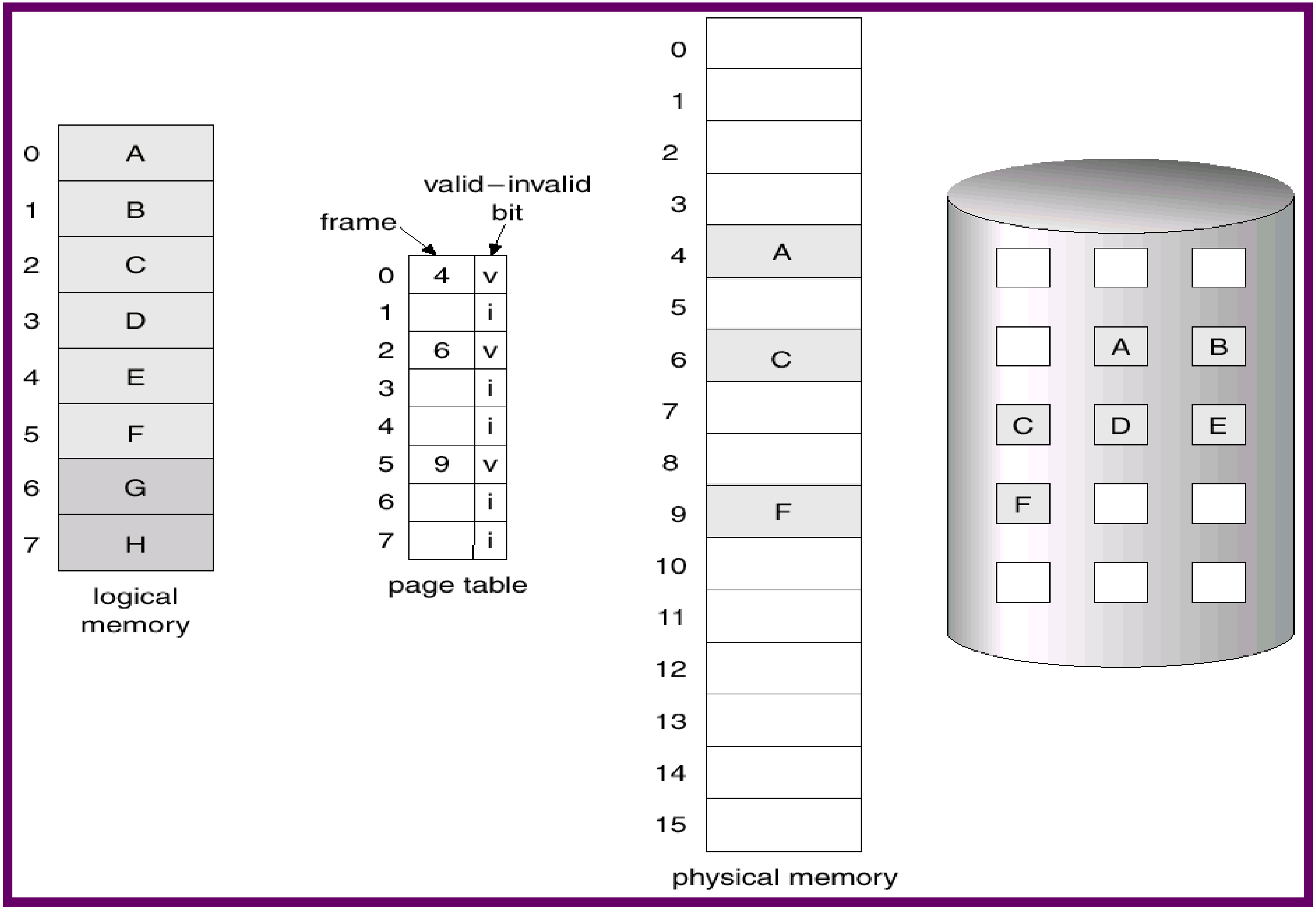
- With each page table entry a valid–invalid bit is associated ($1 \Rightarrow$ in-memory, $0 \Rightarrow$ not-in-memory)
- Initially valid–invalid but is set to 0 on all entries.
- Example of a page table snapshot.



page table

- During address translation, if valid–invalid bit in page table entry is $0 \Rightarrow$ page fault.

Page Table When Some Pages Are Not in Main Memory

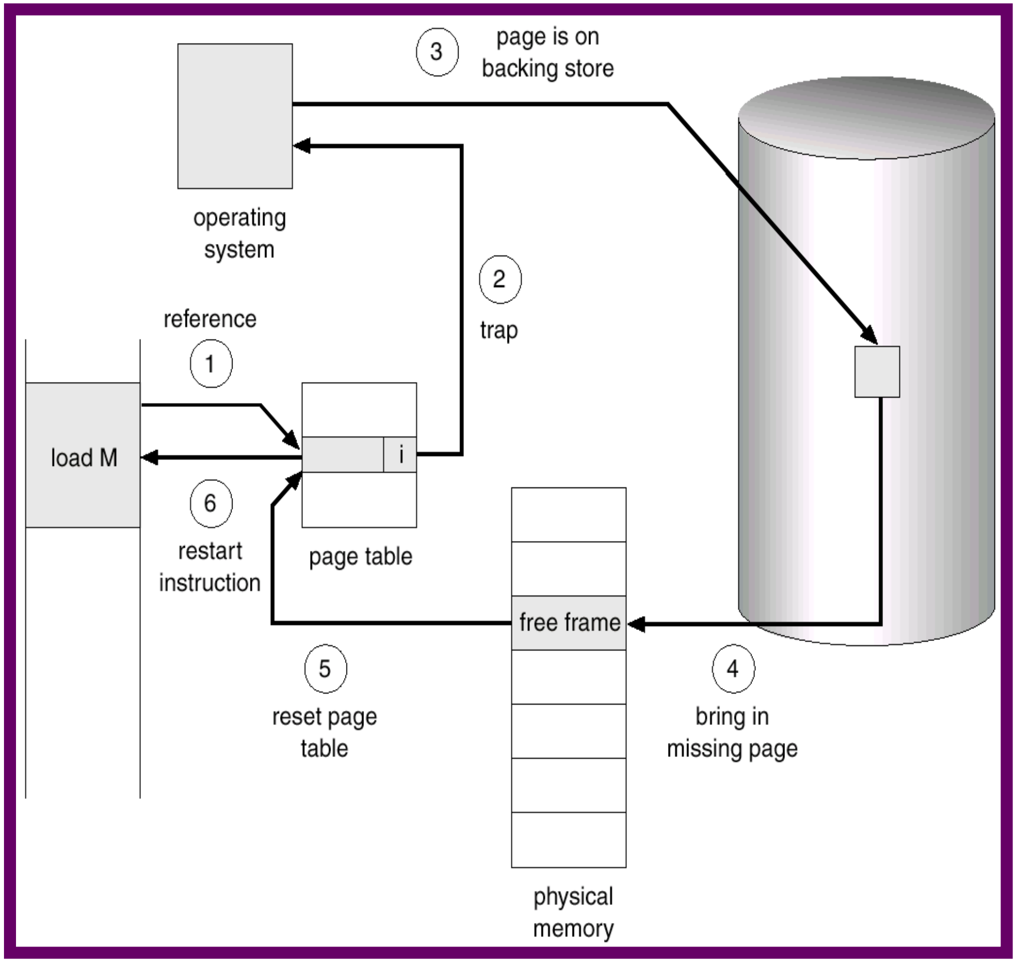


Page Fault

- If there is ever a reference to a page, first reference will trap to OS ⇒ page fault
- OS looks at another table to decide:
  - Invalid reference
    - ⇒ abort.
  - Just not in memory.
    - Get empty frame.
    - Swap page into frame.
    - Reset tables, validation bit = 1.
    - Restart instruction:

Steps in Handling a Page Fault

1. Check the internal table, to determine whether this reference is valid or invalid.
2. If the reference is invalid, then terminate.
3. Find free frame.
4. Schedule disk operation.
5. Modify the internal table to indicate that page is in main memory.
6. Restart the instruction.



What happens if there is no free frame?

- Page replacement – find some page in memory, but not really in use, swap it out.
  - algorithm
  - performance – want an algorithm which will result in minimum number of page faults.

- Same page may be brought into memory several times.

Hardware support to demand paging

- Theoretically some programs access several pages of new memory causing multiple page faults per instruction.
  - But analysis of program show locality of reference.
- Hardware support to demand paging
  - Page table
  - Secondary memory; high speed disk

Software support to demand paging

- Additional software is also required.
  - Restarting of instruction after page fault.
  - Page fault could occur any time during execution.
  - Ex: Add the contents of A and B and replace the result in C
    - 1.Fetch and decode the instruction
    - Fetch A
    - Fetch B
    - Add A and B
    - Store the sum in C.
  - If the page is faulted if we try to store C, we have to restart the instruction.

Software support to demand paging…

- Difficulty occurs if the instruction modifies several different locations.
- In IBM 360/370 MVC (Move character) instruction, we can move 256 bytes from one location to another.
  - source and destination may overlap
- If the page fault occurs after partial moving, we can not redo the instruction, if regions overlap.
- Solution:
  - 1. Use micro code to access both ends of blocks
    - ✔ If page fault is going to occur, it will occur.
  - 2. Use temporary registers to hold the values of temporary registers
    - ✔ If a page fault occurs old values are written back to memory, restoring the memory state to before the instruction was started.

Hardware support and software support to demand paging…

- Also, similar difficulty occurs in machines that use special addressing modes. Uses register as a pointer
  - Auto-increment: increment after using
  - auto-decrement. Decrements before using
  - MOV (R2)+, -(R3)
  - If the page fault occurs while storing in R3, we have to restart the instruction by restoring the values of R2 and R3.
  - if the instruction modifies several different locations.
- Solution: use status register to record the register number and amount modified so that OS can undo the effect of partially executed instruction that causes a page fault.
- **EVERY THING SHOULD BE TRANSAPARENT TO USER**

Performance of Demand Paging

- Let p be the probability of page fault.
- Page Fault Rate $0 \leq p \leq 1.0$
  - if $p = 0$ no page faults
    - effective access time=memory access time.
  - if $p = 1$, every reference is a fault

- Effective Access Time (EAT)

  EAT = $(1 – p)$ x memory access

  $+ p$ (page fault overhead)

- Major operations during page fault:
  - Trap to OS; save user registers and process state; issue a disk read; wait for interrupt from disk; wait for the CPU; restore the process status;

Demand Paging Example

- Memory access time = 100 nanoseconds
- Page fault service time = 25 milliseconds

- Effective access time (EAT) = (1 – p) x 100 + p (25 msec)
  - =100+24,999,900 x p
- EAT is directly proportional to page-fault rate.
- It is important to keep the page-fault rate low.
  - Otherwise EAT increases and slowing the process execution dramatically.

Advantages of VM: Process Creation

- Virtual memory allows other benefits during process creation:

  - Copy-on-Write

  - Memory-Mapped Files

Copy-on-Write

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

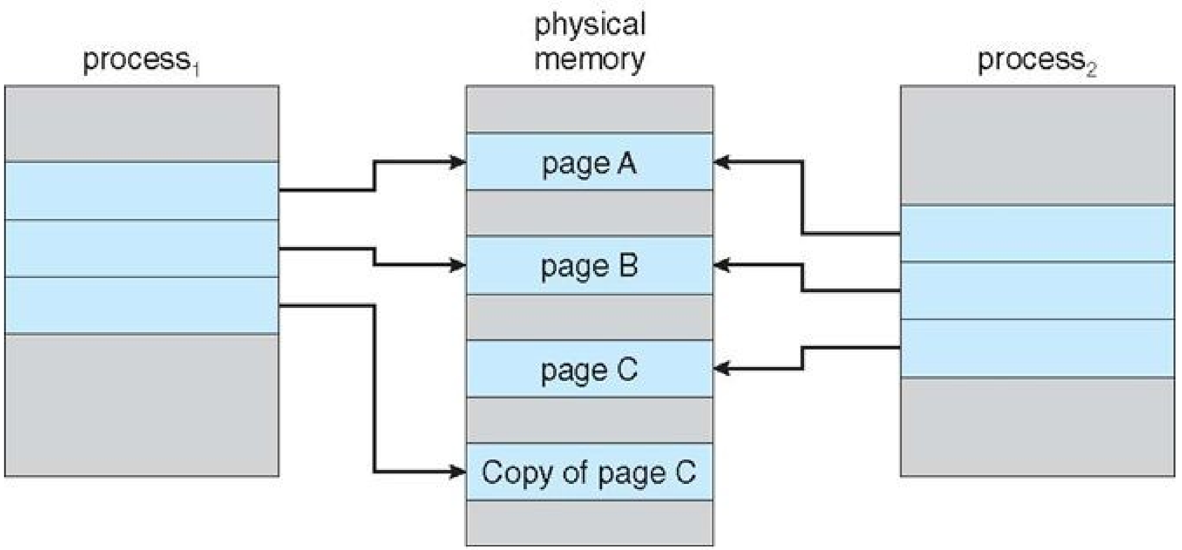
  If either process modifies a shared page, only then the page copied.

- COW allows more efficient process creation as only modified pages are copied.

- Free pages are allocated from a *pool* of zeroed-out pages.

Copy on Write

- *Recall*: the `fork()` system call creates a child process that is a duplicate of its parent
- Since the child might not modify its parents pages, we can employ the copy-on-write technique:
  - The child initially shares all pages with the parent.
  - If either process modifies a page, then a copy of that page is created.



Memory-Mapped Files

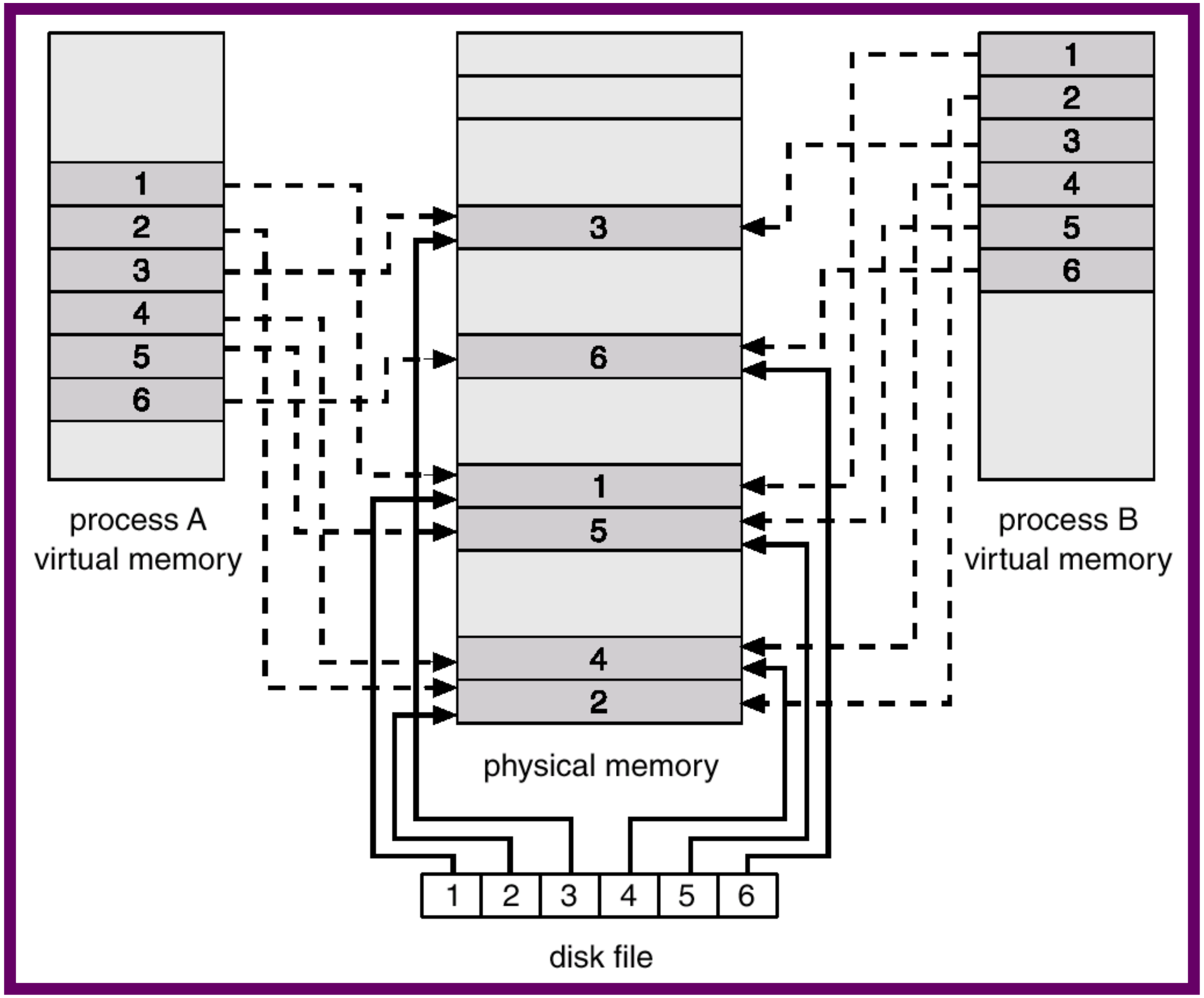
- Memory-mapped file I/O allows file I/O to be treated as routine memory access by *mapping* a disk block to a page in memory.

- 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.

- Simplifies file access by treating file I/O through memory rather than **read() write()** system calls.

- Also allows several processes to map the same file allowing the pages in memory to be shared.

Memory Mapped Files



Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement.

Basic Page Replacement

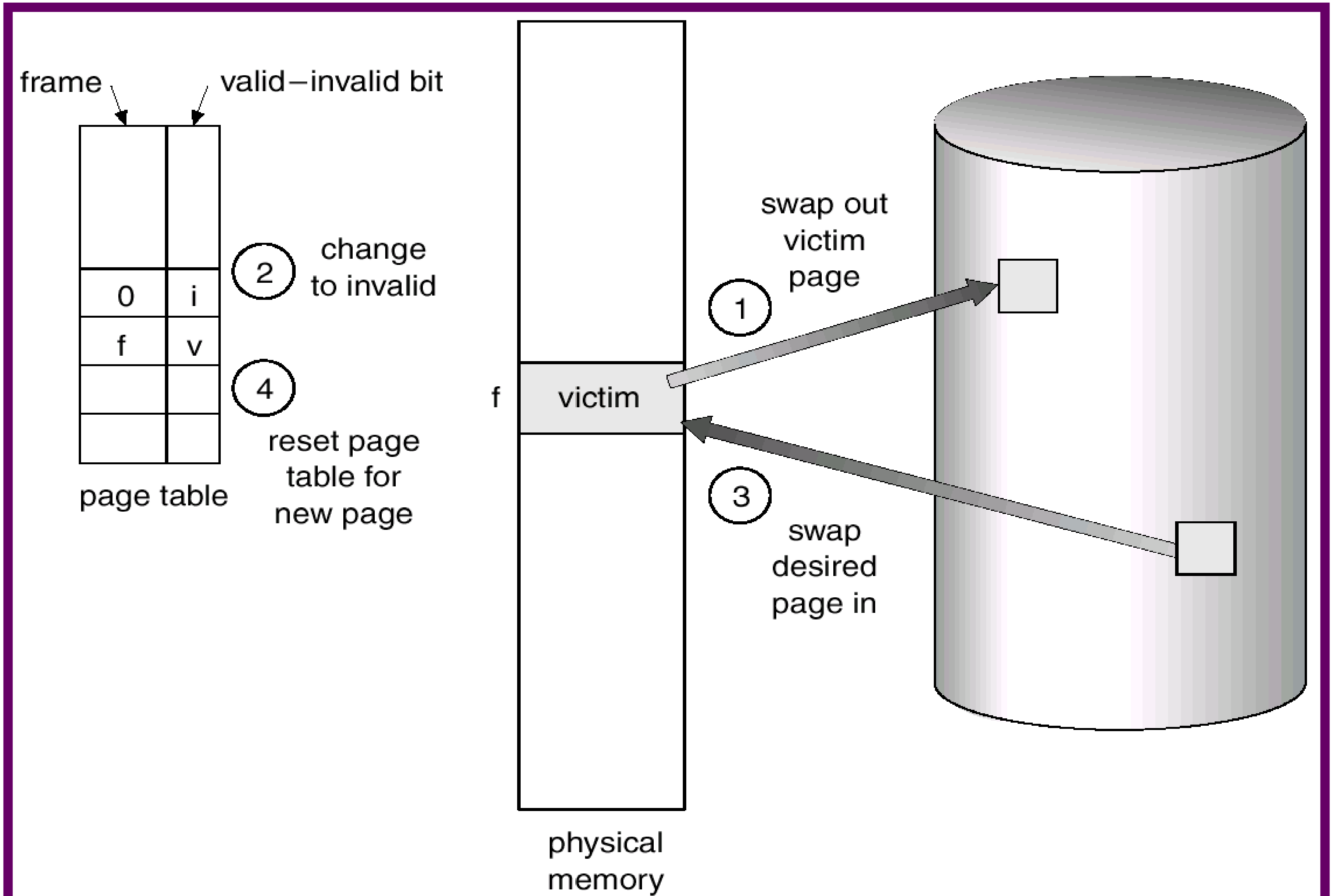
1. Find the location of the desired page on disk.

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.

3. Read the desired page into the (newly) free frame. Update the page table.

4. Restart the process.

Page Replacement



Reducing overhead: modify bit

- Use *modify* (*dirty*) *bit* to reduce overhead of page transfers – only modified pages are written to disk.

- Modify bit can be used to reduce the overhead with the help of hardware.
- It is set indicating the page has been modified.
- While replacing a page
  - If the modify bit is set, we must write that page to disk.
  - If it is not set we can avoid overwriting it, if is not overwritten.

Page Replacement Algorithms

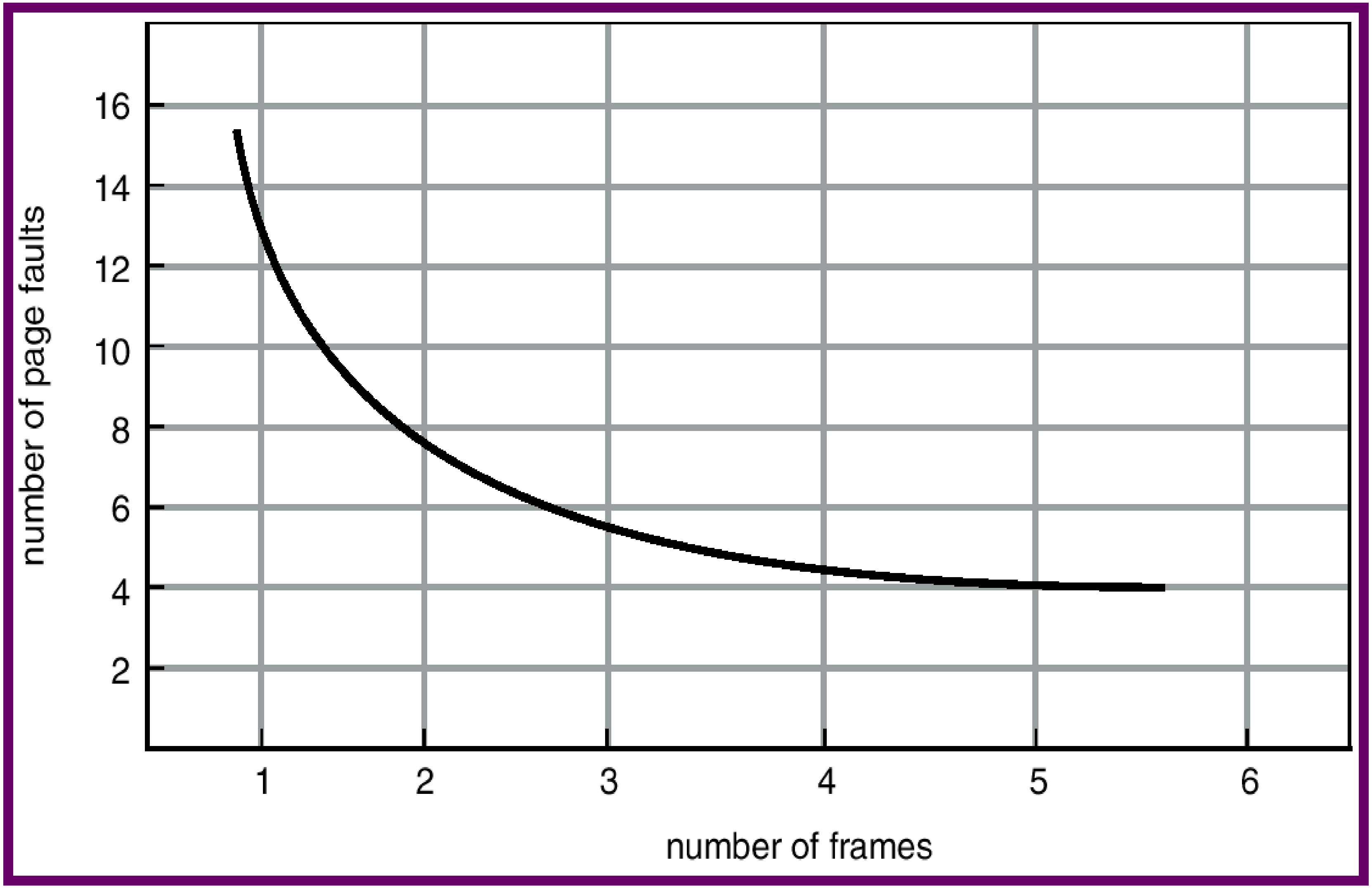
- Page replacement completes the separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory.
  - Enormous virtual memory can be provided on a smaller physical memory.
- Two major problems are solved to implement demand paging
  - **Frame allocation algorithm**
    - ✔ If multiple processes exist in memory we have to decide the number frames for each process
  - **Page replacement algorithm**
    - ✔ We have to select a frame that is to be replaced.

Page Replacement Algorithms

- Want lowest page-fault rate.
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string.
- In all our examples, the reference string is

  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.
- Address reference divided by page size.
  - If the address reference is 0432 and page size is 100 then the reference number is 0432/100=4

Graph of Page Faults Versus The Number of Frames



First-In-First-Out (FIFO) Algorithm

- Oldest page is replaced; FIFO queue; replace the head of the queue
  - Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)

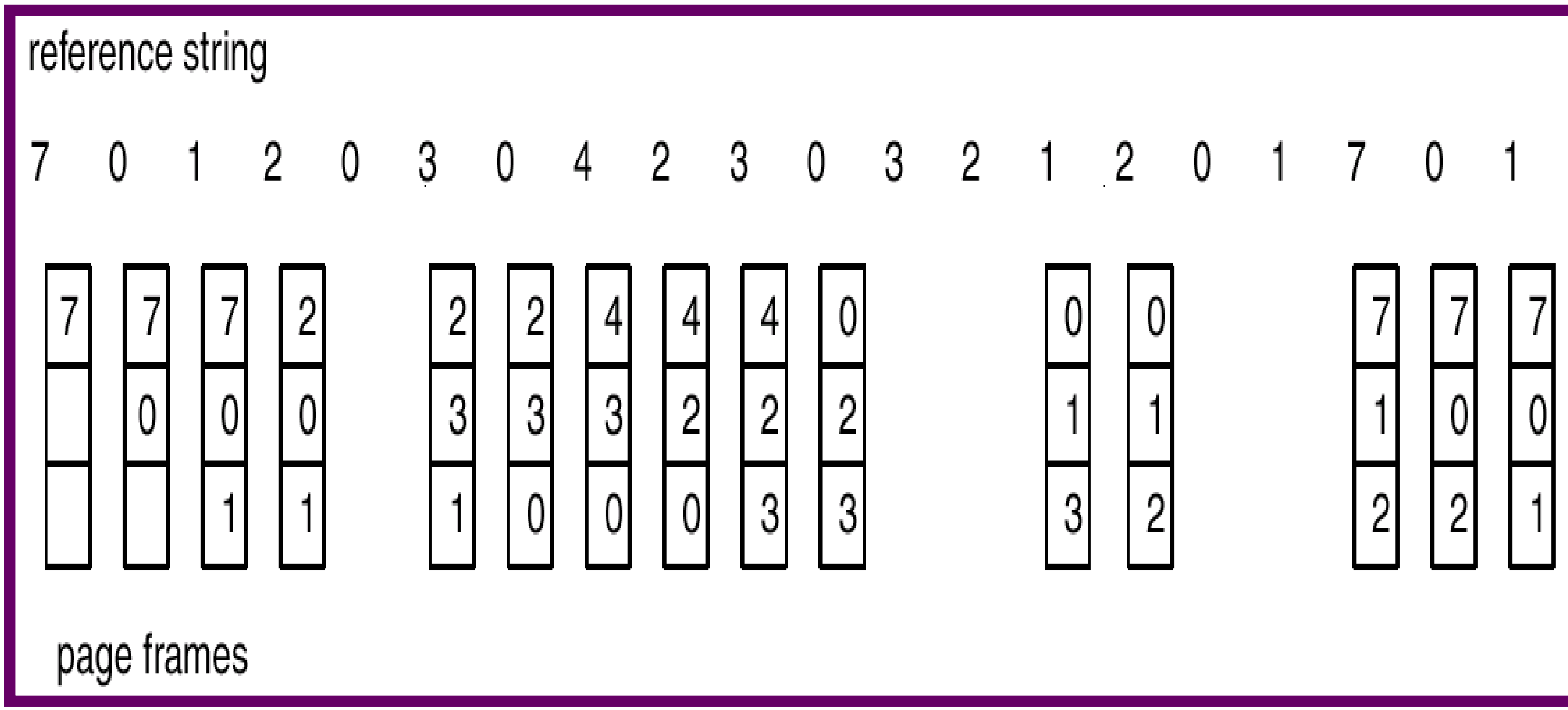
1	1	4	5	
2	2	1	3	9 page faults
3	3	2	4	

- 4 frames

1	1	5	4	
2	2	1	5	10 page faults
3	3	2		
4	4	3		

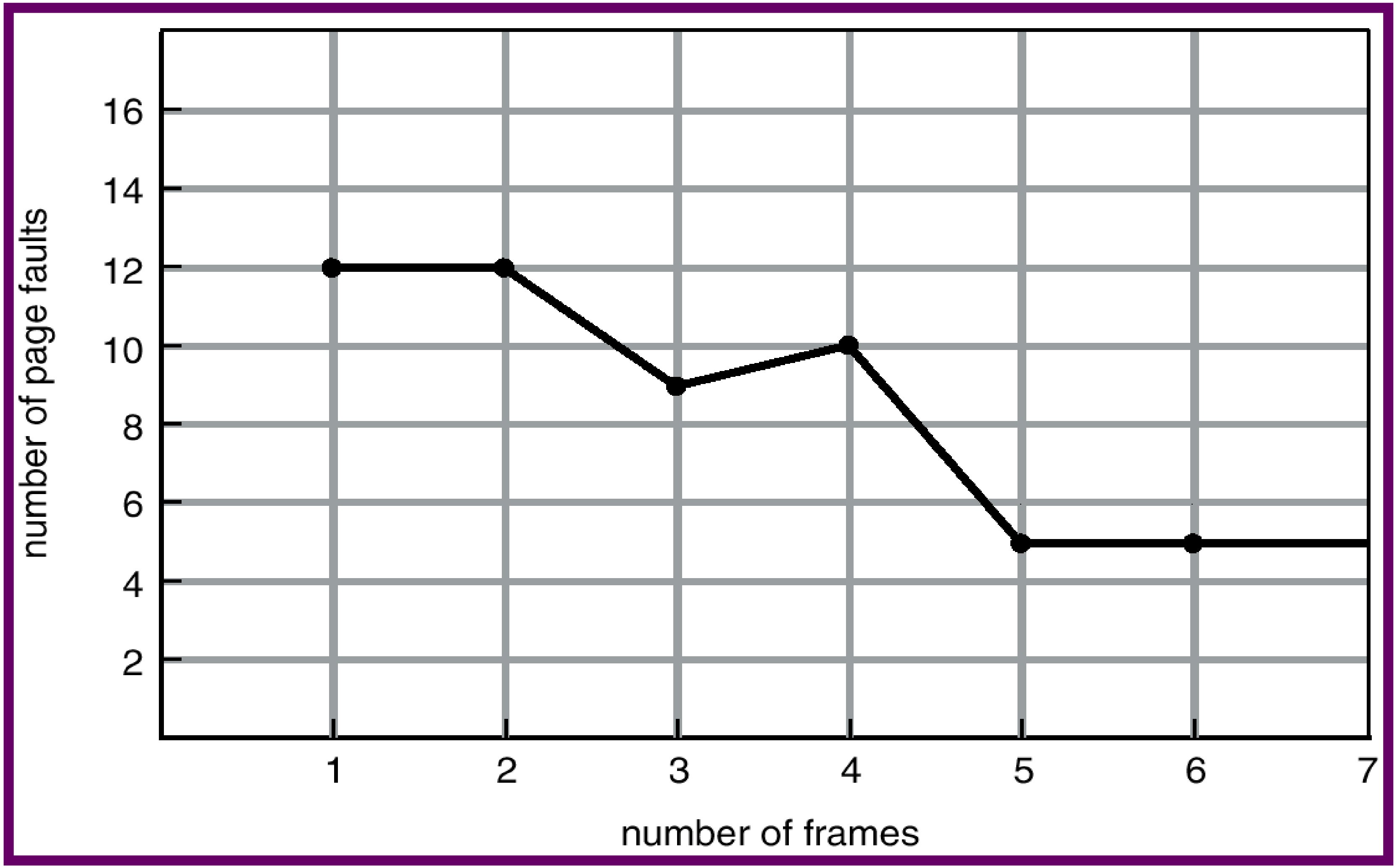
- FIFO Replacement – Belady’s Anomaly
  - more frames ⇒ more page faults

FIFO Page Replacement



FIFO Illustrating Belady's Anamoly

For some algs, page fault rate increases with number of pages.



Optimal Algorithm

- Replace page that will not be used for longest period of time.
- 4 frames example

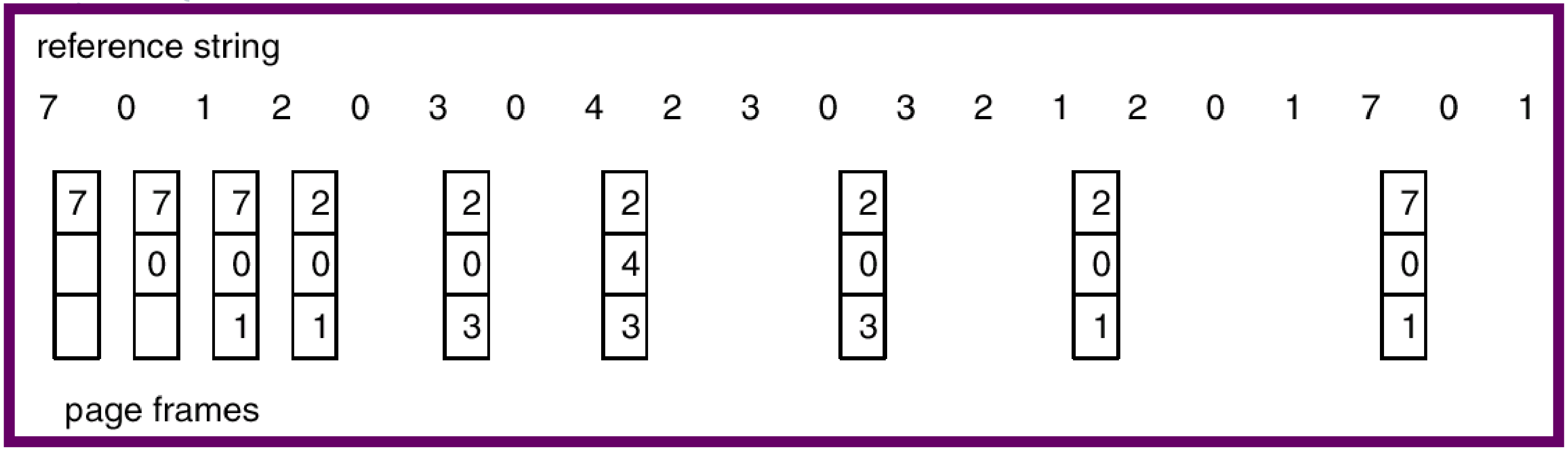
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

1	4
2	
3	
4	5

6 page faults

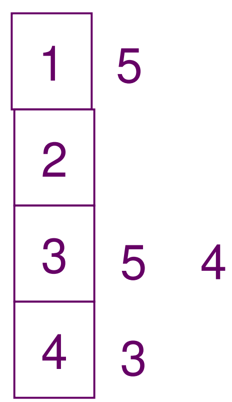
- How do you know this?
- Used for measuring how well your algorithm performs.

Optimal Page Replacement

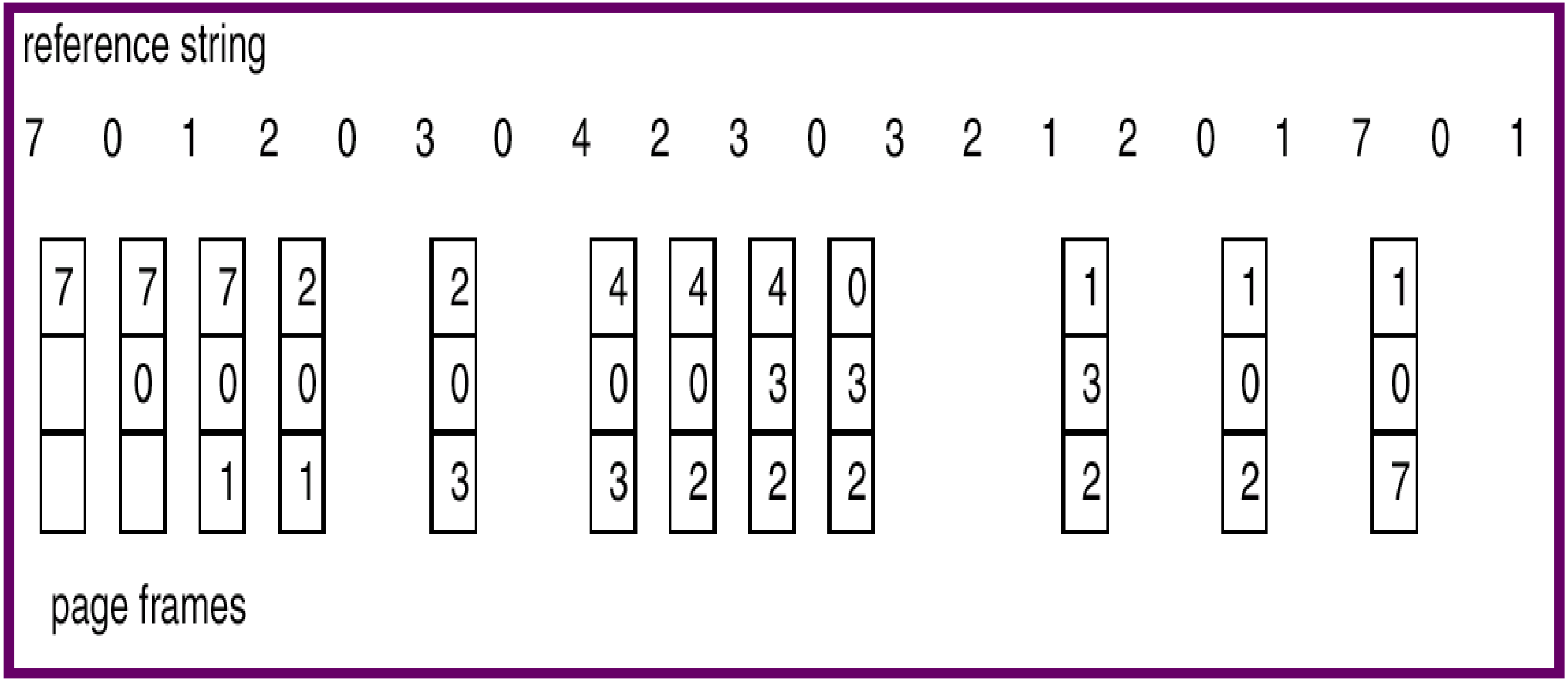


Least Recently Used (LRU) Algorithm

- The page that has not been used for longest period of time is replaced.
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



LRU Page Replacement

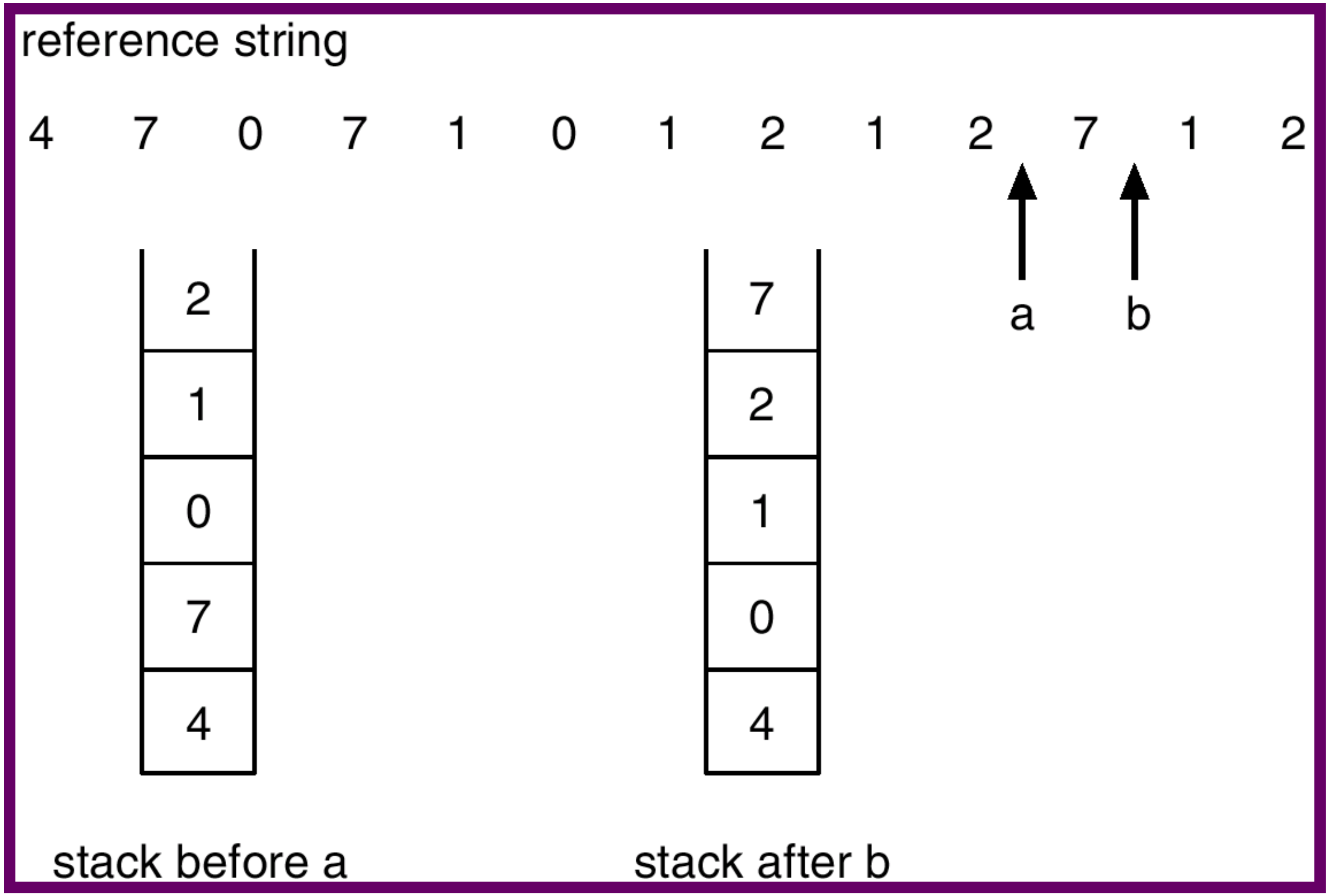


- The performance is good. But, How to Implement ?

LRU Algorithm Implementation

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter.
  - When a page needs to be changed, look at the counters to determine which are to change.
  - Issues:
    - Search pf page table to find LRU page, Overflow of clock,..
- Stack implementation – keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
    - Update is expensive
  - No search for replacement
  - Top is the most recently used page and bottom is the LRU page.

Use Of A Stack to Record The Most Recent Page References



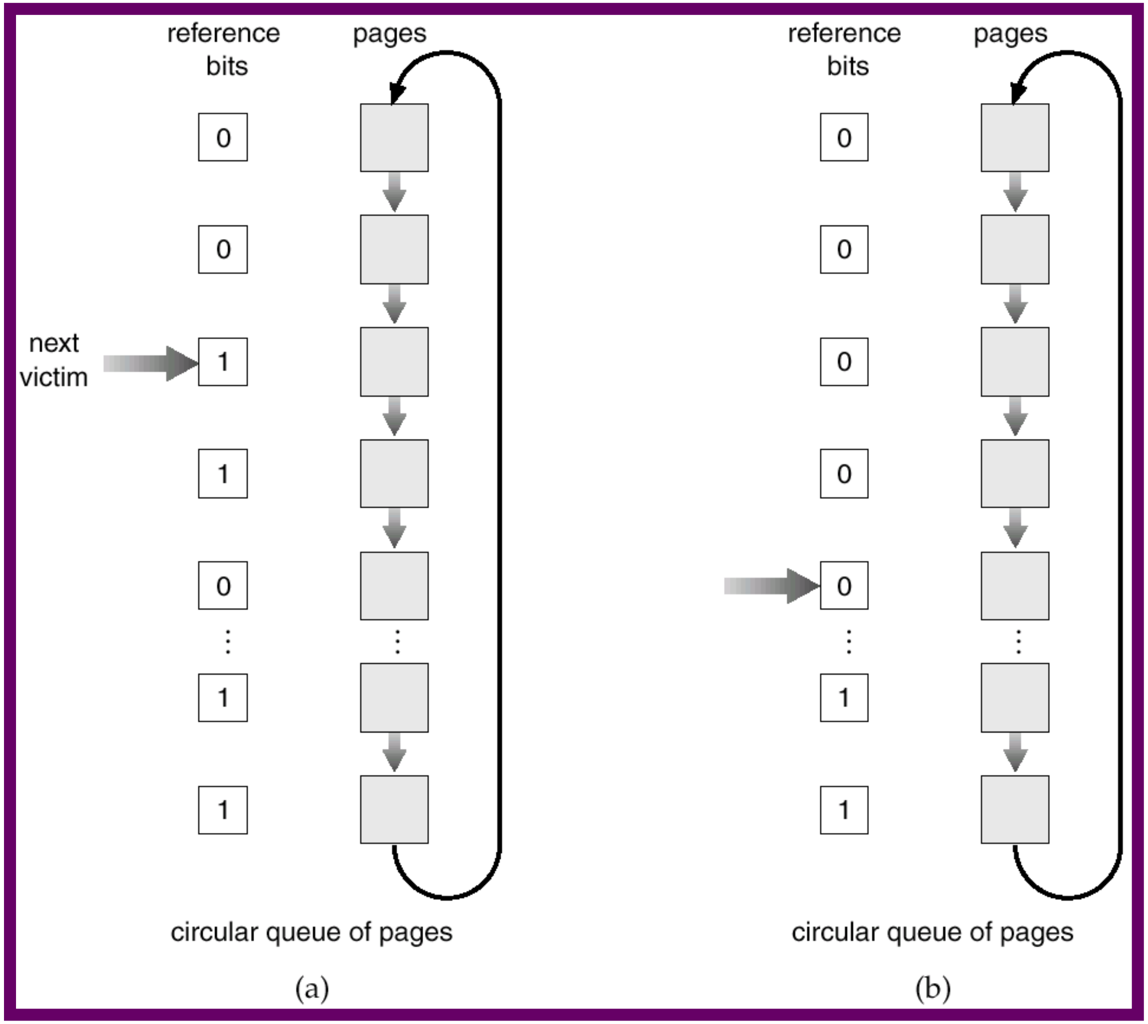
Performance issue: Stack and Counters

- The updating of stack or clock must be done on every memory reference.
- If we use interrupt for every reference, to allow software to update data structures, it would slow every reference by a factor of 10.
  - Few systems tolerate such degradation in performance.
- Sol:
  - Systems follow LRU approximation implemented through hardware.

LRU Approximation Algorithms

- Reference bit
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1.
  - Replace the one which is 0 (if one exists). We do not know the order, however.
- Additional ordering:
  - By maintaining 8-bit byte for each page.
  - Shifting can be used to record the history (interrupt to OS for every 100 msec).
- Second chance
  - Need one reference bit.
  - Clock replacement.
  - If page to be replaced (in clock order) has reference bit = 1. then:
    - ✔ set reference bit 0; arrival time is set to current time.
    - ✔ leave page in memory.
    - ✔ replace next page (in clock order), subject to same rules.
  - Similar to FIFO if all bits are set.

Second-Chance (clock) Page-Replacement Algorithm



Enhanced-second chance algorithm

- Use reference bit and modify bit as an ordered pair.
  - (0,0) neither recently used nor modified – best page to replace.
  - (0,1) not recently used but modified- not quite good; to be written before replacement.
  - (1,0) recently used but clean – it probably used again soon.
  - (1,1) recently used and modified- it probably will be used again soon; to be written before replacement.
- Each page is one of four classes.

Other algorithms: Counting Algorithms

- Keep a counter of the number of references that have been made to each page.

- LFU Algorithm: replaces page with smallest count.

- MFU Algorithm: based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

- MFU and LFU are not used
  - Implementation is expensive
  - Do not approximate OPT well

Allocation of Frames

- Each process needs **minimum** number of pages.
- If there is a single process, entire available memory can be allocated.
- Multi-programming puts two or more processes in memory at same time.
- We must allocate minimum number of frames to each process.
- Two major allocation schemes.
  - fixed allocation
  - priority allocation

Minimum number of Frames

- Each process needs **minimum** number of pages.
- Minimum # of frames is defined based on the computer architecture.
  - Equal to maximum number of memory references per instruction.
    - ✔ LOAD may refer indirect reference that could also reference an indirect address.
  - # of frames = number of indirections.
    - ✔ Counter can be used to measure the number of indirections and trap the OS.
  - MVC in IBM370 machine requires 6 frames.
- Two major allocation schemes.
  - fixed allocation
  - priority allocation

Fixed Allocation

- Equal allocation – e.g., if 100 frames and 5 processes, give each 20 pages.
- Proportional allocation – Allocate according to the size of process.
  - $s_i = \text{size of process } p_i$
  - $S = \sum s_i$
  - $m = \text{total number of frames}$
  - $a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$

$$m = 64$$
$$s_i = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 64 \approx 5$$
$$a_2 = \frac{127}{137} \times 64 \approx 59$$

Priority Allocation

- Use a proportional allocation scheme using priorities rather than size.

- If process P_i generates a page fault,
  - select for replacement one of its frames.
  - select for replacement a frame from a process with lower priority number.

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.
- **Local** replacement – each process selects from only its own set of allocated frames.
- With local replacement, # of frames does not change.
  - Performance depends on the paging behavior of the process.
  - Free frames may not be used
- With global replacement, a process can take a frame from another process.
  - Performance depends not only paging behavior of that process, but also paging behavior of other processes.
- In practice global replacement is used.

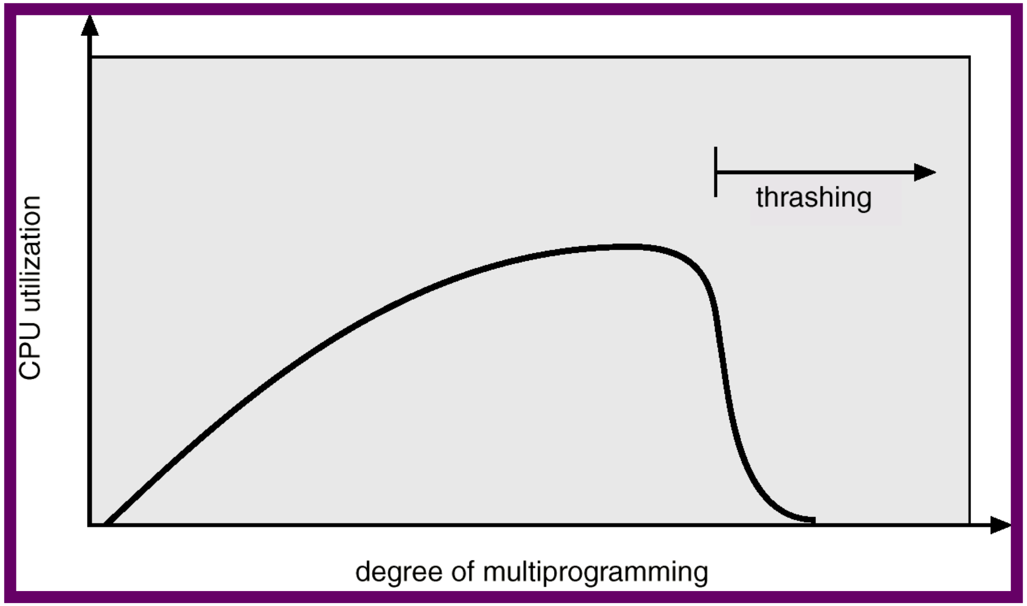
Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
  - low CPU utilization.
  - operating system thinks that it needs to increase the degree of multiprogramming.
  - another process is added to the system.

- Thrashing is High paging activity.

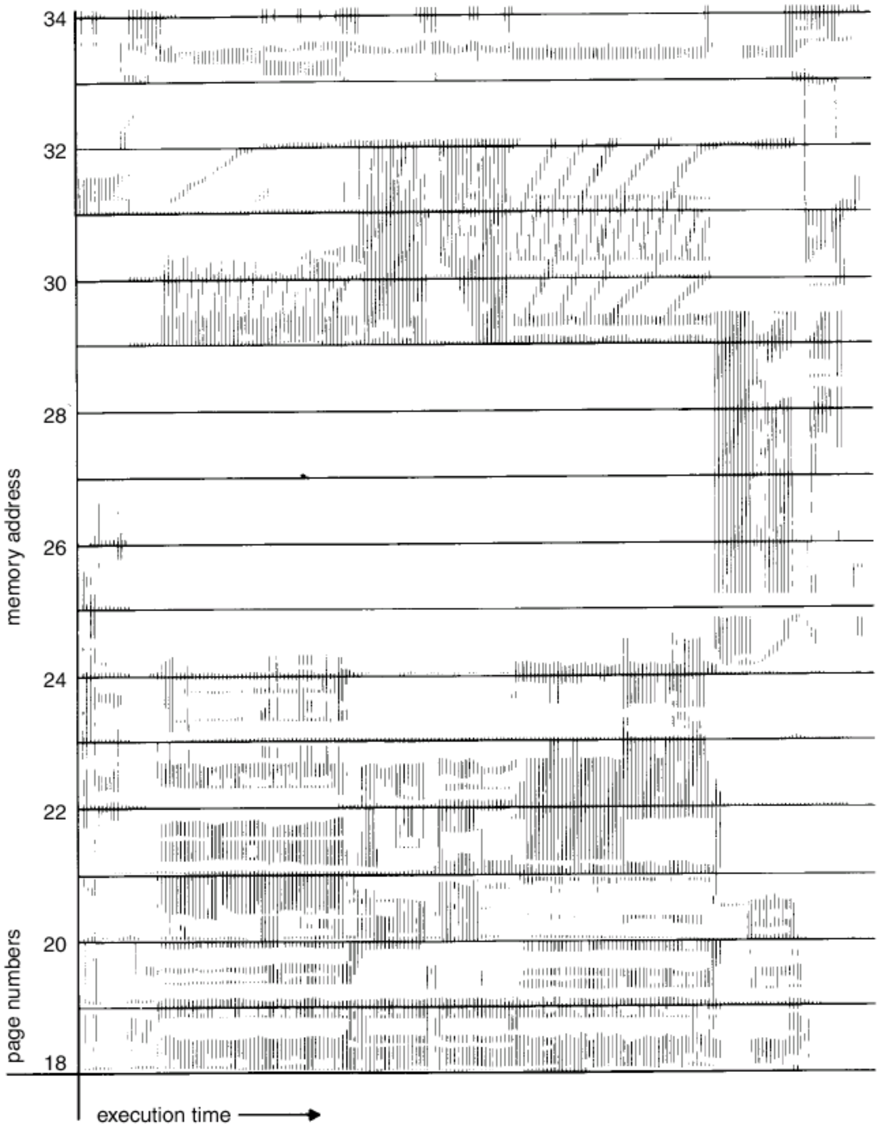
- **Thrashing** $\equiv$ a process is spending more time in swapping pages in and out.
- If the process does not have # of frames equivalent to # of active pages, it will very quickly page fault.
- Since all the pages are in active use it will page fault again.

Thrashing



- Why does 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

Locality In A Memory-Reference Pattern



Causes of thrashing

- OS monitors CPU utilization
  - If it is low, increases the degree of MPL
- Consider that a process enters new execution phase and starts faulting.
- It takes pages from other processes
- Since other processes need those pages, they also fault, taking pages from other processes.
- The queue increases for paging device and ready queue empties
- CPU utilization decreases.
- Solution: provide process as many frames as it needs.
- But how we know how many frames it needs ?
- Locality model provides hope.

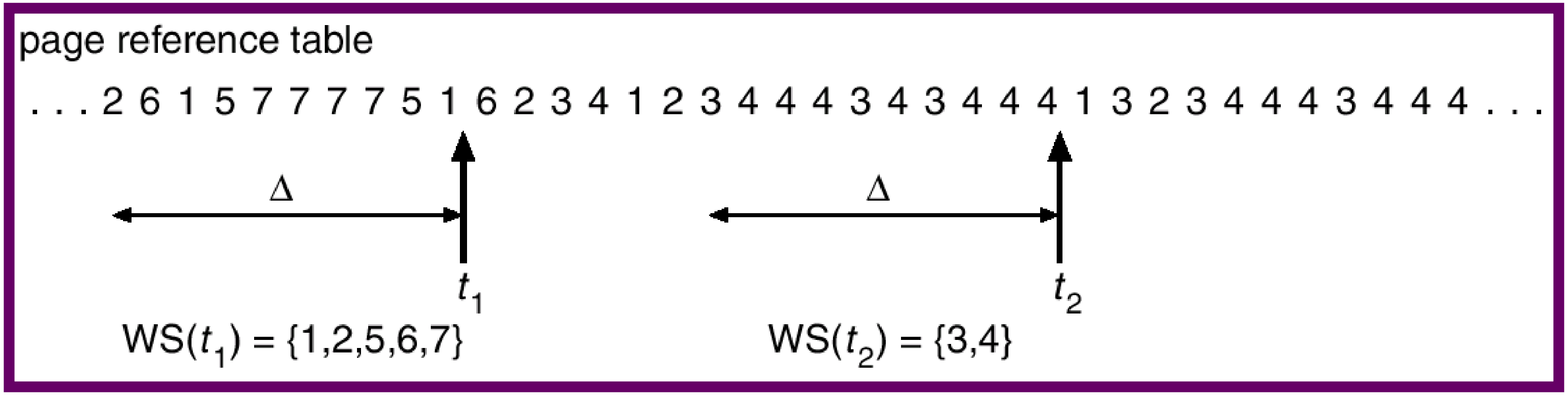
Locality model

- Locality is a set of pages that are actively used together.
- A program is composed of several different localities which may overlap.
  - Ex: even when a subroutine is called it defines a new locality.
- The locality model states that all the programs exhibit this memory reference structure.
- **This is the main reason for caching and virtual memory!**
- **If we allocate enough frames to a process to accommodate its current locality, it faults till all pages are in that locality are in the MM. Then it will not fault.**
- **If we allocate fewer frames than current locality, the process will thrash.**

Working-Set Model

- Based on locality
- Define a parameter Δ;
- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instruction
- Most recent references are examined
- 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
- if $D > m \Rightarrow$ Thrashing
- Policy: if $D >$ m, then suspend one of the processes.

Working-set model



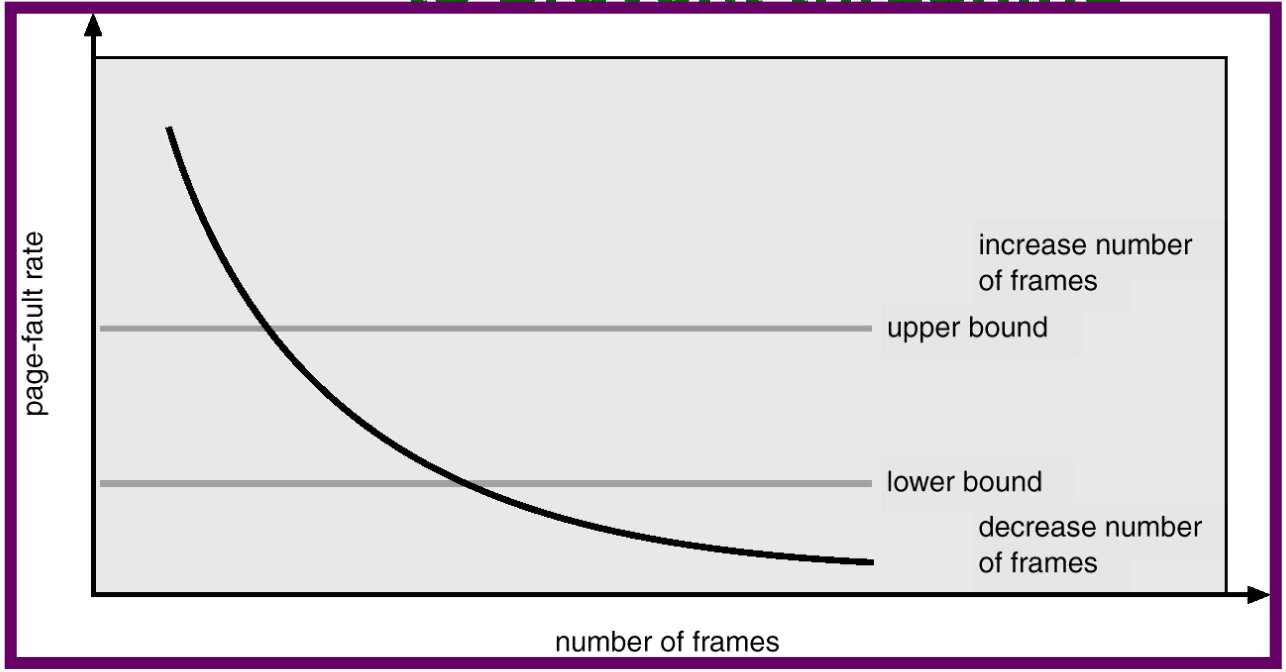
Working Set

- OS monitors the WS of each process allocates to that working set enough frames equal to WS size.
- If there are enough extra frames, another process can be initiated.
- If D>m, OS suspends a process, and its frames are allocated to other processes.
- The WS strategy prevents thrashing by keeping MPL as high as possible.
- However, we have to keep track of working set.

Keeping track of 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?
  - We can not tell when the reference was occurred.
  - Accuracy can be increased by increasing frequency of interrupts which also increases the cost.

Page-Fault Frequency approach to prevent thrashing



- Thrashing has a high page-fault rate.
- Solution: Control or establish "acceptable" page-fault rate.
  - If page fault rate is too low (below lower bound), process loses frame.
  - If page fault l rate is too high (exceeds upper bound), process gains frame.
- Process is suspended if no free frames are available. The freed frames are distributed among other processes.

Allocation of Kernel memory

- Buddy allocation
  - Allocates fixed size segment consisting of physically contiguous pages
  - Uses power-of-2 allocator
    - Adjacent buddies can be joined to meet a bigger request
- Slab allocation
  - A slap is made up of one or more physically contiguous pages.
  - Use different caches for different size/kernel data structure.
  - Caches are mapped to slabs.

Other Considerations

- The selection of a page replacement algorithm and allocation policy are major decisions. There are many other considerations as well.
- Prepaging
  - Bring entire WS to the memory to prevent high level of initial paging.
- Page size selection
  - Fragmentation
    - Memory is better utilized if we have a small page size as pages are units of allocation.
  - table size
    - Large page size is desirable to decrease table size.
  - I/O overhead
    - Minimize I/O time argues for a larger page size.
  - Locality
    - With smaller page size, the locality will be improved.

Other Considerations (Cont.)

- **TLB Reach –**
  - Hit ratio should be increased.
    - ✓ The amount of memory accessible from the TLB.
  - Ideally, the working set of each process is stored in the TLB. Otherwise there is a high degree of page faults.
  - TLB Reach = (TLB Size) X (Page Size)
  - **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.

Other Considerations (Cont.)

- **System performance can be improved if the user is completely aware of the paged nature of memory.**
- **Program structure**
  - **int A[][] = new int[1024][1024];**
  - Each row is stored in one page
  - Program 1

```
for (j = 0; j < A.length; j++)
    for (i = 0; i < A.length; i++)
        A[i,j] = 0;
```

    1024 x 1024 page faults

  - Program 2

```
for (i = 0; i < A.length; i++)
    for (j = 0; j < A.length; j++)
        A[i,j] = 0;
```

    1024 page faults

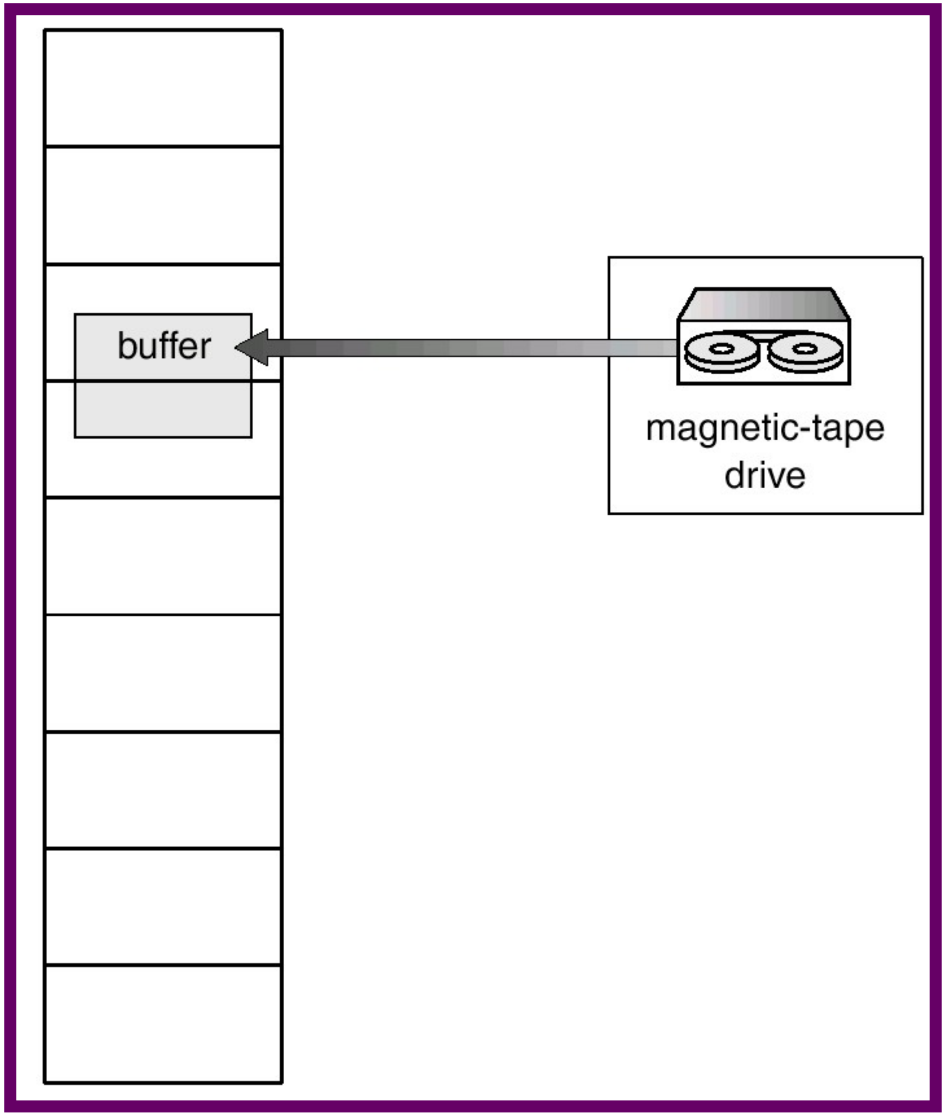
- **Careful selection of data structures and programming structures can increase locality, lower the page fault rate and the number of pages in the working set.**
- Java (no pointers) has better locality of reference than C or C++ due to pointers that randomize page references.

Other Considerations (Cont.)

- **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.

Reason Why Frames Used For I/O Must Be In Memory



Operating System Examples

- Windows NT
- Solaris 2

Windows NT

- 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 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.

Solaris 2

- Maintains a list of free pages to assign faulting processes.
- **Lotsfree** – threshold parameter to begin paging.
- Paging is peformed by *pageout* process.
- Pageout scans pages using modified clock algorithm.
- **Scanrate** is the rate at which pages are scanned. This ranged from **slowscan** to **fastscan**.
- Pageout is called more frequently depending upon the amount of free memory available.

Solar Page Scanner