Operating Systems 2023 Spring Term
Week 12

Dr. Emrah İnan (emrahinan@iyte.edu.tr)

Virtual Memory

May 25, 2023

Week 11: Sample Glossary

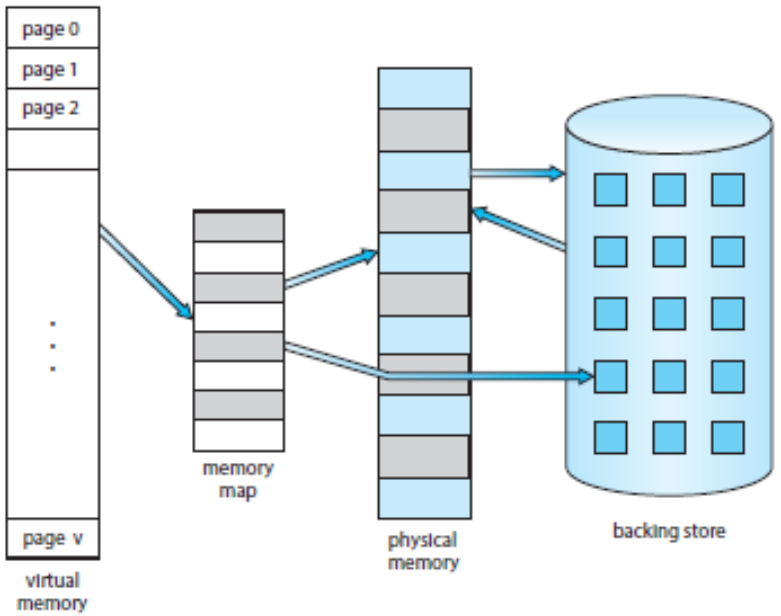
- **swapping:** Moving a process between main memory and a backing store. A process may be swapped out to free main memory temporarily and then swapped back in to continue execution. (on Page 1271)
- **paging:** A common memory management scheme that avoids external fragmentation by splitting physical memory into fixed-sized frames and logical memory into blocks of the same size called pages.
- **demand paging:** In memory management, bringing in pages from storage as needed rather than, e.g., in their entirety at process load time. (on Page 1245)

Background

- The memory-management algorithms outlined in Chapter 9 are necessary because of one basic requirement: **the instructions being executed must be in physical memory.** The first approach to meeting this requirement is to place the entire logical address space in physical memory. Dynamic linking can help to ease this restriction, but it generally requires special precautions and extra work by the programmer. The requirement that instructions must be in physical memory to be executed seems both necessary and reasonable; but it is also unfortunate, since it limits the size of a program to the size of physical memory. In fact, an examination of real programs shows us that, in many cases, the entire program is not needed.

Virtual Memory That is Larger Than Physical Memory

Virtual memory involves the separation of logical memory as perceived by developers from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available. Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available; she can concentrate instead on programming the problem that is to be solved.



Demand Paging

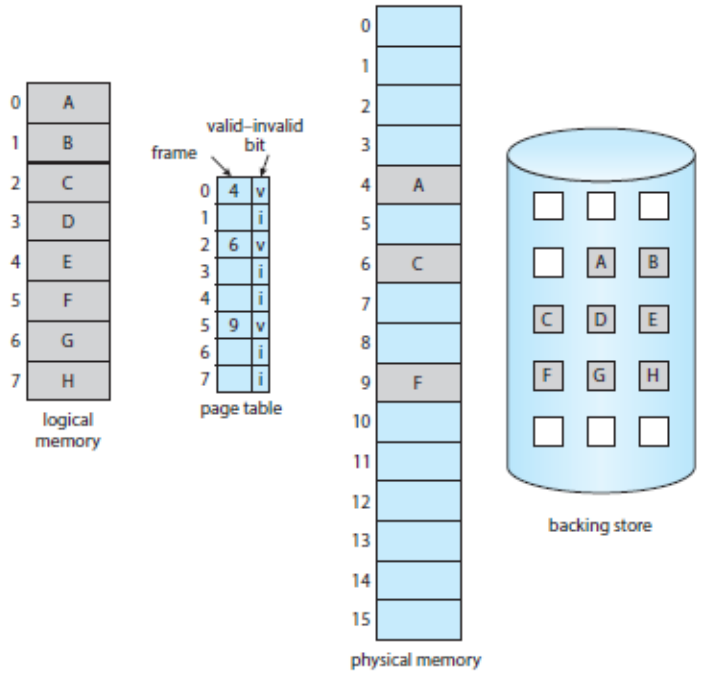
An executable program might be loaded from secondary storage into memory.

- Option (1): to load the entire program in physical memory at program execution time. However, a problem with this approach is that we may not initially need the entire program in memory. Suppose a program starts with a list of available options from which the user is to select. Loading the entire program into memory results in loading the executable code for all options, regardless of whether or not an option is ultimately selected by the user.
- Option (2): to load pages only as they are needed. With demand-paged virtual memory, pages are loaded only when they are demanded during program execution. Pages that are never accessed are thus never loaded into physical memory.

(+) by loading only the portions of programs that are needed, memory is used more efficiently.

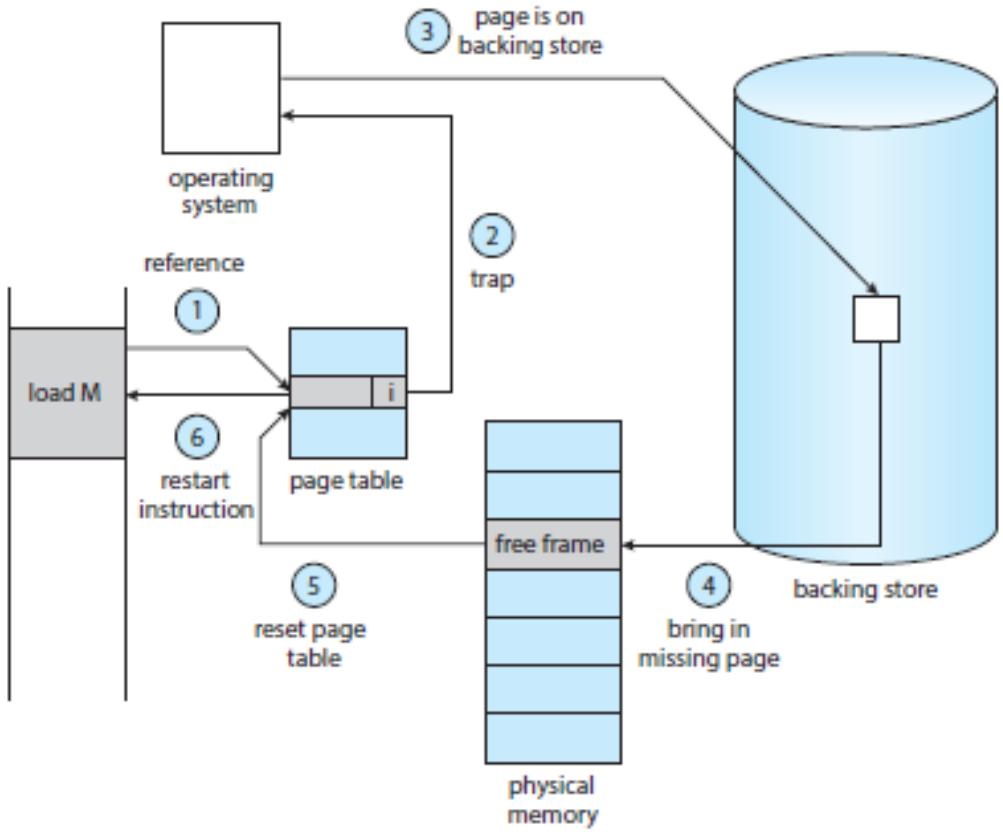
Page Table When Some Pages Are Not in Main Memory

When the bit is set to "valid," the associated page is both legal and in memory. If the bit is set to "invalid," the page either is not valid (that is, not in the logical address space of the process) or is valid but is currently in secondary storage. The page-table entry for a page that is brought into memory is set as usual, but the page-table entry for a page that is not currently in memory is simply marked invalid. (Notice that marking a page invalid will have no effect if the process never attempts to access that page.)



Steps in Handling a Page Fault I

Access to a page marked invalid causes a page fault. The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system.



Steps in Handling a Page Fault II

1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.
2. If the reference was invalid, we terminate the process. If it was valid but we have not yet brought in that page, we now page it in.
3. We find a free frame (by taking one from the free-frame list, for example).
4. We schedule a secondary storage operation to read the desired page into the newly allocated frame.
5. When the storage read is complete,we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

Aspects of Demand Paging

- Extreme case – start process with no pages in memory
  OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  And for every other process pages on first access
  Pure demand paging: never bring a page into memory until it is required.
- Actually, a given instruction could access multiple pages -> multiple page faults
  Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
  Pain decreased because of locality of reference
- Hardware support needed for demand paging
  Page table with valid / invalid bit
  Secondary memory (swap device with swap space)
  Instruction restart

Instruction Restart

- A crucial requirement for demand paging is the ability to restart any instruction after a page fault. Because we save the state (registers, condition code, instruction counter) of the interrupted process when the page fault occurs, we must be able to restart the process in exactly the same place and state, except that the desired page is now in memory and is accessible.
- In most cases, this requirement is easy to meet. A page fault may occur at any memory reference. If the page fault occurs on the instruction fetch, we can restart by fetching the instruction again. If a page fault occurs while we are fetching an operand, we must fetch and decode the instruction again and then fetch the operand.

Free-Frame List

When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory.

Most operating systems maintain a **free-frame list** -- a pool of free frames for satisfying such requests.

Operating system typically allocate free frames using a technique known as **zero-fill-on-demand** -- the content of the frames zeroed-out before being allocated.

When a system starts up, all available memory is placed on the free-frame list.

Stages in Demand Paging I – Worse Case

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:
   a) Wait in a queue for this device until the read request is serviced
   b) Wait for the device seek and/or latency time
   c) Begin the transfer of the page to a free frame

Stages in Demand Paging II – Worse Case

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)

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

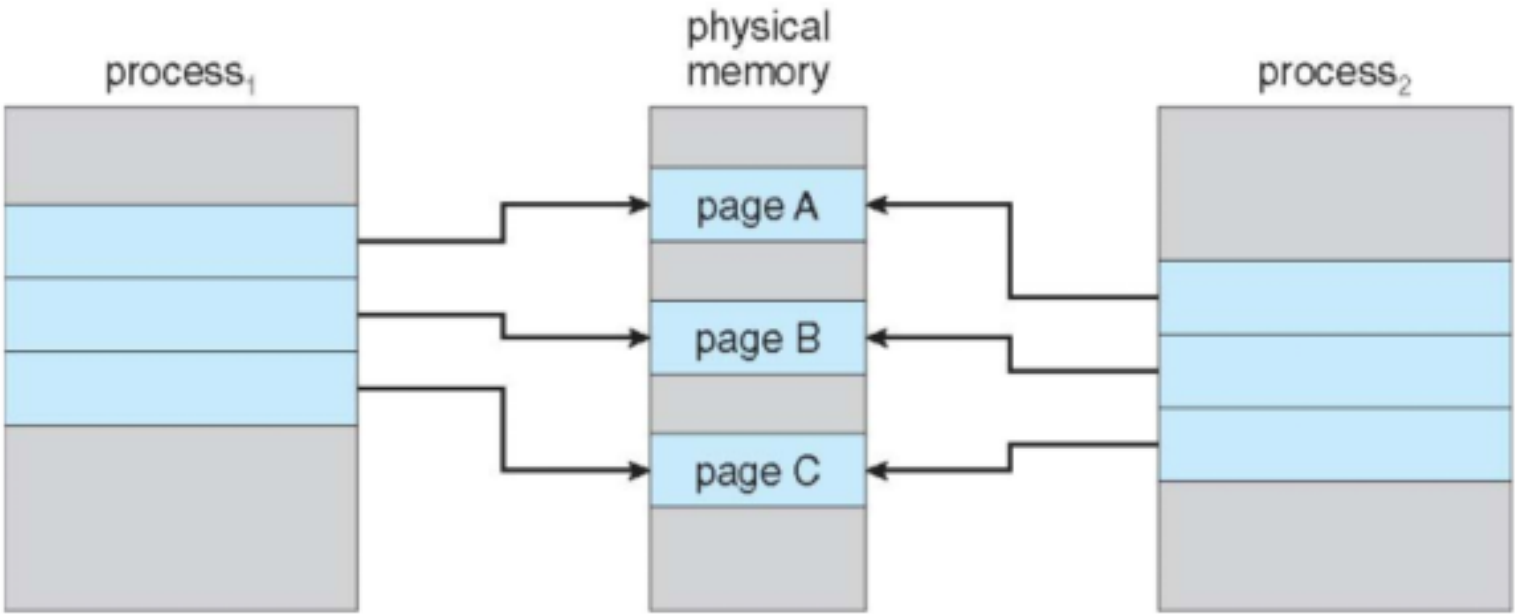
p < .0000025

< one page fault in every 400,000 memory accesses

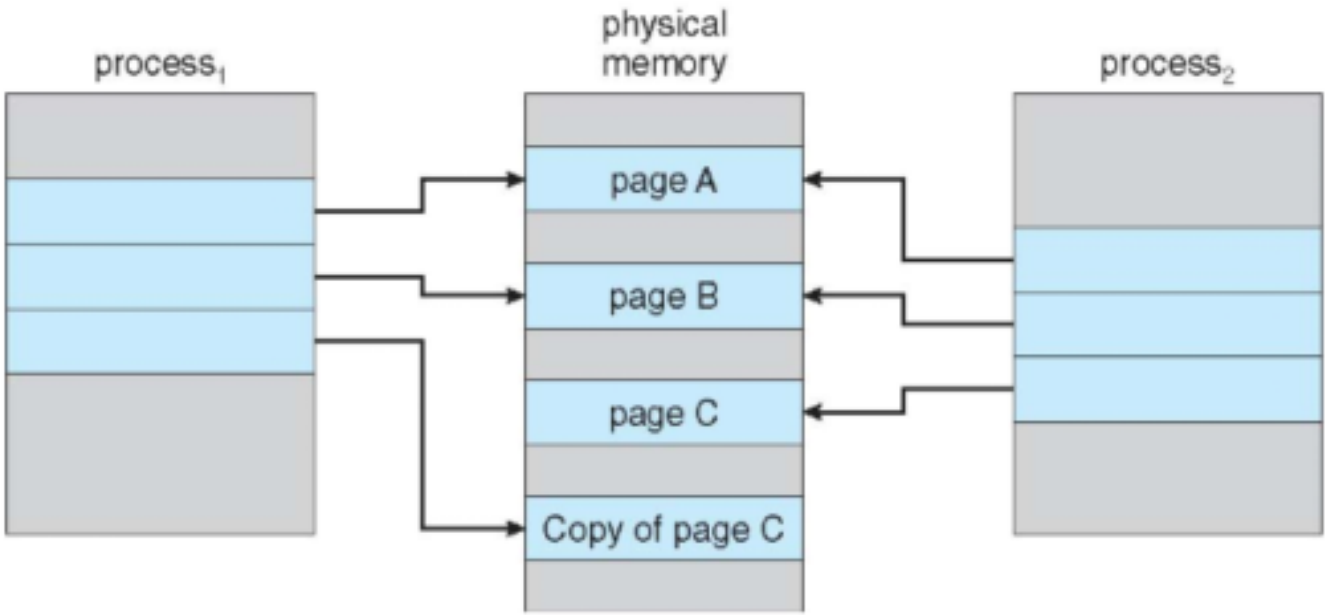
Copy-on-Write

- Copy-on-Write (COW) allows both parent and child processes to initially share the same pages in memory (If either process modifies a shared page, only then is the page copied)
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a pool of zero-fill-on-demand pages
  Pool should always have free frames for fast demand page execution (Don't want to have to free a frame as well as other processing on page fault)
- vfork(): With vfork(), the parent process is suspended, and the child process uses the address space of the parent. Because vfork() does not use copy-on-write, if the child process changes any pages of the parent's address space, the altered pages will be visible to the parent once it resumes. Therefore, vfork() must be used with caution to ensure that the child process does not modify the address space of the parent.

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



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

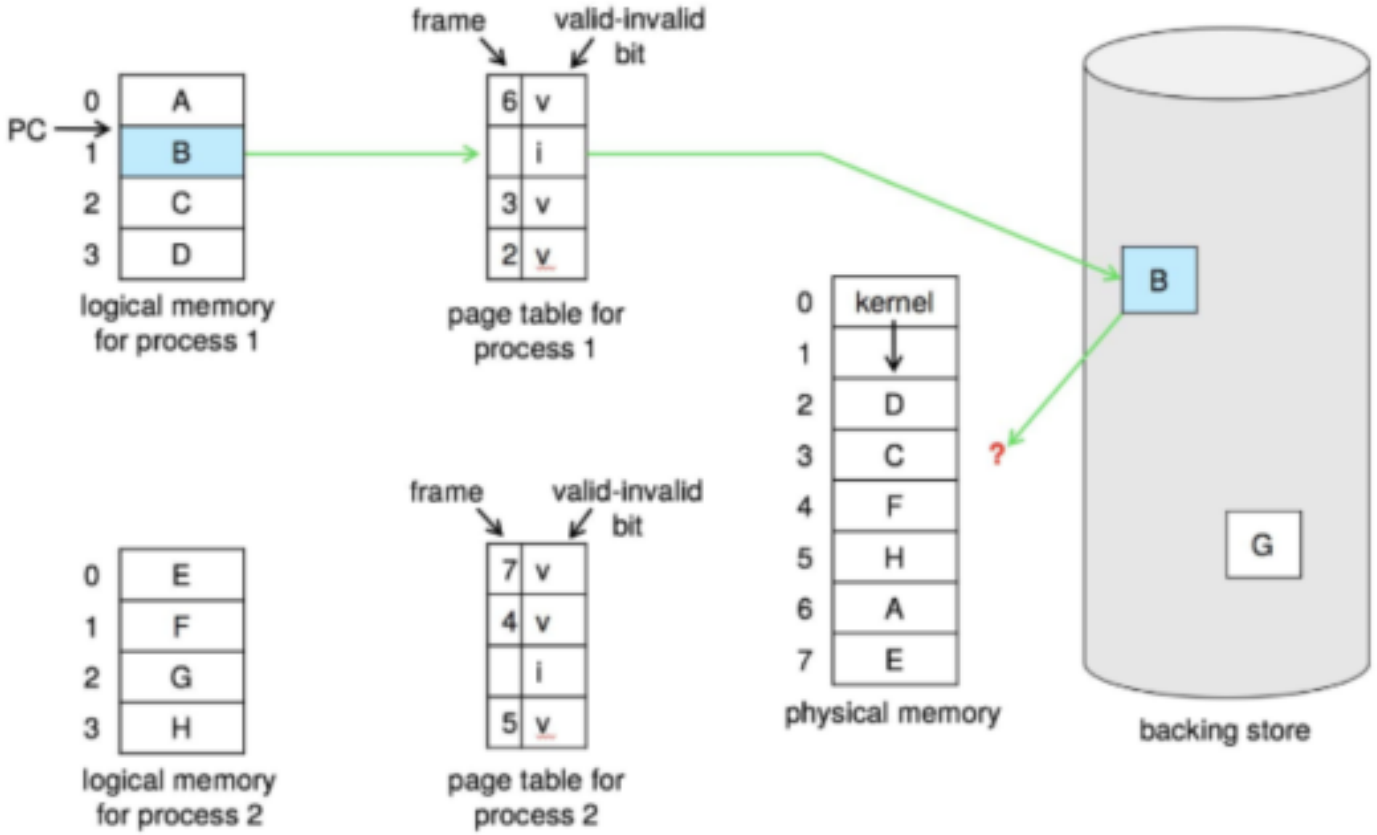
Page Replacement

- 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

Over-allocation of memory manifests itself as follows. While a process is executing, a page fault occurs. The operating system determines where the desired page is residing on secondary storage but then finds that there are no free frames on the free-frame list; all memory is in use.

Need For Page Replacement

This situation is illustrated in Figure, where the fact that there are no free frames is depicted by a question mark.



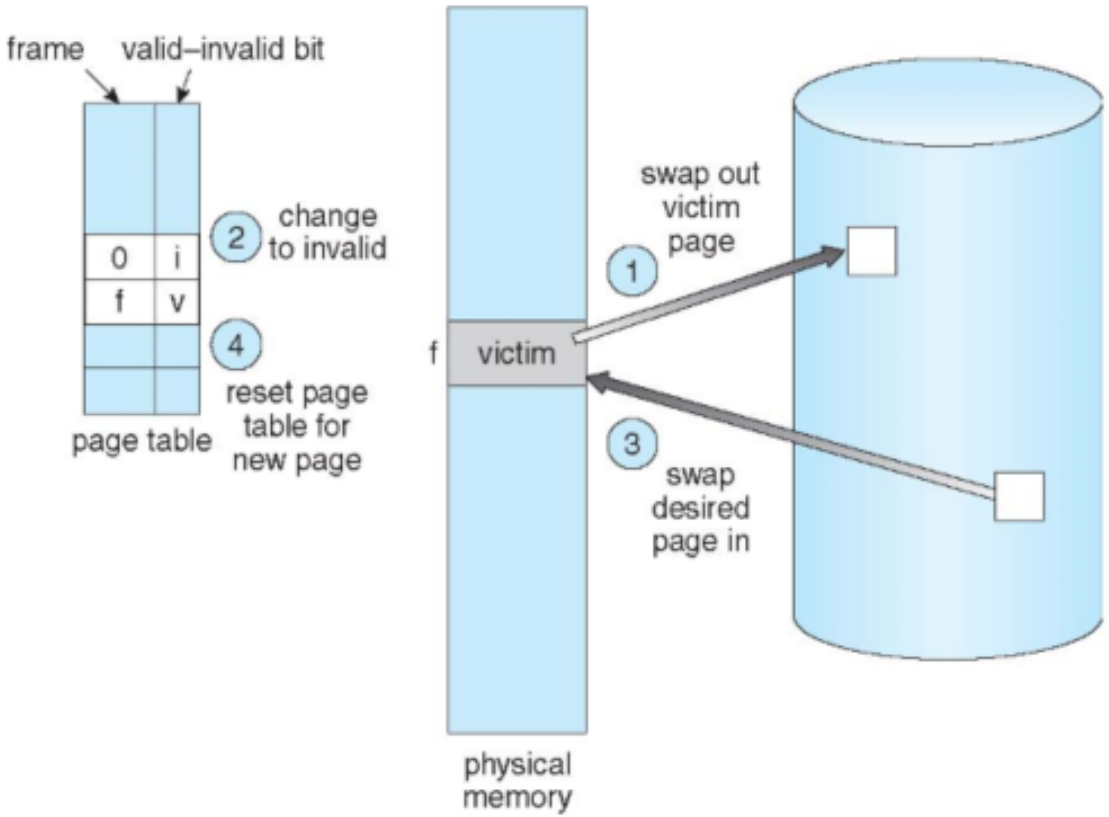
Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim frame**
   - Write victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap

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

Page Replacement

Page replacement takes the following approach. If no frame is free,we find one that is not currently being used and free it. We can free a frame by writing its contents to swap space and changing the page table (and all other tables) to indicate that the page is no longer in memory



Page and Frame Replacement Algorithms

Page replacement is basic to demand paging. It completes the separation between logicalmemory and physical memory.

Frame-allocation algorithm determines

- How many frames to give each process
- Which frames to replace

Page-replacement algorithm

- Want lowest page-fault rate on both first access and re-access

Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string

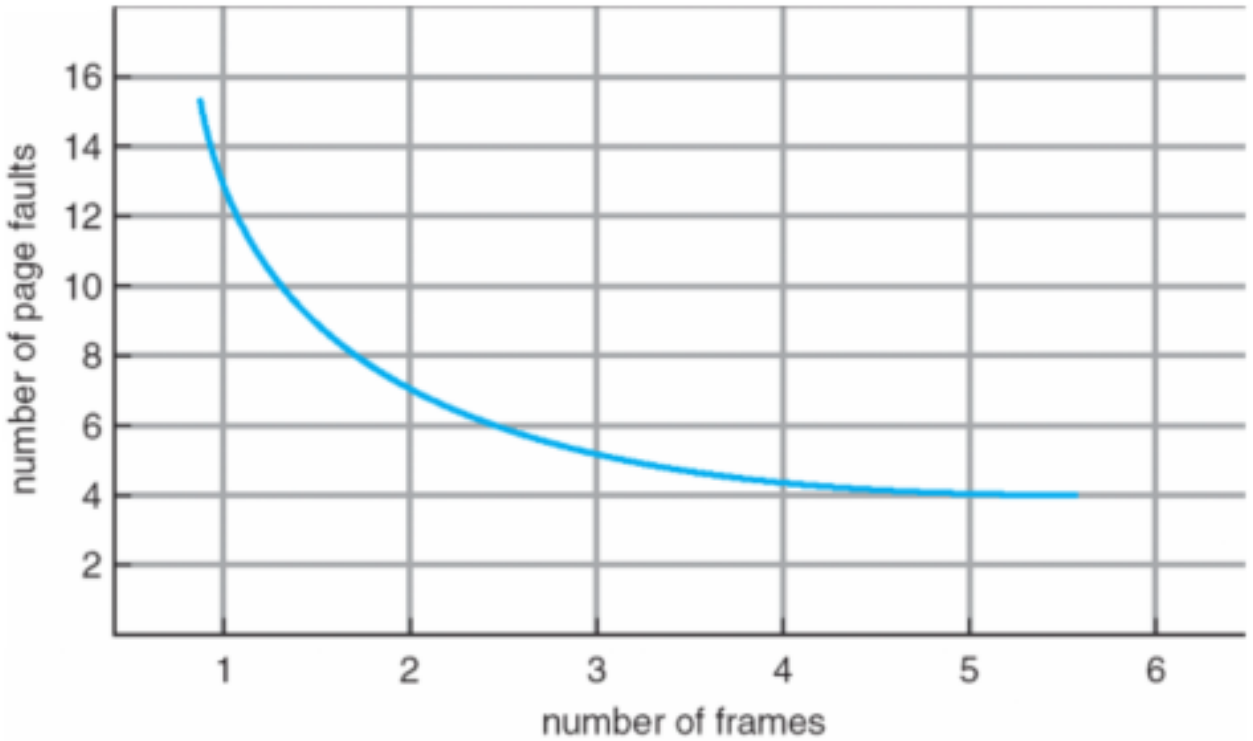
- String is just page numbers, not full addresses
- Repeated access to the same page does not cause a page fault
- Results depend on number of frames available

In all our examples, the **reference string** of referenced page numbers is

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

Graph of Page Faults Versus the Number of Frames

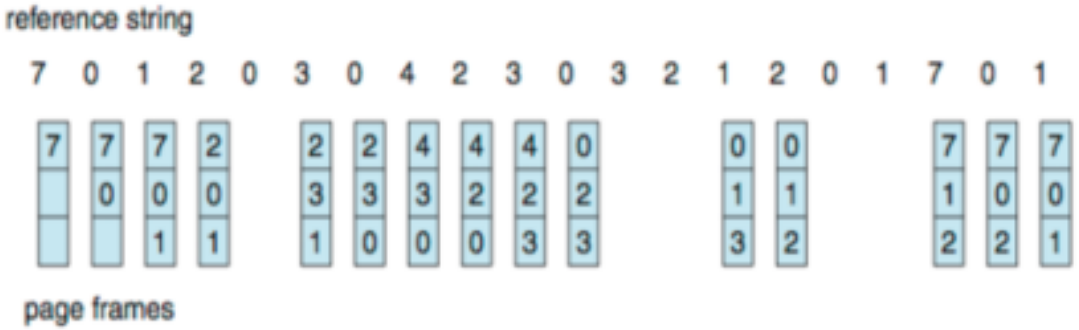
To determine the number of page faults for a particular reference string and page-replacement algorithm, we also need to know the number of page frames available. Obviously, as the number of frames available increases, the number of page faults decreases.



First-In-First-Out (FIFO) Algorithm

Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**

3 frames (3 pages can be in memory at a time per process)



15 page faults

Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5

Adding more frames can cause more page faults!

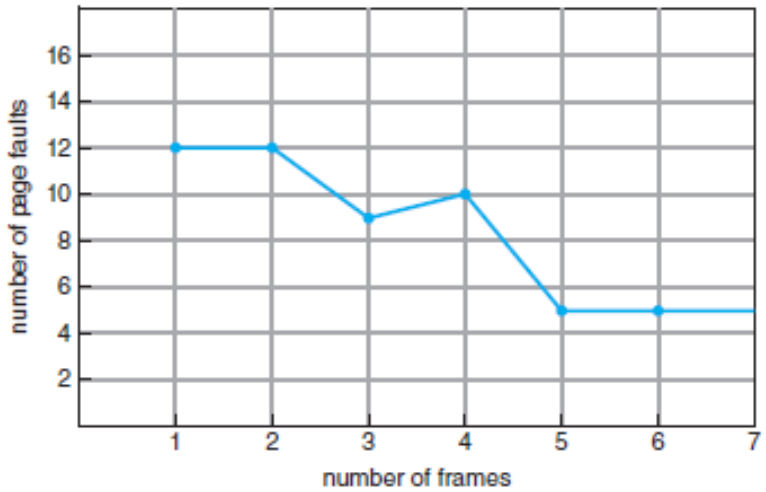
Belady' s Anomaly

How to track ages of pages?

Just use a FIFO queue

FIFO Illustrating Belady's Anomaly

Figure shows the curve of page faults for this reference string versus the number of available frames. Notice that the number of faults for four frames (ten) is greater than the number of faults for three frames (nine)! This most unexpected result is known as Belady's anomaly: for some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increases. We would expect that giving more memory to a process would improve its performance. In some early research, investigators noticed that this assumption was not always true. Belady's anomaly was discovered as a result.



Optimal Algorithm

One result of the discovery of Belady's anomaly was the search for an optimal page-replacement algorithm—the algorithm that has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly (called OPT or MIN)

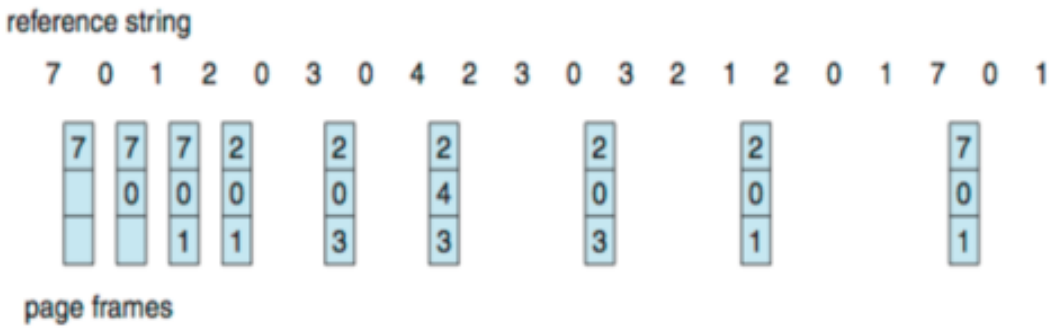
Replace page that will not be used for longest period of time

- 9 is optimal for the example

How do you know this?

- Can't read the future

Used for measuring how well your algorithm performs



Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string. (a similar situation with the SJF CPU-scheduling algorithm)

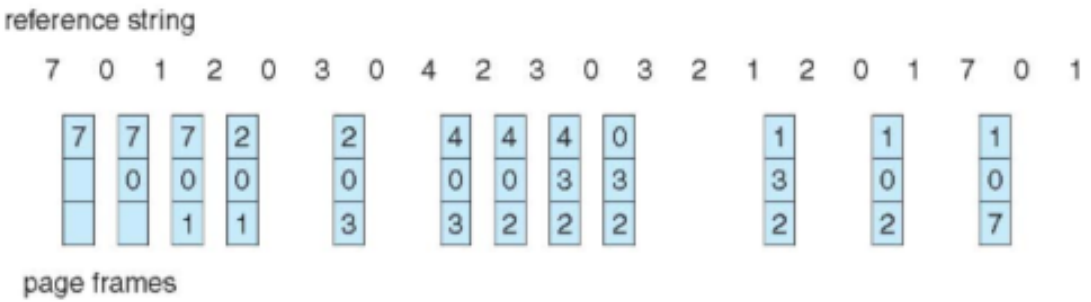
Least Recently Used (LRU) Algorithm I

If the optimal algorithm is not feasible, perhaps an approximation of the optimal algorithm is possible. The key distinction between the FIFO and OPT algorithms (other than looking backward versus forward in time) is that the FIFO algorithm uses the time when a page was brought into memory, whereas the OPT algorithm uses the time when a page is to be used.

Use past knowledge rather than future

Replace page that has not been used in the most amount of time

Associate time of last use with each page



12 faults – better than FIFO but worse than OPT

Generally good algorithm and frequently used

But how to implement?

Least Recently Used (LRU) Algorithm II

The LRU policy is often used as a page-replacement algorithm and is considered to be good. The major problem is how to implement LRU replacement. An LRU page-replacement algorithm may require substantial hardware assistance. Two implementations are feasible:

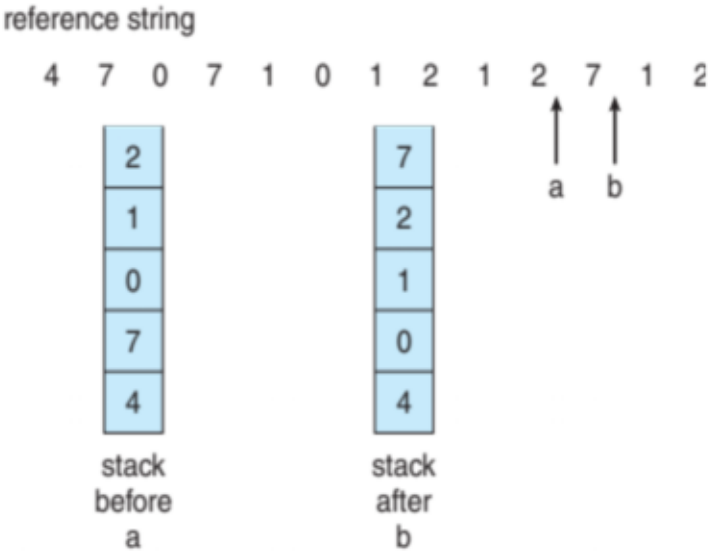
- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - Search through table needed
- Stack implementation
  - Keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - But each update more expensive
  - No search for replacement

LRU Algorithm III

Another approach to implementing LRU replacement is to keep a stack of page numbers. Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the most recently used page is always at the top of the stack, and the least recently used page is always at the bottom (Figure).

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

Use Of A Stack to Record Most Recent Page References



LRU Approximation Algorithms

- LRU needs special hardware and still slow
- Reference bit
  With each page associate a bit, initially = 0
  When page is referenced bit set to 1
  Replace any with reference bit = 0 (if one exists) -> We do not know the order, however
- 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

Allocation of Frames

- Each process needs minimum number of frames
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  instruction is 6 bytes, might span 2 pages
  2 pages to handle from
  2 pages to handle to
- Maximum of course is total frames in the system
- Two major allocation schemes
  fixed allocation
  priority allocation
- Many variations: We can require that the operating system allocate all its buffer and table space from the free-frame list. When this space is not in use by the operating system, it can be used to support user paging.We can try to keep three free frames reserved on the free-frame list at all times. Thus, when a page fault occurs, there is a free frame available to page into.

Fixed Allocation

Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames

Keep some as free frame buffer pool

Proportional allocation – Allocate according to the size of process

Dynamic as degree of multiprogramming, process sizes change

- s_i = size of process p_i
- $S = \sum s_i$
- m = total number of frames
- a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$m = 64$

$s_1 = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \times 62 \approx 4$

$a_2 = \frac{127}{137} \times 62 \approx 57$

Global vs. Local Allocation

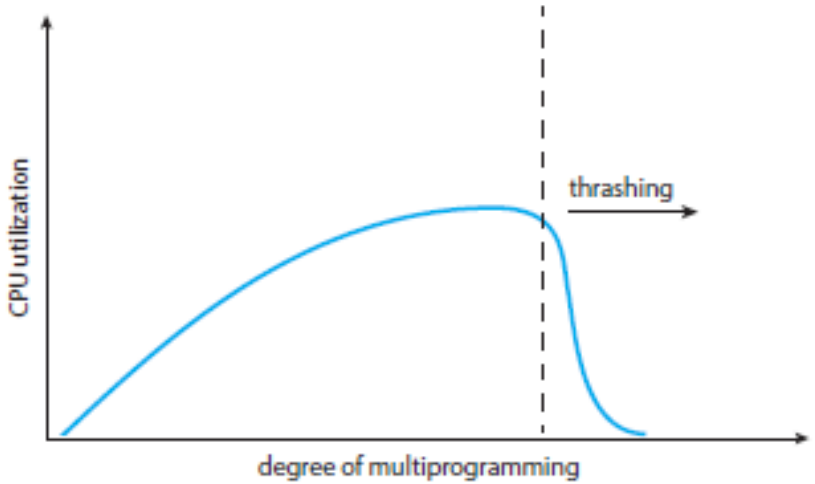
- Global replacement – process selects a replacement frame from the set of all frames; one process can take a frame from another
  But then process execution time can vary greatly
  But greater throughput so more common
- Local replacement – each process selects from only its own set of allocated frames
  More consistent per-process performance
  But possibly underutilized memory

Thrashing I

If a process does not have “enough” pages, the page-fault rate is very high

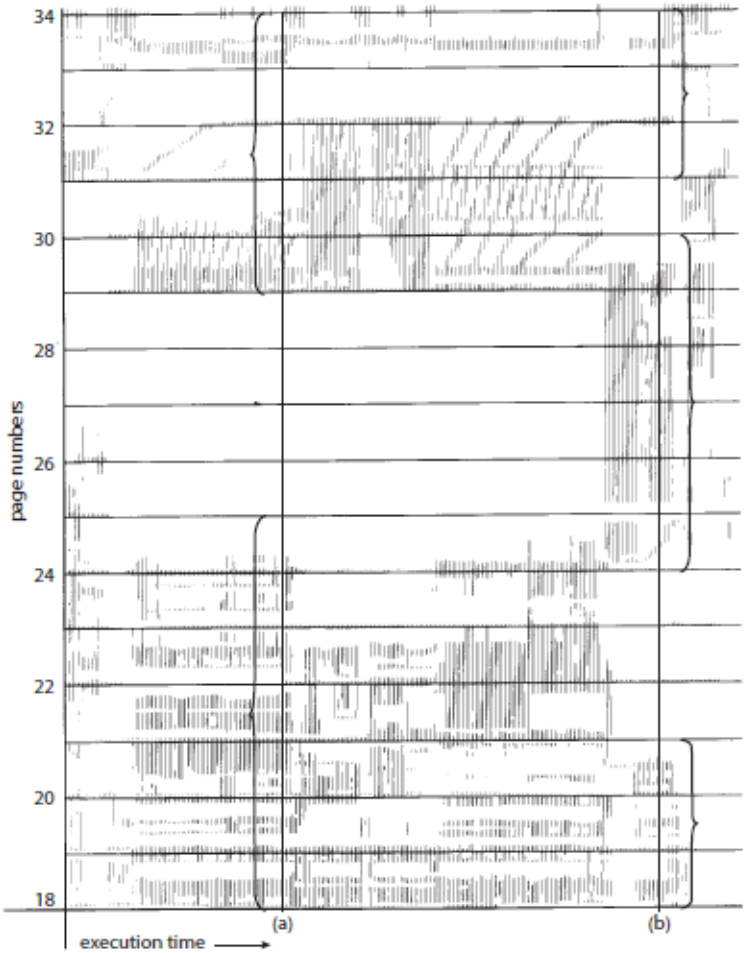
- Page fault to get page
- Replace existing frame
- But quickly need replaced frame back
- This leads to:
  - Low CPU utilization
  - Operating system thinking that it needs to increase the degree of multiprogramming
  - Another process added to the system

Thrashing II



Why does paging work?
Locality model
Process migrates from one locality to another.
Localities may overlap.
Why does thrashing occur?
$\sum$ size of locality > total memory size

Locality In A Memory-Reference Pattern



The working set strategy starts by looking at how many frames a process is actually using. This approach defines the locality model of process execution. A locality is a set of pages that are actively used together.

Working-Set Model I

$\Delta \equiv$ 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

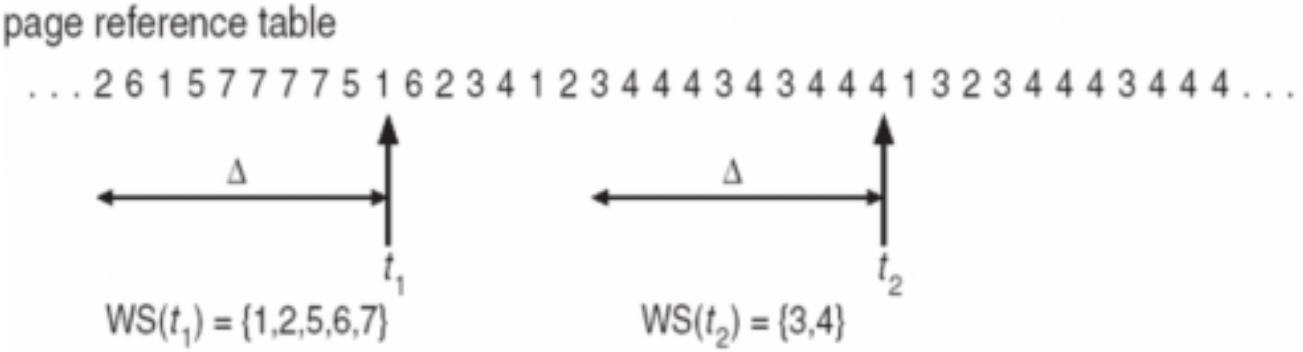
$D = \Sigma\ WSS_i \equiv$ total demand frames

- Approximation of locality

Working-Set Model II

if $D > m \Rightarrow$ Thrashing

Policy if $D >$ m, then suspend or swap out one of the processes

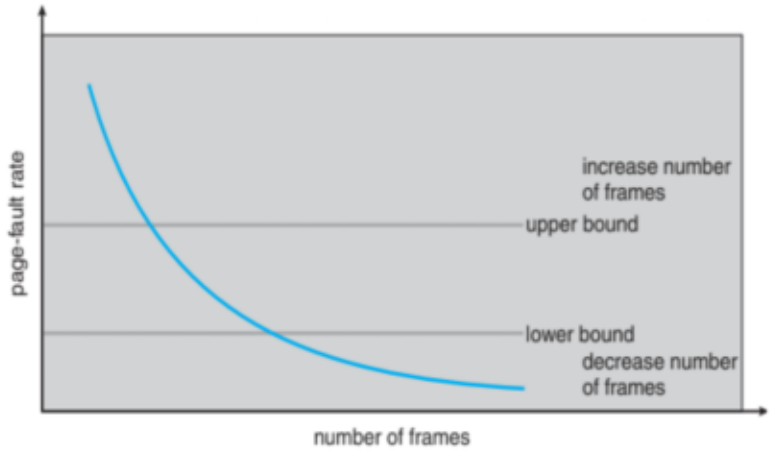


Page-Fault Frequency

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



An obvious property of pure demand paging is the large number of page faults that occur when a process is started. This situation results from trying to get the initial locality into memory. Prepaging is an attempt to prevent this high level of initial paging. The strategy is to bring some—or all—of the pages that will be needed into memory at one time.

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