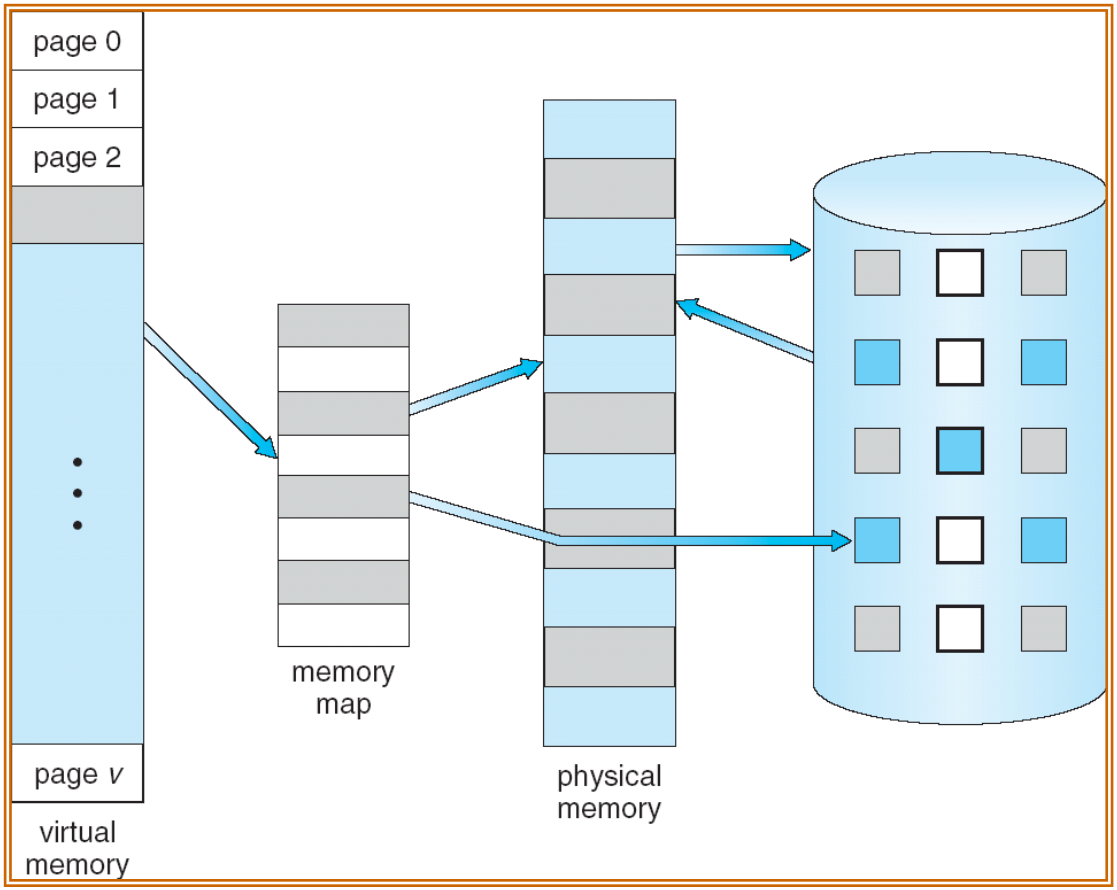
Virtual Memory

Background

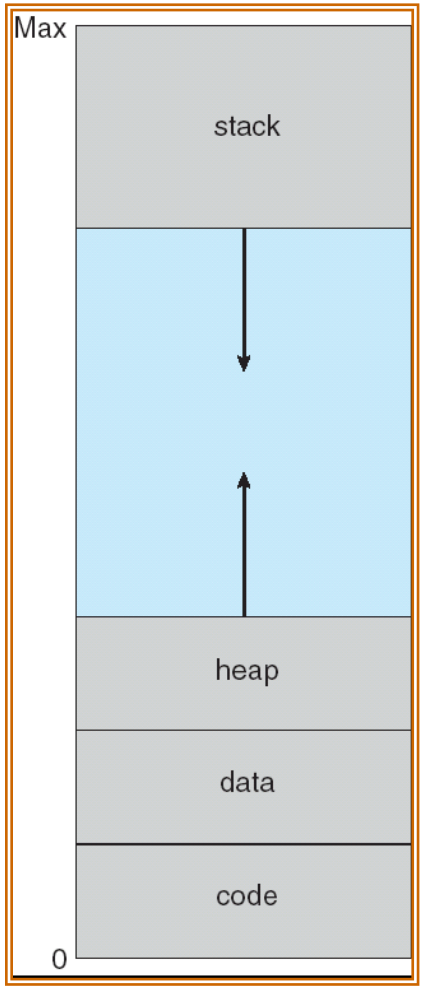
- **Virtual memory** – separation of user logical memory from physical memory.
  - Only part of the program needs to be in memory for execution.
  - Logical address space can therefore be much larger than physical address space.
  - Allows address spaces to be shared by several processes.
  - Allows for more efficient process creation.

- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

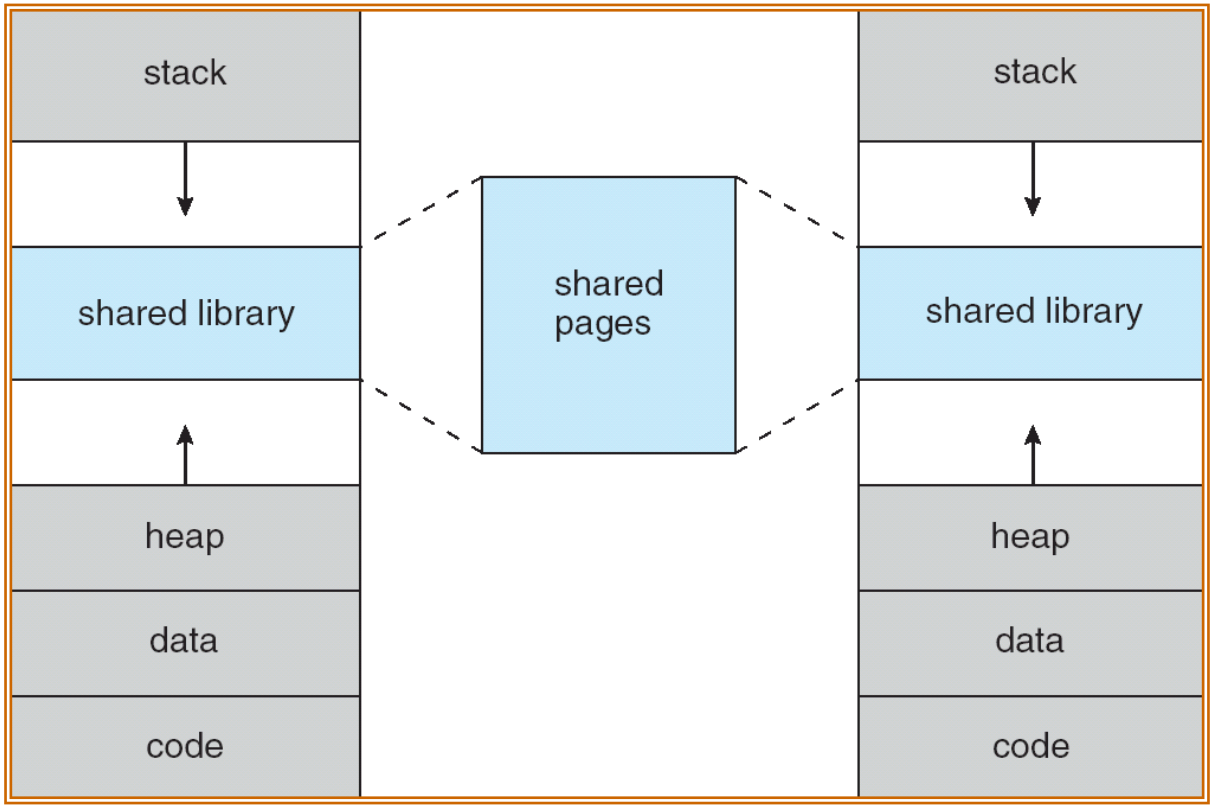
Virtual Memory That is Larger Than Physical Memory



Virtual-address Space



Shared Library Using Virtual Memory



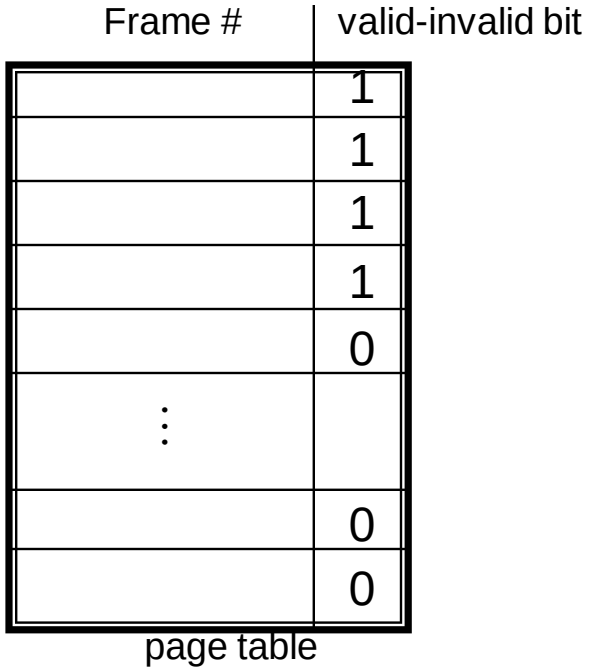
Demand Paging

- 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 into memory

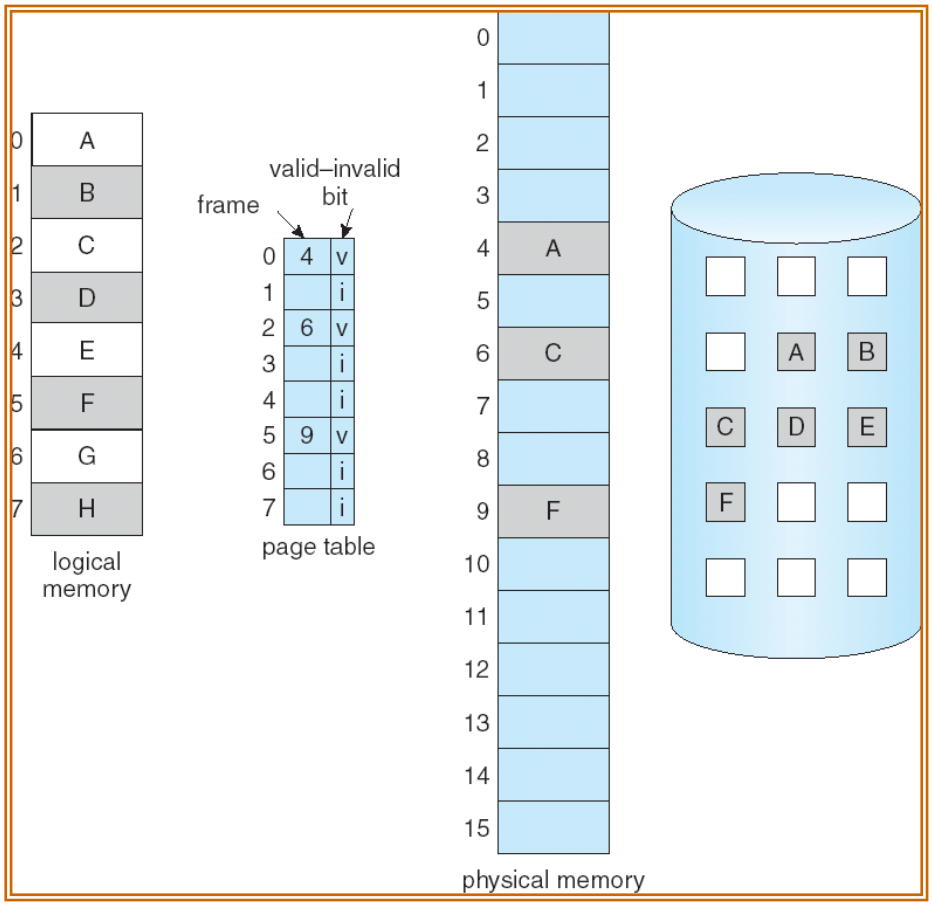
Valid-Invalid Bit

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



- During address translation, if valid–invalid bit in page table entry is 0 ⇒ page fault

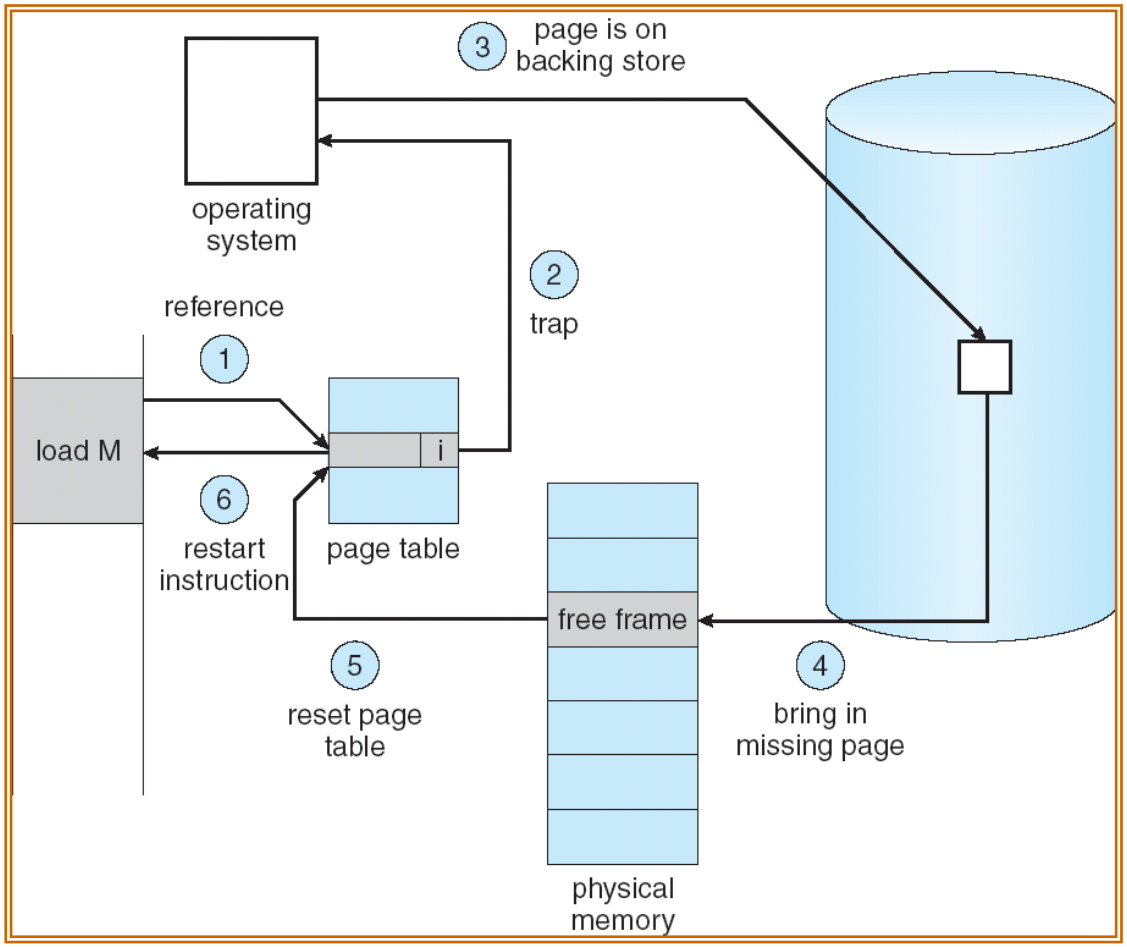
Page Table When Some Pages Are Not in Main Memory



Page Fault

- First reference to a memory location in a page will cause a page fault trap to the OS
- OS checks an internal table (usually kept with the PCB) to decide:
  - Invalid reference $\Rightarrow$ abort.
  - Just not in memory.
- Obtain empty frame.
- Read page into frame.
- Reset tables, valid bit = `1`.
- Restart instruction

Steps in Handling a Page Fault



What happens if there is no free frame?

- Page replacement – find a page in memory that is not currently in use, page 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

Performance of Demand Paging

- Probability of a page fault is $0 \leq p \leq 1.0$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault

- Effective Access Time

$$t_{eff} = \text{memory access time}$$
$$+ p \text{ (page fault overhead}$$
$$+ \text{[write page out]}$$
$$+ \text{read page in}$$
$$+ \text{restart overhead)}$$

Demand Paging Example

- Assume:
  - a memory access time of 1 μs
  - a page read/write time of 10 ms (10,000 μs)
  - negligible cost to service the page fault interrupt and to restart the instruction
- First, assume that a free frame (or a frame containing a readonly page) has been selected to receive the page from disk
- therefore, $t_{eff} = 1 + p \cdot 10000$ [μs]
- if we want $t_{eff} \leq 2$μs, we can solve for the maximum value of p
  - $1 + p \cdot 10000 \leq 2 \rightarrow p \leq 10^{-4}$
- If the frame chosen to receive the page had to be written first to disk, then $t_{eff} = 1 + p \cdot 20000$ [μs], implying that $p \leq 5 \times 10^{-5}$

Process Creation

- Virtual memory allows other benefits during process creation:
  - 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
    - Free pages are allocated from a **pool** of zeroed-out pages
  - Memory-Mapped Files (later)

Page Replacement

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

Basic Page Replacement

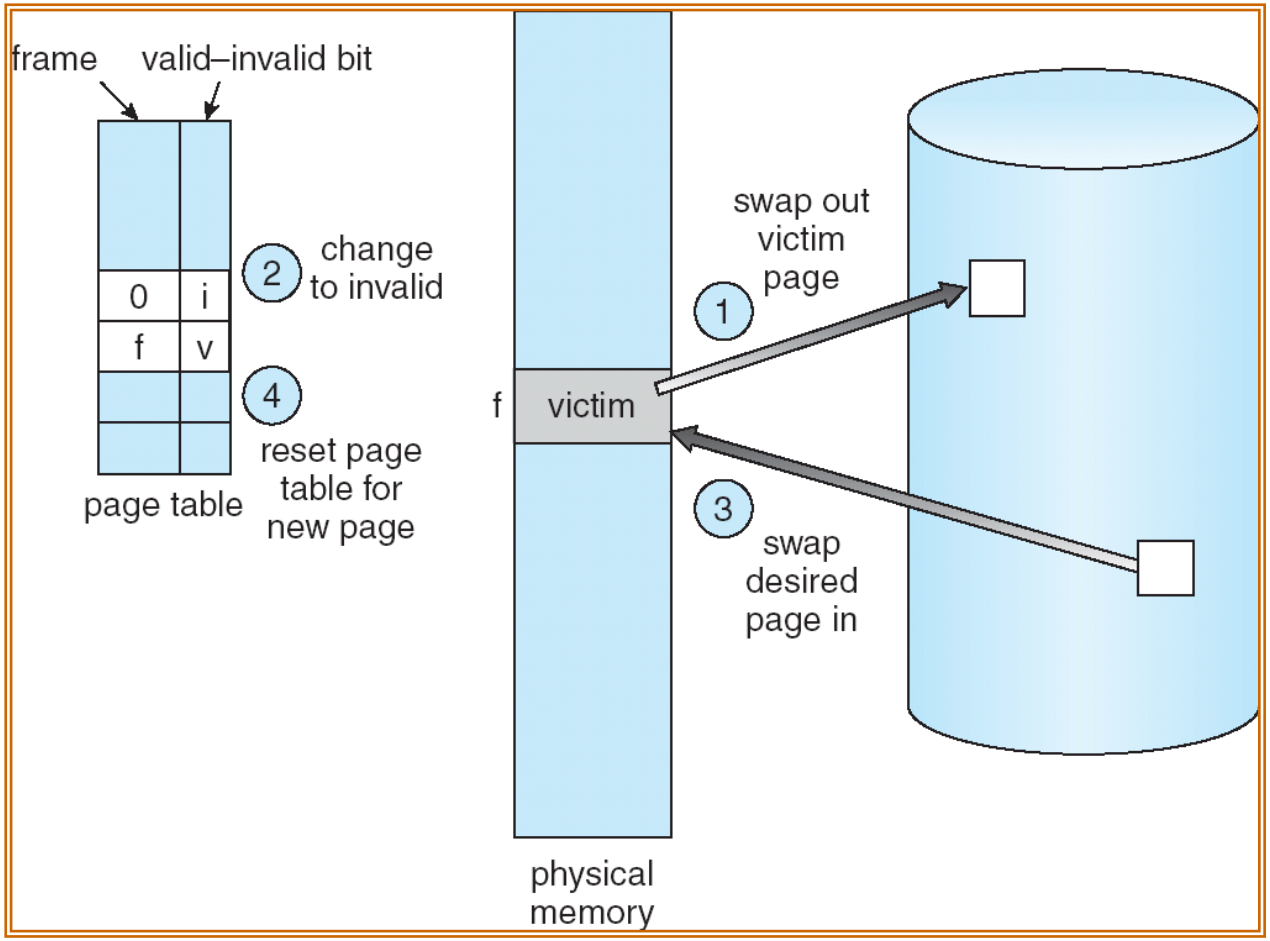
1. Find the location of the desired page on disk

2. Find a free frame:
   i. If there is a free frame, use it
   ii. If there is no free frame, use a page replacement algorithm to select a **victim** frame
   iii. If the victim frame is dirty, write it to backing store

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

4. Restart the process

Page Replacement

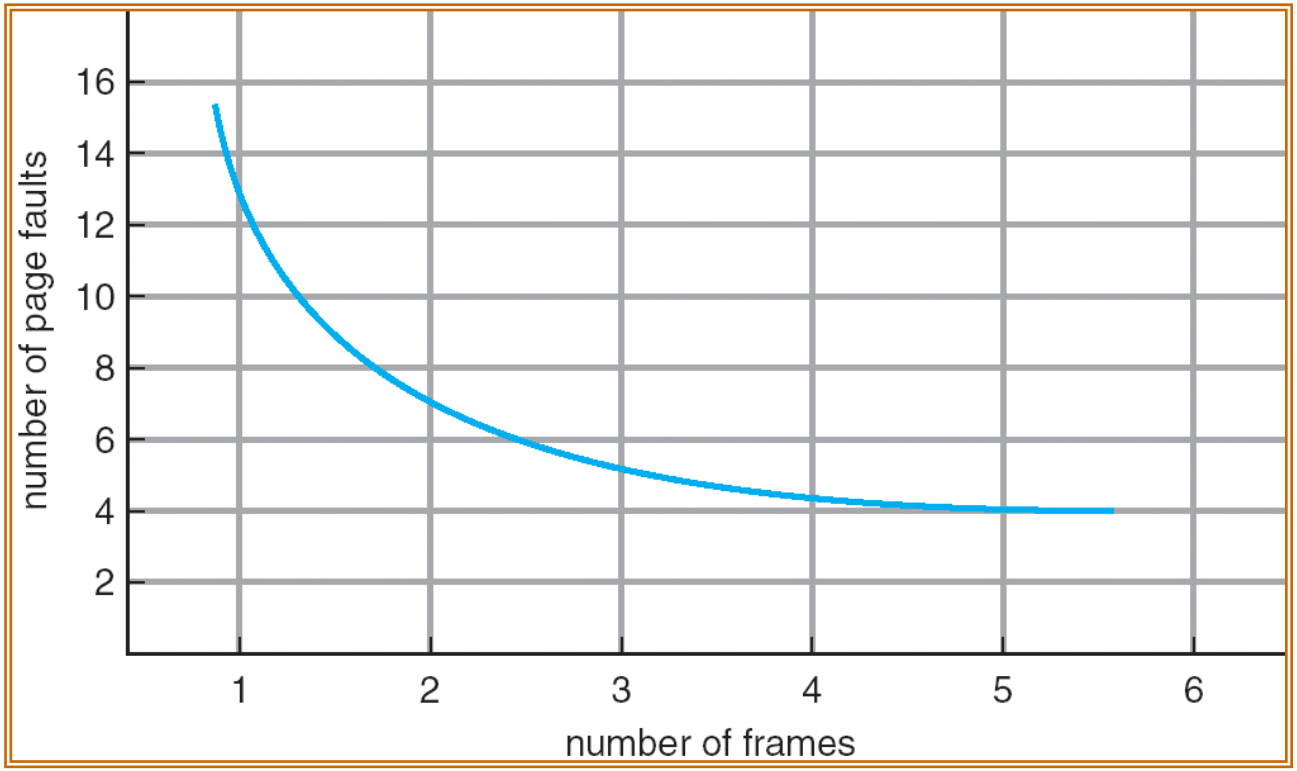


Page Replacement Algorithms

- 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`

Graph of Page Faults Versus The Number of Frames



First-In-First-Out (FIFO) Algorithm

- 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	1	4	4	4	5	5	5	5	5	5
		2	2	2	1	1	1	1	1	3	3	3
			3	3	3	2	2	2	2	2	4	4
Reference string	1	2	3	4	1	2	5	1	2	3	4	5

9 page faults

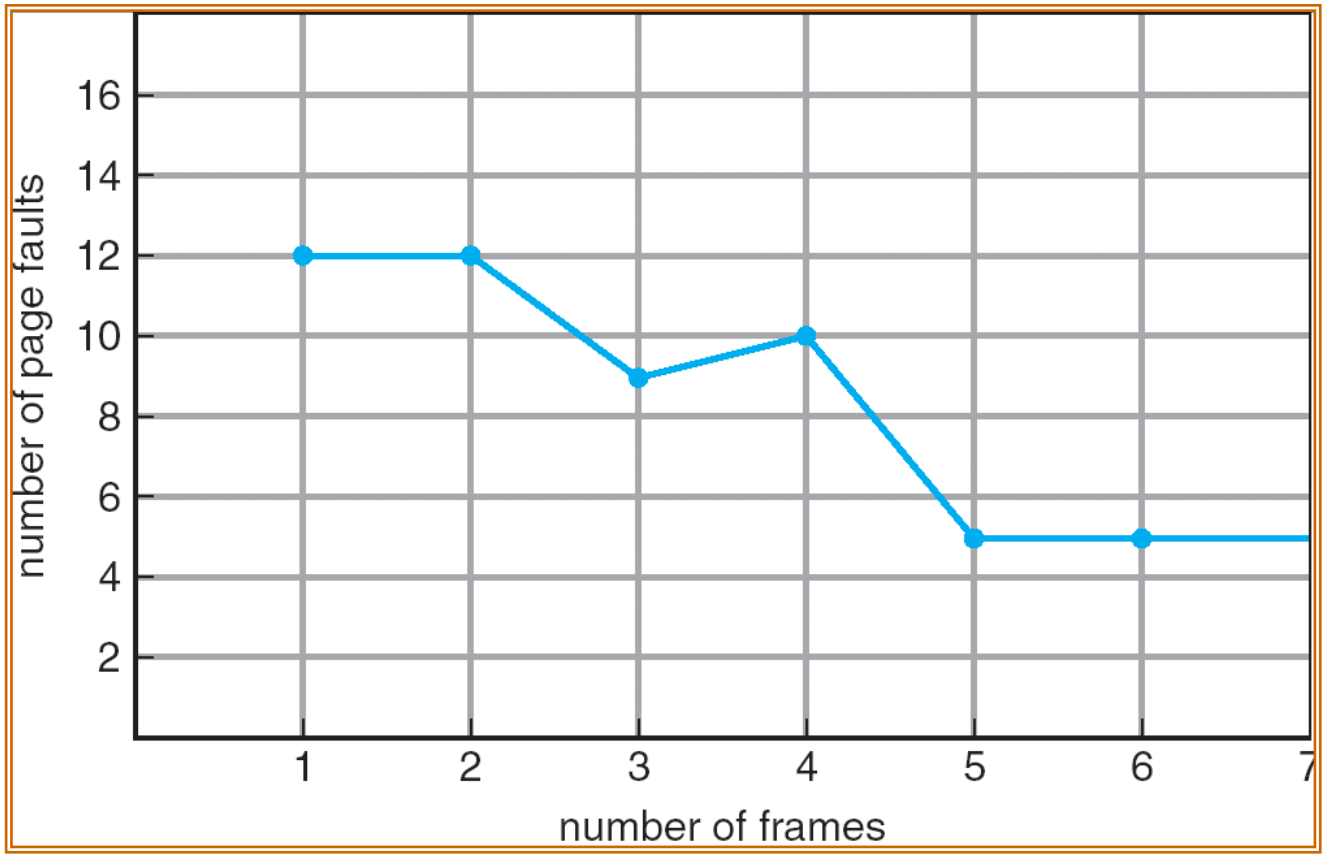
- 4 frames

	1	1	1	1	1	1	5	5	5	5	4	4
		2	2	2	2	2	2	1	1	1	1	5
			3	3	3	3	3	3	2	2	2	2
				4	4	4	4	4	4	3	3	3
Reference string	1	2	3	4	1	2	5	1	2	3	4	5

10 page faults

- FIFO Replacement exhibits Belady's Anomaly
  - more frames $\Rightarrow$ more page faults

FIFO Illustrating Belady's Anomaly



Optimal Algorithm

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

1	1	1	1	1	1	1	1	1	1	4	4
	2	2	2	2	2	2	2	2	2	2	2
		3	3	3	3	3	3	3	3	3	3
			4	4	4	5	5	5	5	5	5
Reference string	1	2	3	4	1	2	5	1	2	3	4	5

6 page faults

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

Least Recently Used (LRU) Algorithm

1	1	1	1	1	1	1	1	1	1	1	5
	2	2	2	2	2	2	2	2	2	2	2
		3	3	3	3	5	5	5	5	4	4
			4	4	4	4	4	4	3	3	3
Reference string	1	2	3	4	1	2	5	1	2	3	4	5

8 page faults

- 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

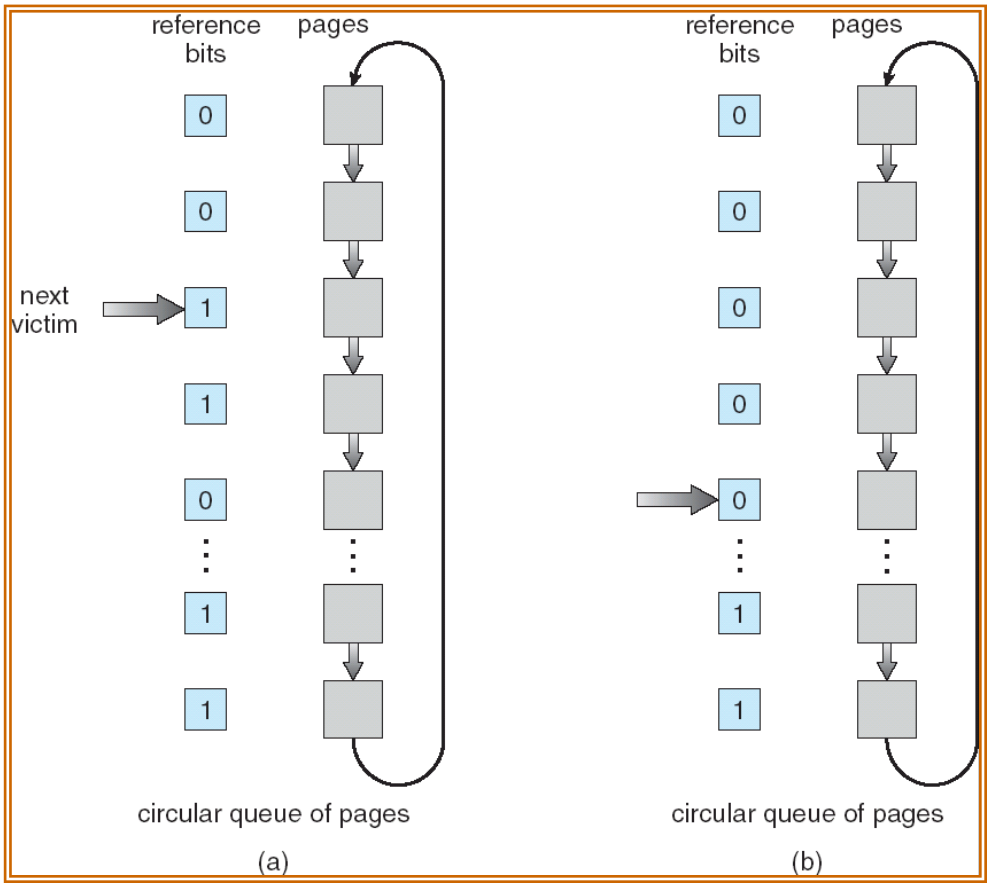
LRU Algorithm (Cont.)

- Stack implementation – keep a stack of page numbers in a doubly linked list form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - No search for replacement

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.
- Second chance
  - Need reference bit
  - Clock replacement
  - If page to be replaced (in clock order) has reference bit = 1 then:
    - set reference bit 0
    - leave page in memory
    - replace next page (in clock order), subject to same rules

Second-Chance (clock) Page-Replacement Algorithm



2-handed Clock Algorithm

- Start with normal Clock Algorithm
- As with Clock, have the Victim Hand, V
- Introduce additional hand, the Clearing Hand, C
- Assuming that V takes on values of 0..N-1, C = V + Δ mod N
- When page replacement is required:
- While (ref bit of V == 1) {
  V = (V+1) % N; C = (C+1) % N; clear ref bit of C;
  }
  replace V;
- If Δ very small, similar to normal Clock
- As Δ gets larger, provides more time/opportunity for cleared reference bits to be reset

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

Allocation of Frames

- Each process needs a *minimum* number of pages
- 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*
- Two major allocation schemes
  - fixed allocation
  - priority allocation

Fixed Allocation

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

$$m = 64$$

$$s_i = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 64 \approx 5$$

$$a_2 = \frac{127}{137} \times 64 \approx 59$$

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

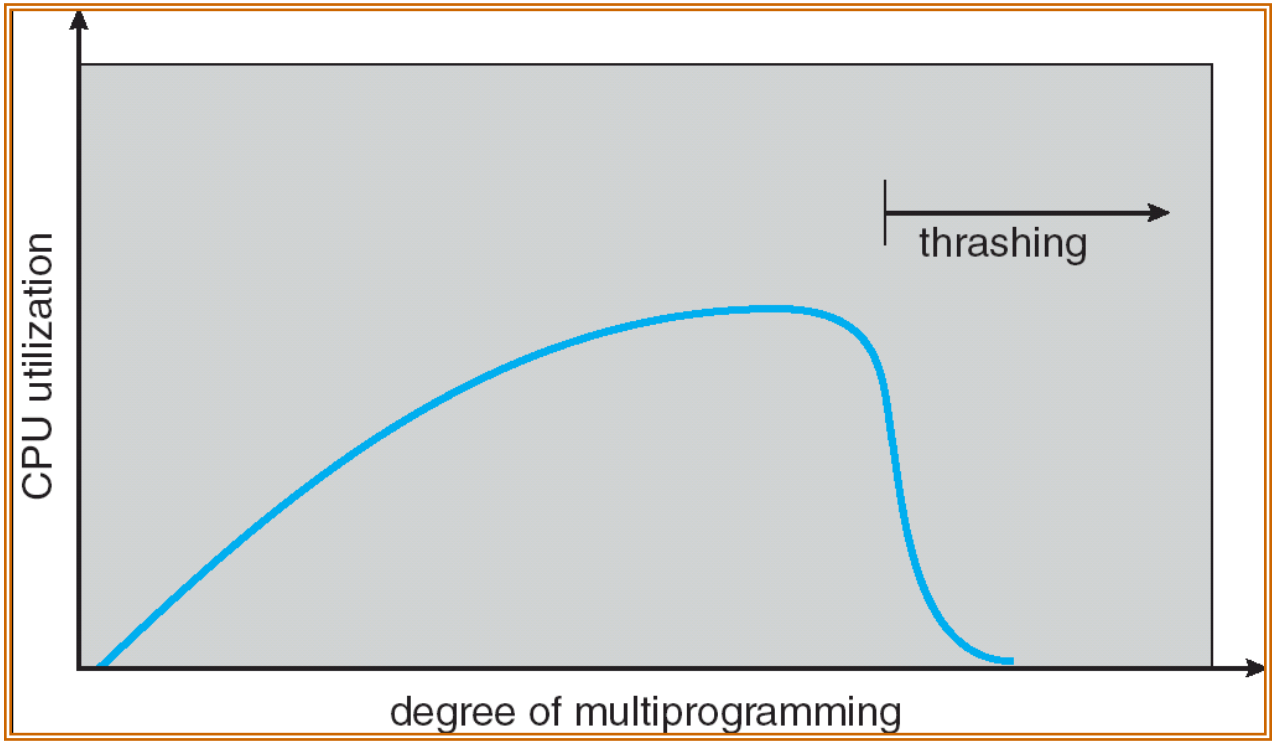
Global vs. Local Allocation

- **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

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 added to the system
- **Thrashing** ≡ a process is busy paging pages in and out

Thrashing (Cont.)

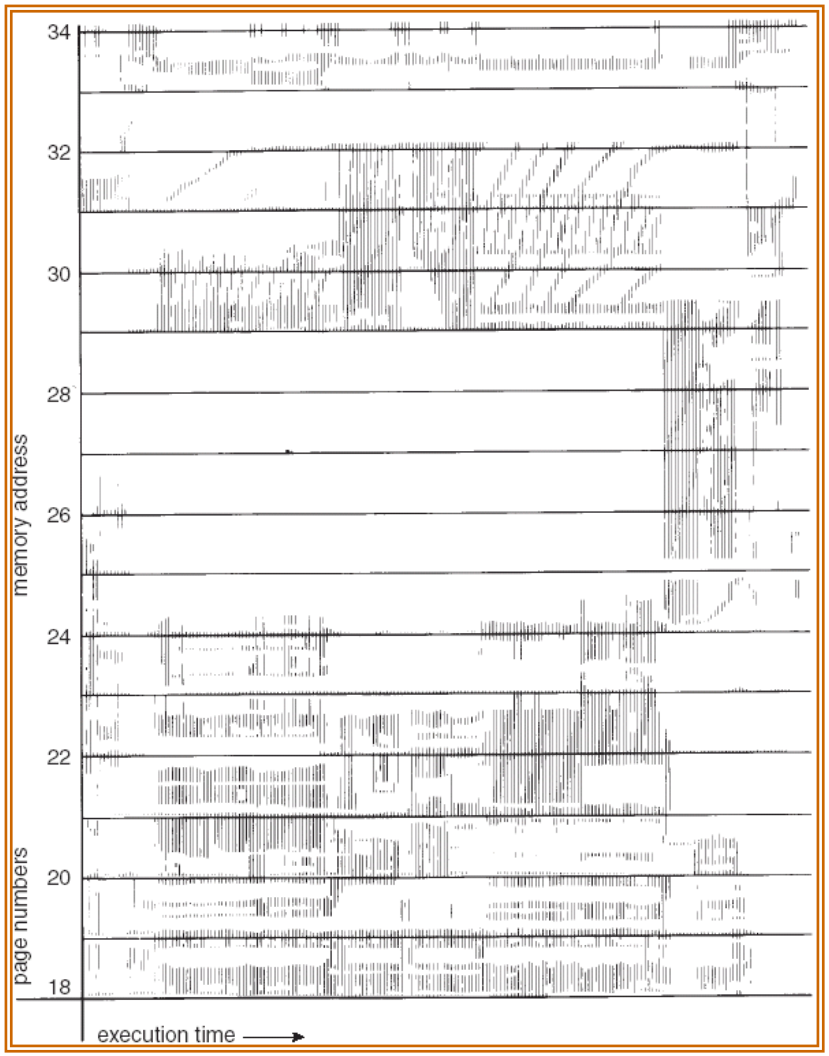


Demand Paging and Thrashing

- 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

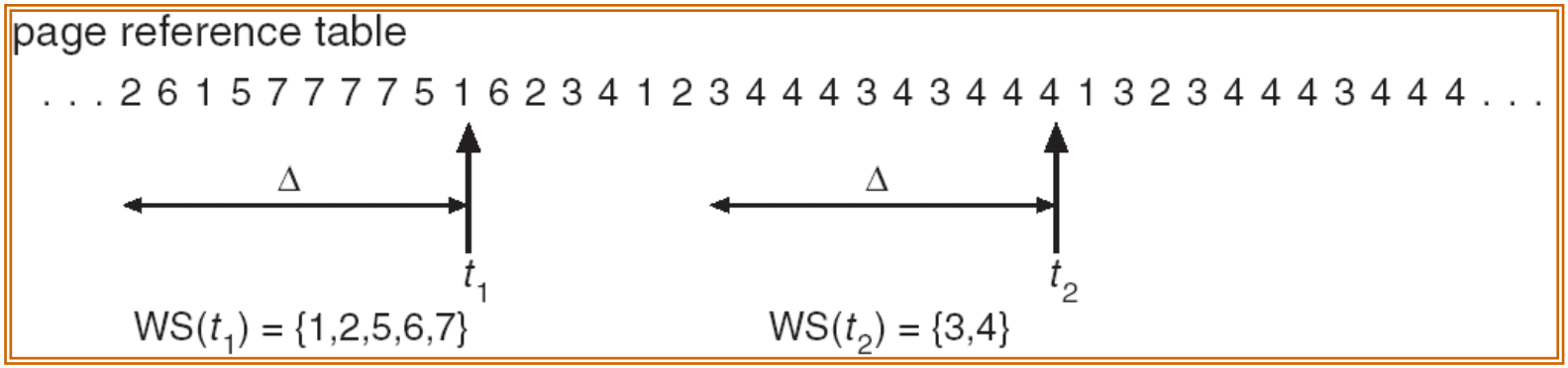
Locality In A Memory-Reference Pattern



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
- if $D >$ *m (total number of frames)* $\Rightarrow$ Thrashing
- Policy if $D >$ m, then suspend one of the processes and swap it out - i.e. the process is no longer eligible to be scheduled

Working-set model

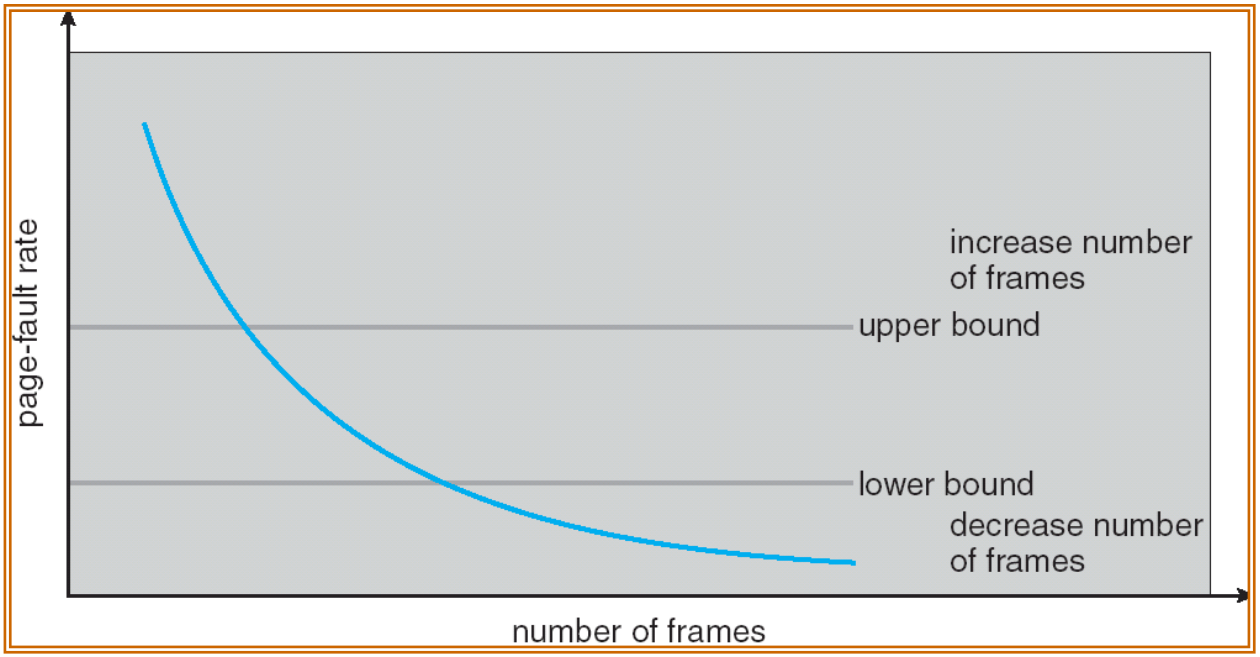


Keeping Track of the Working Set

- 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 set 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

Page-Fault Frequency Scheme

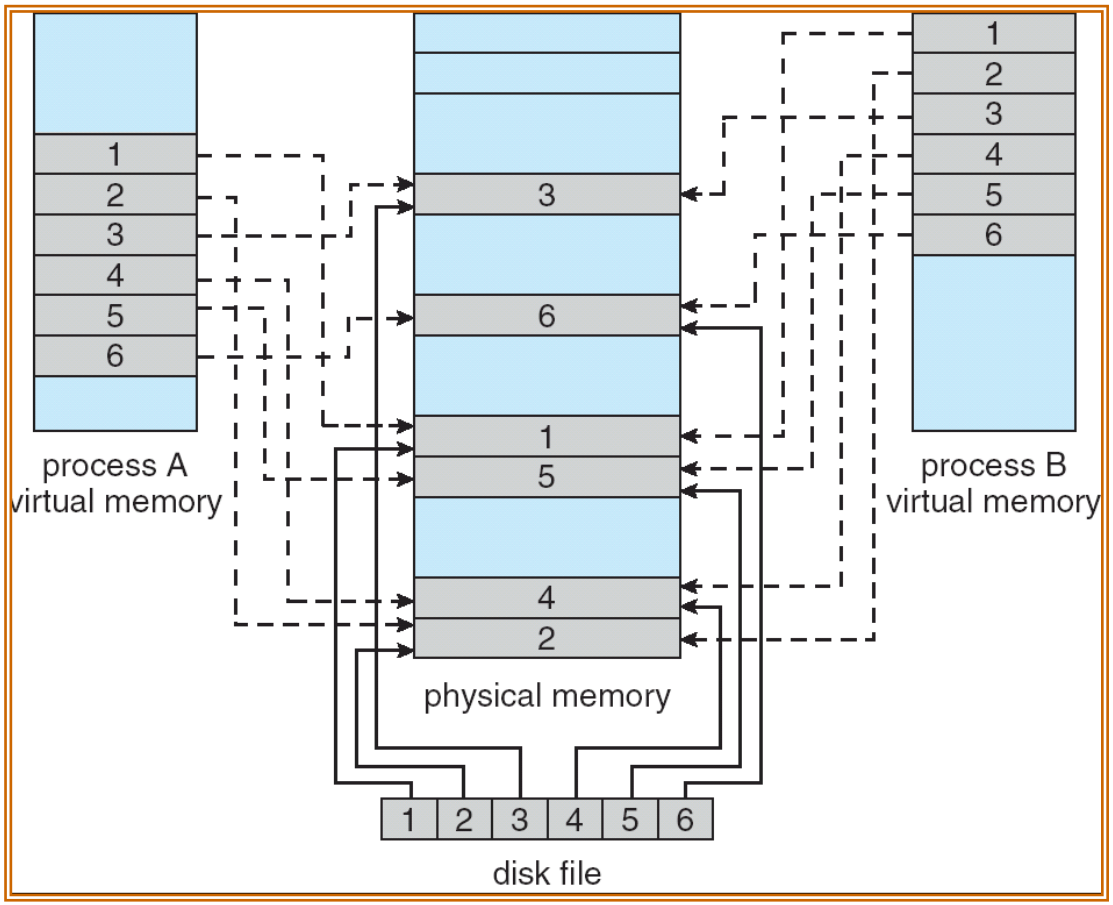
- Establish "acceptable" page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



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



Other Issues -- 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 is the fraction of the pages actually used
    - Is the cost of $s * \alpha$ saved pages faults > or < than the cost of prepaging $s * (1 - \alpha)$ unnecessary pages?
    - α near zero $\Rightarrow$ prepaging loses

Other Issues – Page Size

- Page size selection must take into consideration:
  - fragmentation
  - table size
  - I/O overhead
  - locality

Other Issues – 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 the process will spend a considerable amount of time resolving memory references in the page table rather than the TLB.
- 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. Note that this requires that the operating system, not the hardware, has to manage the TLB.

Other Issues – Program Structure

- 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

Other Issues – I/O interlock

- **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 to prevent being selected for eviction by a page replacement algorithm.

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