Lecture 6b Virtual memory

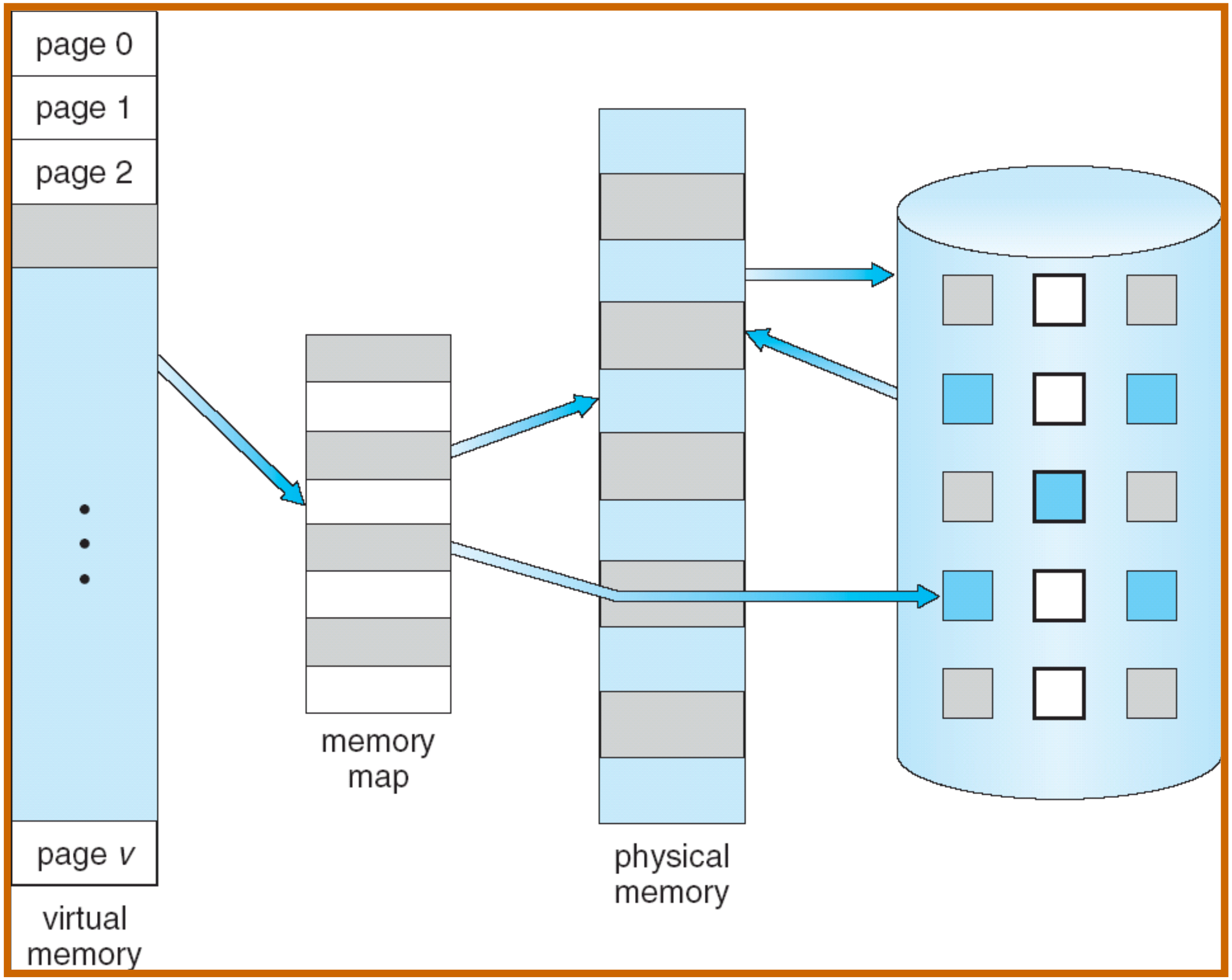
Content

1. Virtual memory concept
2. Paging on demand
3. Page replacement
4. Algorithm LRU and it’s approximation
5. Process memory allocation, problem of thrashing

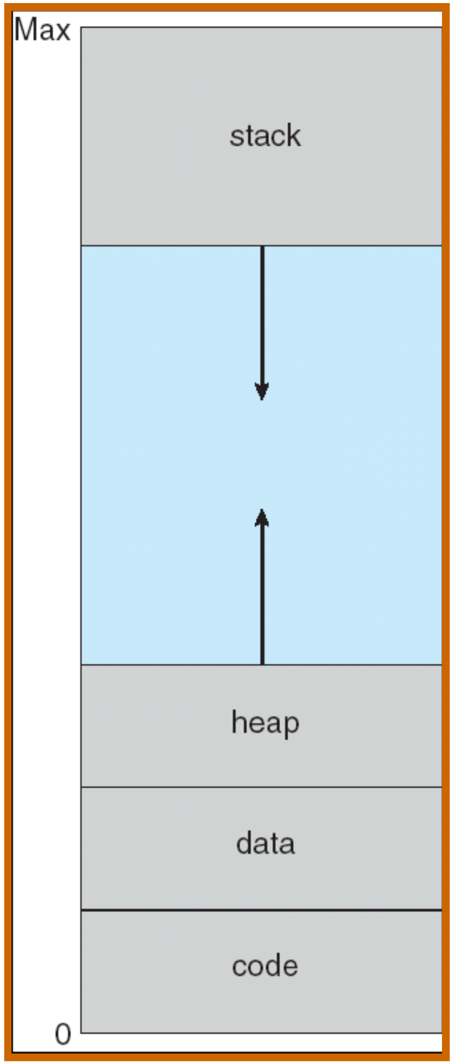
Virtual memory

- Virtual memory
  - Separation of physical memory from user logical memory space
  - 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.
- Synonyms
  - Virtual memory – logical memory
  - Real memory – physical memory

Virtual Memory That is Larger Than Physical Memory

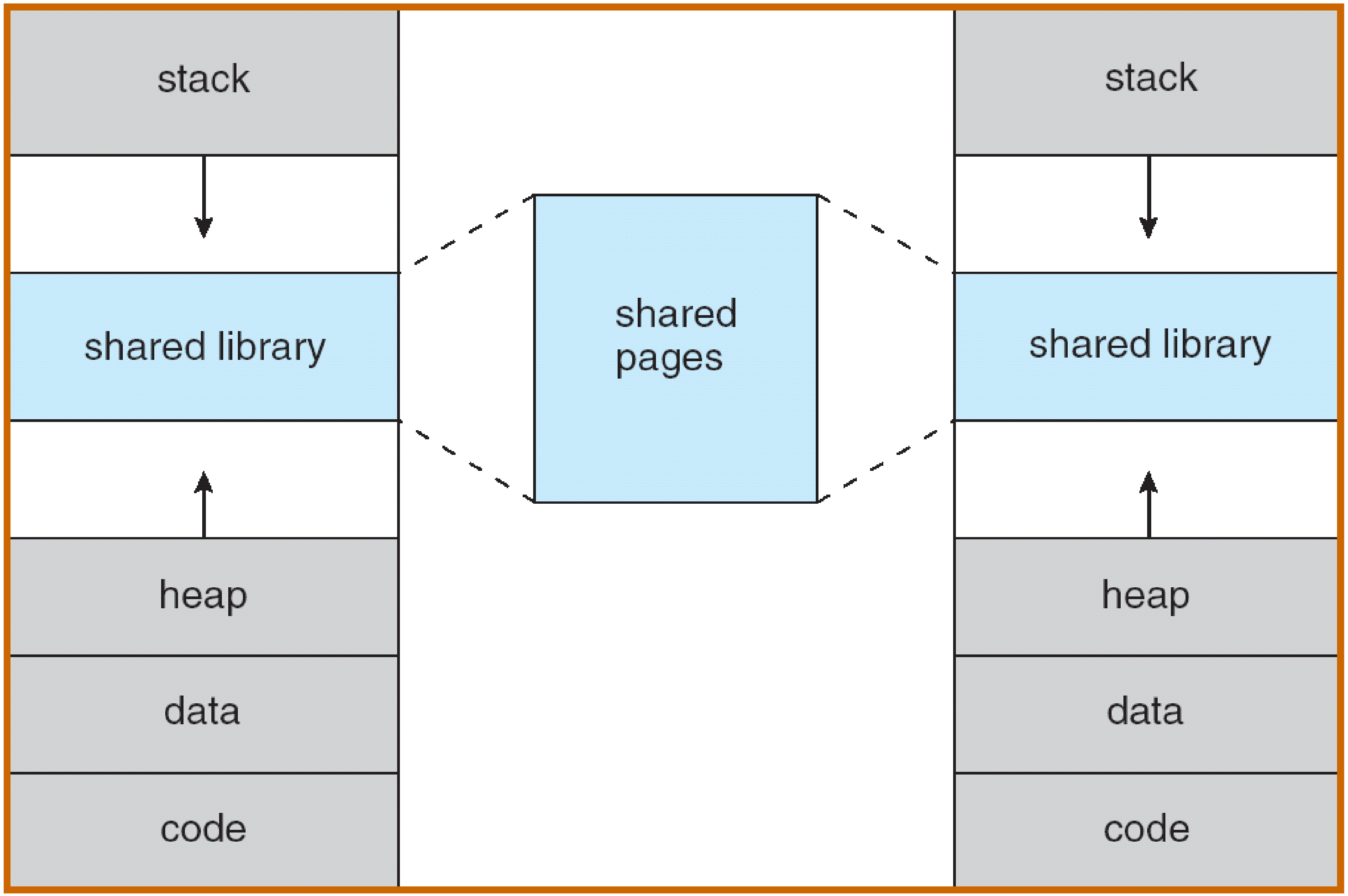


Virtual-address Space



- Process start brings only initial part of the program into real memory. The virtual address space is whole initialized.
- Dynamic exchange of virtual space and physical space is according context reference.
- Translation from virtual to physical space is done by page or segment table
- Each item in this table contains:
    - *valid/invalid* attribute – whether the page if in memory or not
    - *resident set* is set of pages in memory
    - reference outside resident set create *page/segment*

Shared Library Using Virtual Memory



Page fault

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

Frame #	valid-invalid bit
	1
	0
	1
	1
	0
	0
⋮	⋮
	1
	0

page table

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

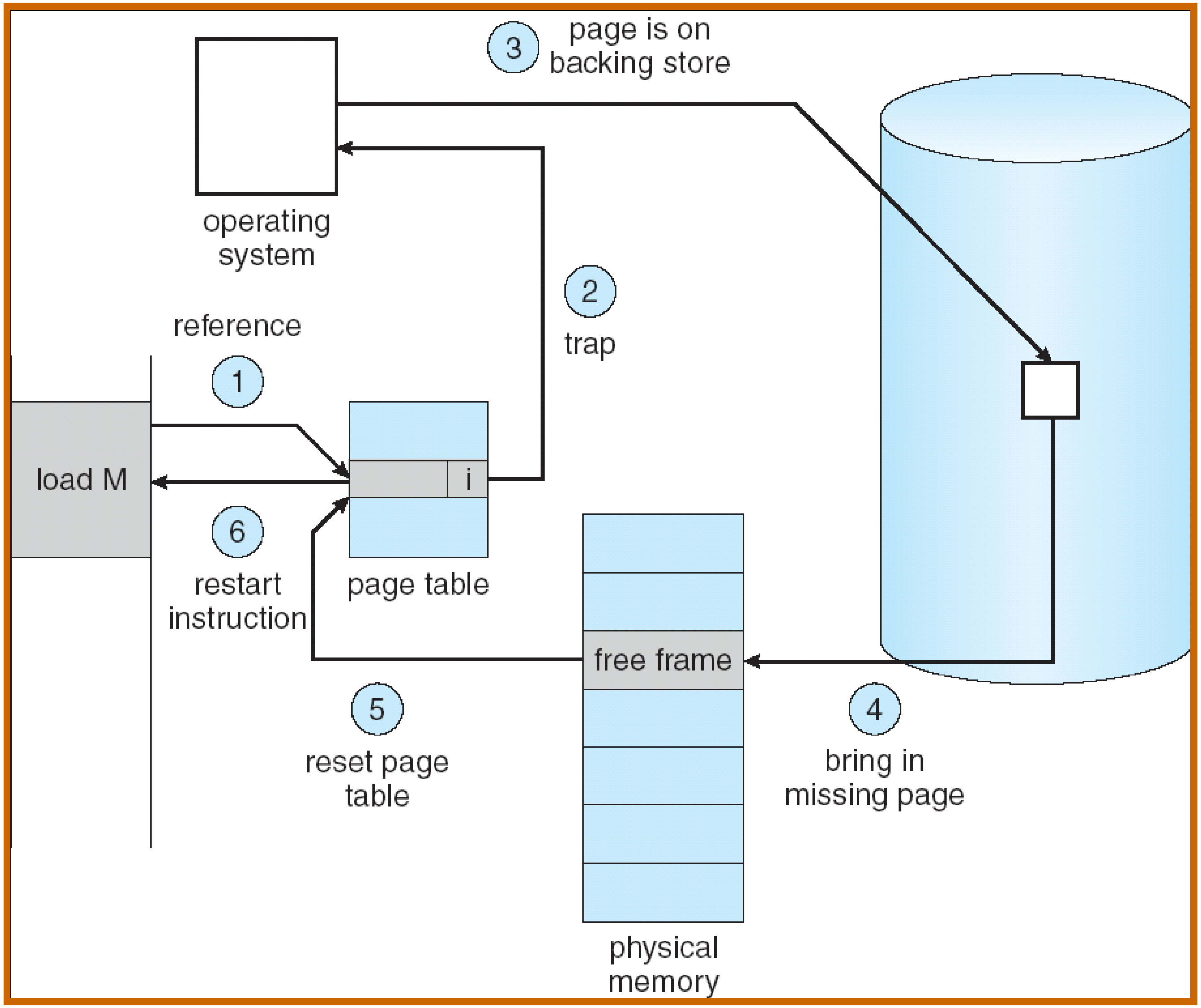
Paging techniques

- Paging implementations
  - *Demand Paging (Demand Segmentation)*
  - Lazy method, do nothing in advance
  - *Paging at process creation*
  - Program is inserted into memory during process start-up
  - *Pre-paging*
  - Load page into memory that will be probably used
  - *Swap pre-fetch*
  - With page fault load neighborhood pages
  - *Pre-cleaning*
  - Dirty pages are stored into disk

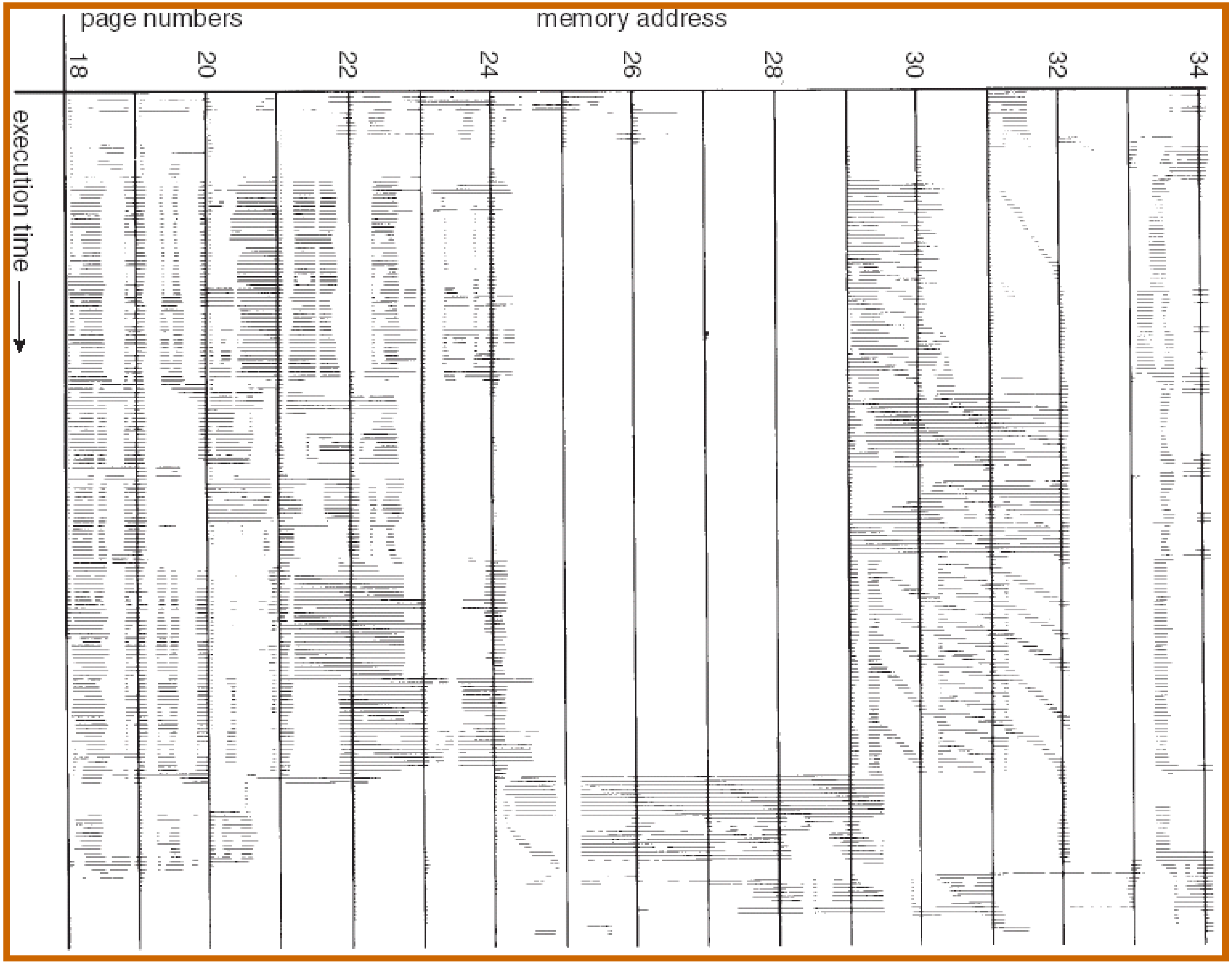
Demand Paging

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
  - Slow start of application
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ page fault ⇒ bring to memory
- Page fault solution
  - Process with page fault is put to waiting queue
  - OS starts I/O operation to put page into memory
  - Other processes can run
  - After finishing I/O operation the process is marked as ready

Steps in Handling a Page Fault



Locality In A Memory-Reference Pattern



Locality principle

- Reference to instructions and data creates clusters
- Exists time locality and space locality
  - Program execution is (excluding jump and calls) sequential
  - Usually program uses only small number of functions in time interval
  - Iterative approach uses small number of repeating instructions
  - Common data structures are arrays or list of records in neighborhoods memory locations.
- It's possible to create only approximation of future usage of pages
- Main memory can be full
  - First release memory to get free frames

Other paging techniques

- Improvements of demand paging
  - *Pre-paging*
  - Neighborhood pages in virtual space usually depend and can be loaded together – speedup loading
  - Locality principle – process will probably use the neighborhood page soon
  - Load more pages together
  - Very important for start of the process
  - Advantage: Decrease number of page faults
  - Disadvantage: unused page are loaded too
  - Pre-cleaning
  - If the computer has free capacity for I/O operations, it is possible to run copying of changed (dirty) pages to disk in advance
  - Advantage: to free page very fast, only to change validity bit
  - Disadvantage: The page can be modified in future - boondoggle

What happens if there is no free frame?

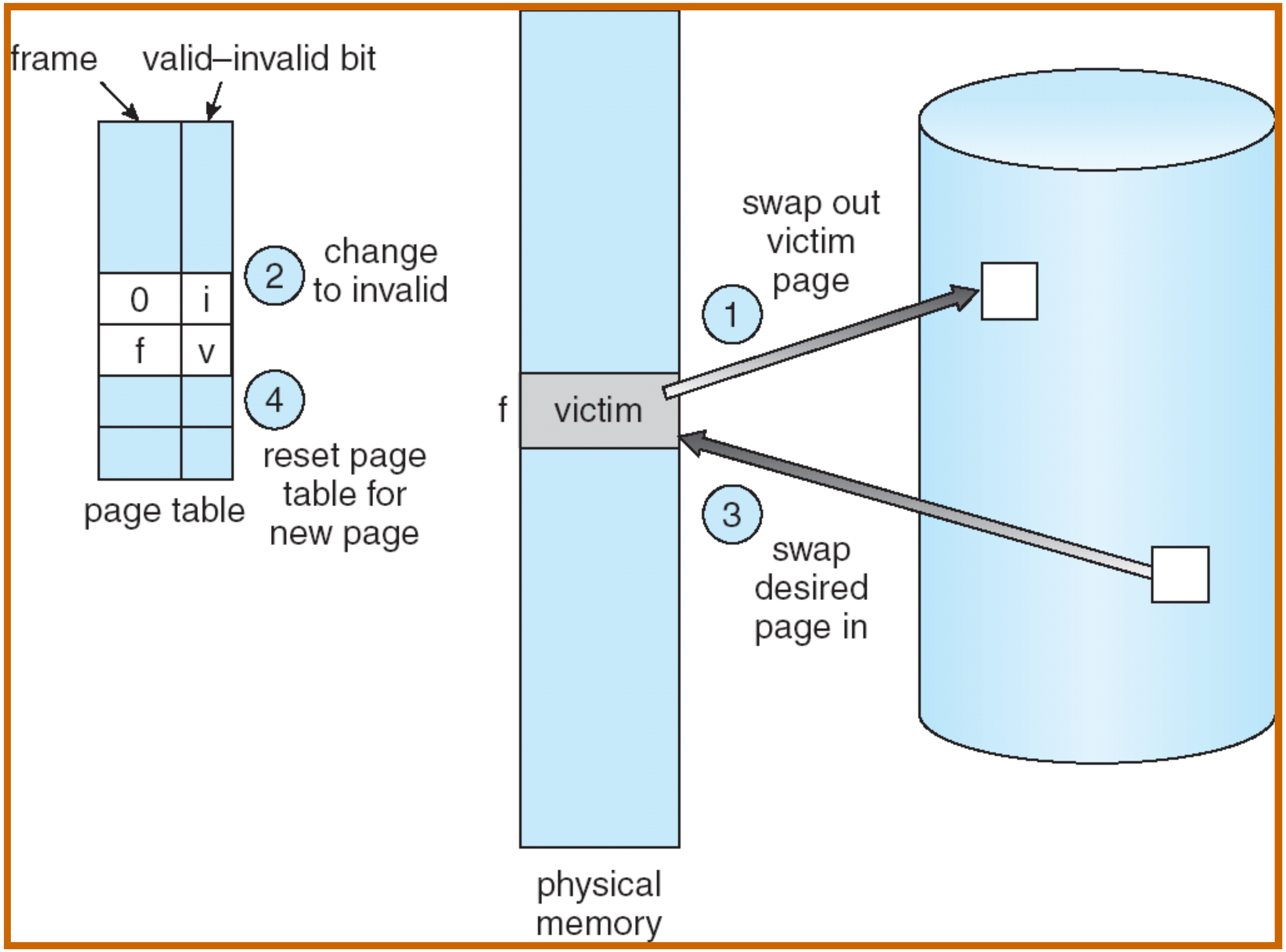
- Page replacement – find some page (victim) in memory, but not really in use, swap it out
  - algorithm
  - performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

Page Replacement

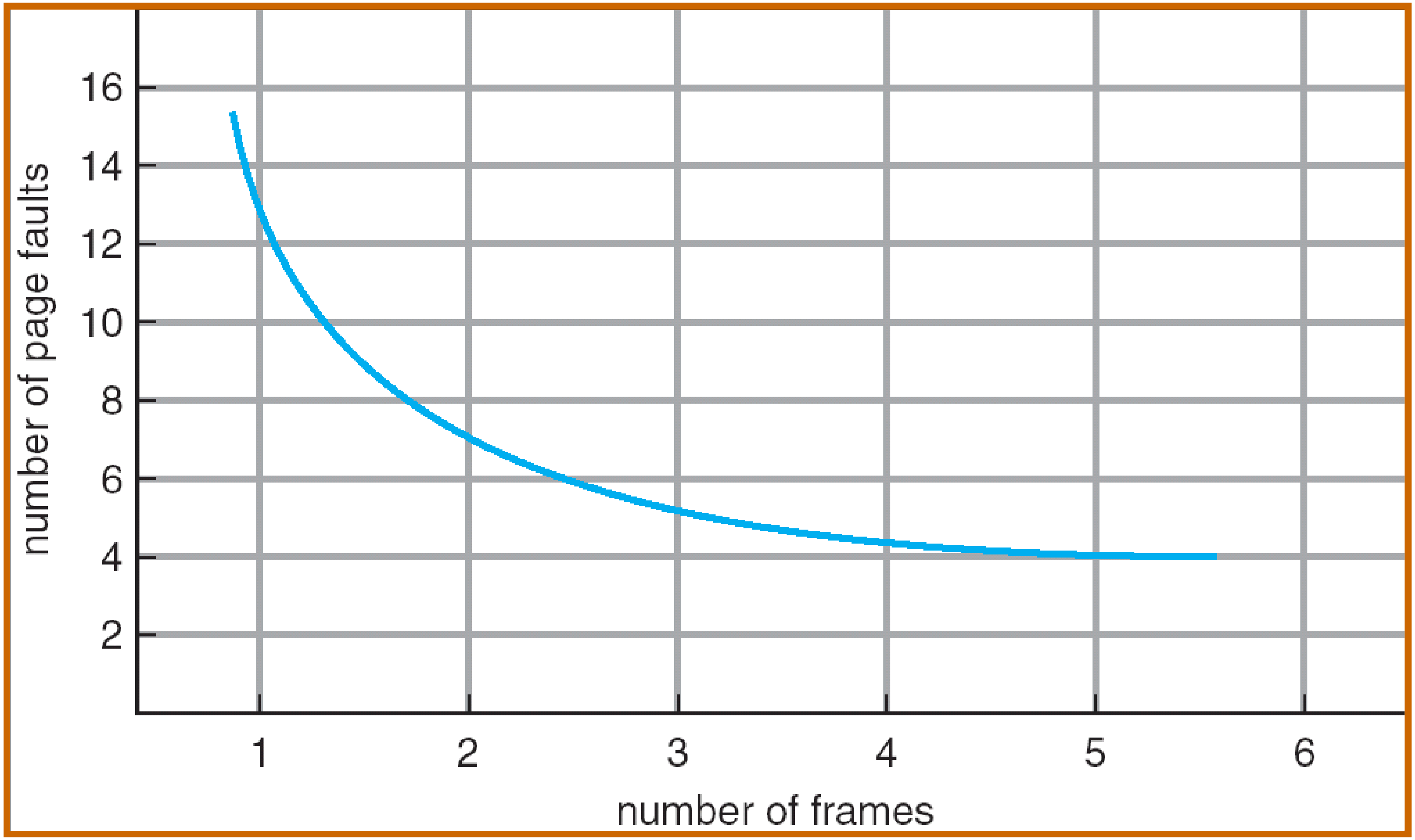
- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
- Some pages cannot be replaced, they are locked (page table, interrupt functions,...)
- Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk

- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory
- We want to have the 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

Page Replacement with Swapping



Graph of Page Faults Versus The Number of Frames



Algorithm First-In-First-Out (FIFO)

- 3 frames (memory with only 3 frames)

Reference:	1	2	3	4	1	2	5	1	2	3	4	5	
Frame number	Frame content												**Page faults**
1	**1**	1	1	**4**	4	4	**5**	5	5	5	5	5	
2		**2**	2	2	**1**	1	1	1	1	**3**	3	3	
3			**3**	3	3	**2**	2	2	2	2	**4**	4	**9 Page faults**

- 4 frames of memory

Reference:	1	2	3	4	1	2	5	1	2	3	4	5	
Frame number	Frame content												**Page faults**
1	**1**	1	1	1	1	1	**5**	5	5	5	**4**	4	
2		**2**	2	2	2	2	2	**1**	1	1	1	**5**	
3			**3**	3	3	3	3	3	**2**	2	2	2	**10 Page faults**
4				**4**	4	4	4	4	4	**3**	3	3	

  - *Beladyho anomalie* (more frames – more page faults)
- FIFO – simple, not effective
  - Old pages can be very busy

Optimal algorithm

- Victim – Replace page that will not be used for longest period of time
- We need to know the future
  - Can be only predicted
- Used as comparison for other algorithms
- Example: memory with 4 frames
  - As example we know the whole future

Reference:	1	2	3	4	1	2	5	1	2	3	4	5	
Frame number	Frame content												
1	**1**	1	1	1	1	1	1	1	1	1	**4**	4	**6 Page faults**
2		**2**	2	2	2	2	2	2	2	2	2	2	
3			**3**	3	3	3	3	3	3	3	3	3	(The best possible result)
4				**4**	4	4	**5**	5	5	5	5	5	

Least Recently Used

- Prediction is based on history
  - Assumption: Page, that long time was not used will be probably not used in future
- Victim – page, that was not used for the longest period
- LRU is considered as the best approximation of optimal algorithm
- Example: memory with 4 frames
- Best result 6 page faults, LRU 8 page faults, FIFO 10 page faults

Reference:	1	2	3	4	1	2	5	1	2	3	4	5	
Frame number	Frame content												**Page faults**
1	**1**	1	1	1	1	1	1	1	1	1	1	**5**	
2		**2**	2	2	2	2	2	2	2	2	2	2	
3			**3**	3	3	3	**5**	5	5	5	**4**	4	**8 Page faults**
4				**4**	4	4	4	4	4	**3**	3	3	

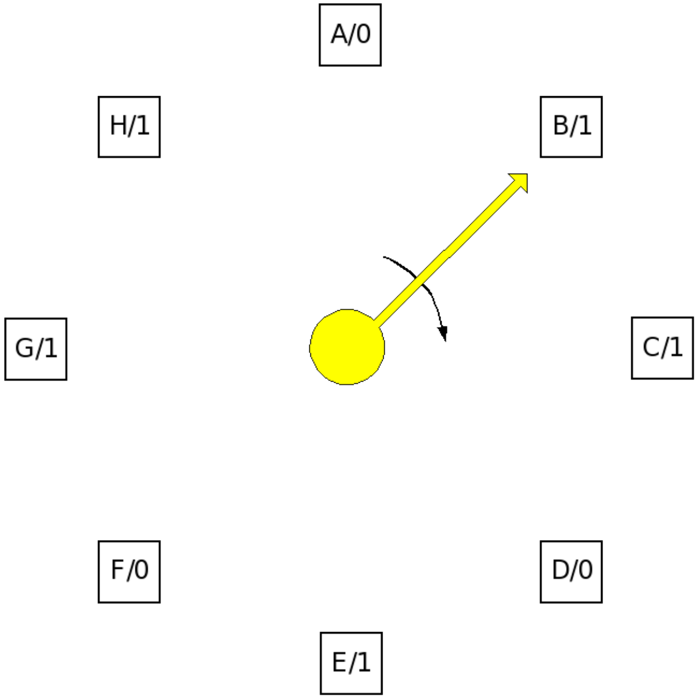
LRU – implementation

- It is not easy to implement LRU
  - The implementation should be fast
  - There must be CPU support for algorithm – update step cannot be solved be SW because is done by each instruction (each memory reading)
- 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
- 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
  - No search for replacement

Approximation of LRU

- 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
  - In fact it is FIFO with second chance

Algorithm Second Chance



Page fault test the frame that is pointed by clock arm.

Depend on access *a*-bit:

- if *a*=0:

  take this page as victim

- if *a*=1:

  turn *a*=0, and keep page in memory

  turn the clock arm forward

- if you have no victim do the same for the next page

  - Numerical simulation of this algorithm shows that it is really close to LRU

Modification LRU

- NRU – not recently used
  - Use *a*-bit and dirty bit *d*-bit
  - Timer regularly clean a-bit and therefore it is possible to have page with d-bit=1 and a-bit=0.
  - Select page in order (*da*): 00, 01, 10, 11
  - Priority of *d*-bit enable to spare disk operation and time
- Ageing
  - a-bit is regularly saved and old-values are shifted
  - Time window is limited by HW architecture
  - If the history of access to page is 0,0,1,0,1, then it corresponds to number 5 (00101)
  - The page with the smallest number well be removed

Counter algorithms

- Reference counter
  - Each frame has reference counter
  - For „*swap-in*“ – the counter is set to 0
  - Each reference increments the counter
- Algorithm LFU (*Least Frequently Used*)
  - replaces page with smallest count
- Algorithm MFU (*Most Frequently Used*)
  - based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Processes and paging

- **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

- Principles of frame allocation
  - Fixed allocation
  - Process receives fixed number of frames (Can be fixed for each process or can depends on it’s virtual space size)
  - Priority allocation
  - Process with higher priority receives more frames to be able to run faster
  - If there is page fault process with higher priority gets frame from process with lower priority

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
- Example:

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

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

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

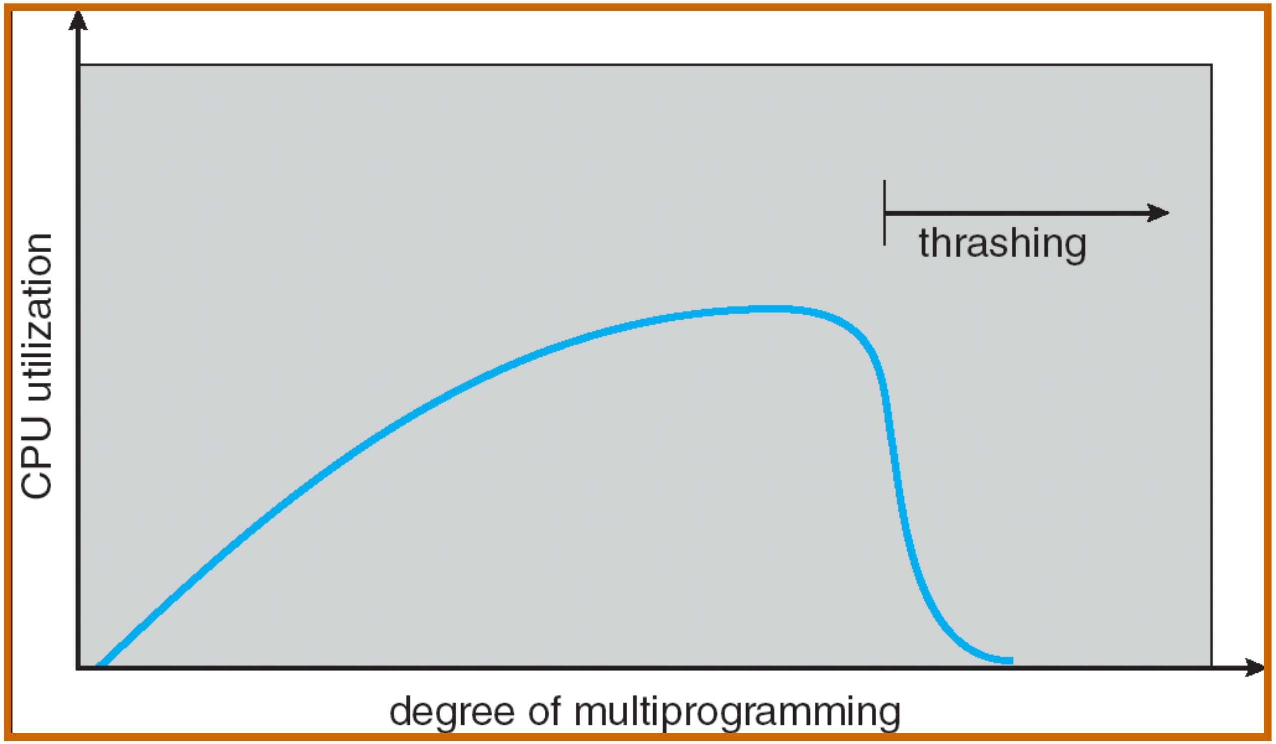
$$a_2 = \frac{127}{137} \times 64 \approx 59$$

Dynamic Allocation

- Priority allocation
  - Use a proportional allocation scheme using priorities rather than size
  - If process *Pi* generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number
- Working set
  - Dynamically detect how many pages is used by each process

Thrashing

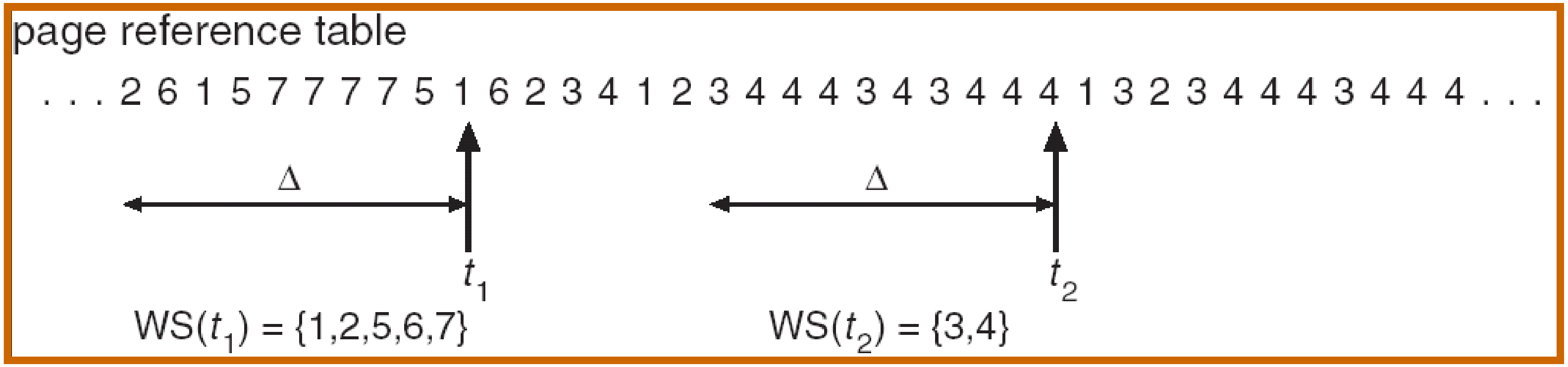
- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system thinks that it needs to increase the degree of multiprogramming
  - another process can be added to the system
- **Thrashing** ≡ a process is busy swapping pages in and out



Working-Set Model

- How many pages process need?
- Working set define set of pages that were used by last N instructions
- Detection of space locality in process
- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instruction
- *WSSi* (working set of Process *Pi*) =
  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
- $D = \Sigma\, WSSi \equiv$ total demand frames
- if $D > m \Rightarrow$ Thrashing
- Policy if D > m, then suspend one of the processes

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 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

Working set

- If sum of working sets for all process Pi- *WSi* exceeds the whole capacity of physical memory it creates *thrashing*

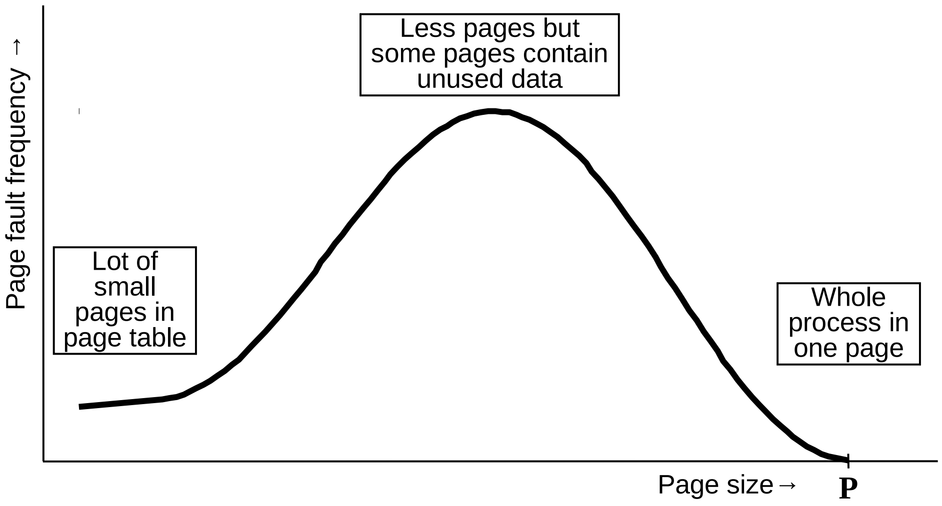
- Simply protection before thrashing
  - Whole one process is swapped out

Page Fault Frequency - PFF

- PFF is a variable-space algorithm that uses a more *ad hoc* approach
- Attempt to equalize the fault rate among all processes, and to have a “tolerable” system-wide fault rate
  - Monitor fault rate for each process
  - If fault rate is above given threshold, give it more memory, so that it faults less
  - If fault rate is bellow threshold, take away memory, so should fault more, allowing someone else to fault less

Page size

- Big pages
  - Small number of page faults
  - Big fragmentation
  - If page size is bigger
    then process size,
    virtual space is not necessary



- Small pages
  - Big number of small pages
  - Page is more frequently in memory → low number of page faults
  - Smaller pages means
  - Smaller fragmentation but decrease the effectivness of disk operations
  - The bigger page table and more complicated selection of victim for swap out
  - Big page table
  - PT must be in memory, cannot be swaped out – PT occupying real memory
  - Placing part of PT into virtual memory leads to more page faults (access to invalid page can create 2 page faults, first fault of page table and fault of page)

Programming techniques and page faults

- Programming techniques have influence to page faults

  double data[512][512];

  – Suppose that double occupy 8 byts

  – Each line of array has 4 KB and is stored in one page 4 KB

  – It is good to know how the data are stored in virtual space

Approach 1:

```
for (j = 0; j <512; j++)
        for (i = 0; i < 512; i++)
                data[i][j] = i*j;
```

Can have
512 x 512 = 262 144
page faults

Approach 2:

```
for (i = 0; i <512; i++)
        for (j = 0; j < 512; j++)
                data[i][j] = i*j;
```

Only 512 page faults

Paging in Windows XP

- Uses demand paging with pre-paging **clusters**. 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
- There can be thrashing
  - Recommended minimal memory size – 128 MB
  - Real minimal memory size – 384 MB