# Virtual Memory Software Issues

Chapter 9

Page replacement and other policies

## Operating System Involvement with Paging

Four times when OS involved with paging

1. Process creation
   - determine program size
   - create page table
2. Process execution
   - Information needed by MMU reset for new process on context-switch
   - TLB flushed
3. Page fault time
   - determine virtual address causing fault
   - figure out where on disk the page is stored
   - Determine a page in memory to be replaced
   - swap target page out, needed page in
4. Process termination time
   - release page table, pages, etc for the terminated process.

## Paging Policies

- Fetch strategies (when to fetch pages)
- Placement strategies (in what page frame to put the new page)
- Replacement strategies (if there are no free page frames, which existing frame to replace)

## Fetch Strategies

- When should a page or segment be brought into primary (main) memory from secondary (disk) storage?
  - Demand fetch (wait until you need it)
  - Anticipatory fetch (predict which page will be needed)
    - Hard to do in practice ➔ not done. Demand paging more popular and is what is used in popular OS's.

# Demand Paging

- Principle: never bring a page into primary memory until it is needed
- Bring a page into memory only when it is needed.
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users at any one time
- Page is needed $\Rightarrow$ reference to it
  - invalid reference $\Rightarrow$ abort
  - Valid reference but not-in-memory $\Rightarrow$ bring to memory

Unix, linux
Windows NT
etc implement this

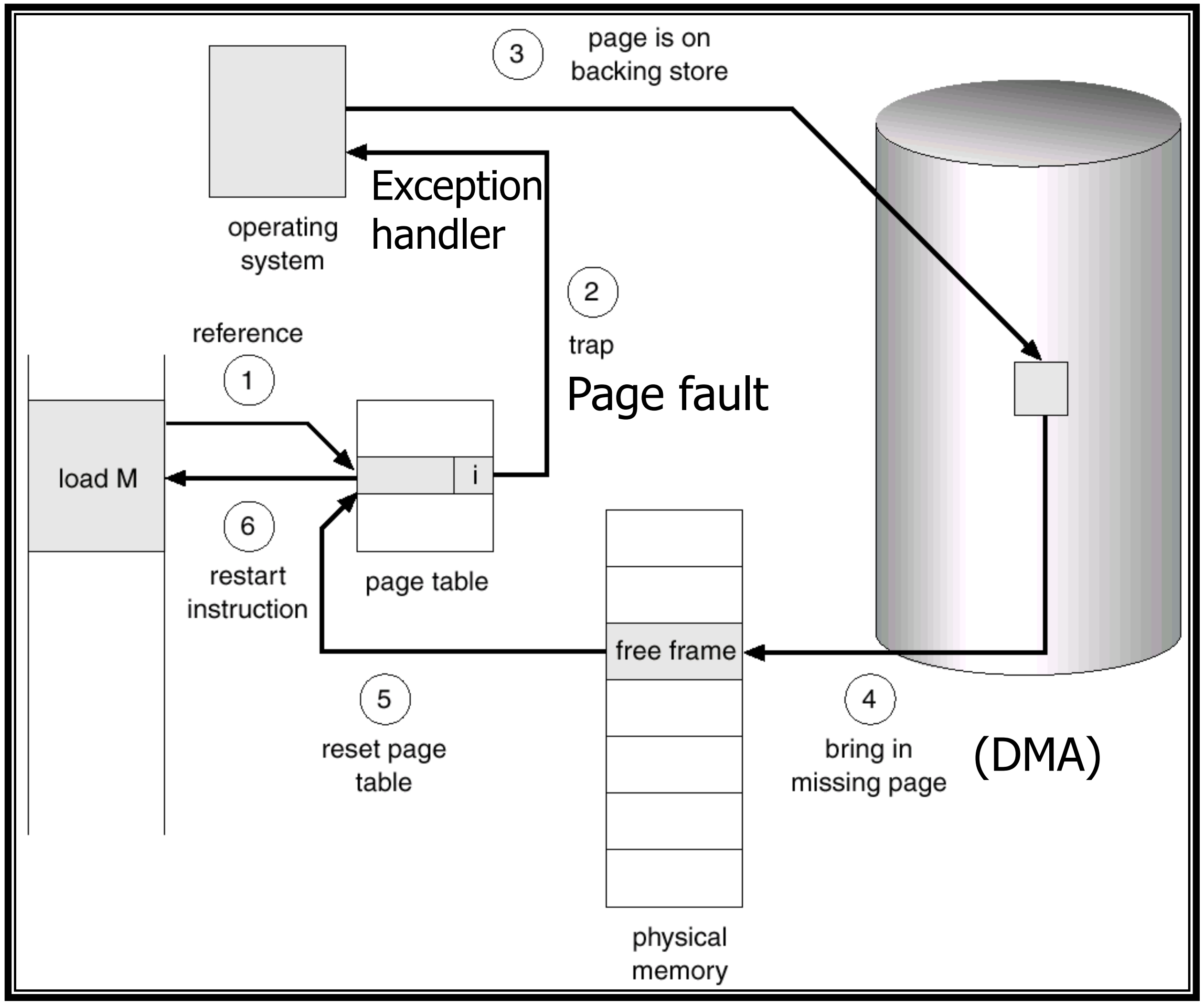
## Page Fault Handling

1. Hardware traps to kernel
2. General registers saved
3. OS determines which virtual page is needed
4. OS checks validity of address, seeks page frame (page (re)placement policy)
5. If selected frame to replace is dirty (modified), write it to disk

# Page Fault Handling

6. OS schedules a disk operation to bring new page in from disk
7. After transfer is complete (indicated by an interrupt), page tables are updated
8. Faulting instruction backed up to when it began (complications can arise: see following slides)
9. Faulting process scheduled (from suspended queue to ready queue)
10. Registers restored (context switch when process is scheduled)
11. Program continues from instruction that generated page fault.

# Steps in Handling a Page Fault

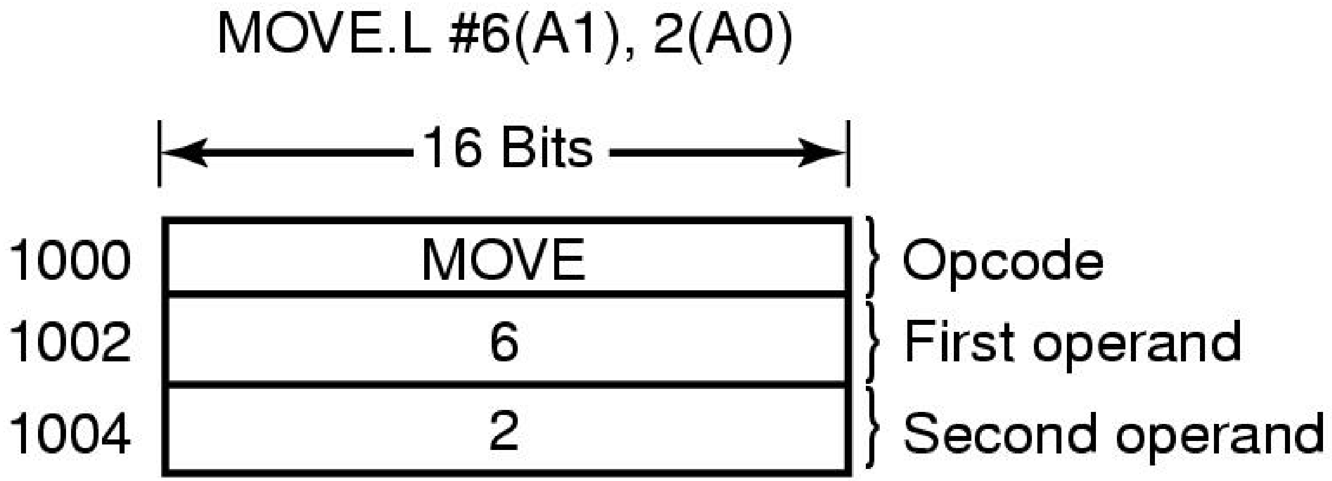


## Restarting an instruction

- Restart any instruction after page fault
  - Lots of issues here
  - page faults occur **in the middle** of instructions
  - With multi-word length instructions, it gets harder to determine where to restart unless the hardware helps you.

## Restarting an instruction

- On some processors, the hardware interrupt handling will save the PC for the instruction that caused the page-fault in a special register (EPC). (e.g., MIPS)
- On other processors, instructions may be more than one word, and it may be harder to determine what address the restart instruction is:



## Restarting an instruction

- Restart any instruction after page fault
  - In pipelined architecture, the CPU is executing a number of instructions in various stages at any given time.
    - Need to handle this properly, determine which instruction to restart, what state of the computation to save.

## Restarting an instruction

- If instruction operands overlap page boundary: (CISC instructions such as multi-byte memcopies)
  - Example: copy 4000 bytes from x to y.
  - Can cross page boundaries
- Either
  - Probe ends of operand (check if pages are in memory first, then start executing instruction; lock the pages in memory during this process)
  - Or save data changed so can undo changes if fault

## Performance of Demand Paging

- $p$ probability of page fault, $ma$ memory access time, $f$ page fault processing time
- Effective access time:
  - $t_{\text{eff}} = (1 - p) \times ma + p \times f$

# Performance of Demand Paging

An example machine and numbers:

| | |
|---|---|
| Memory access time: | 500 nsecs |
| Trap to operating system: | 88 µsecs |
| Issue and wait for disk read: | 15 msecs |
| Process disk interrupt: | 88 µsec |
| Update page table: | 10 µsecs? |
| Reschedule process: | 10 µsecs? |

- Wait for rescheduled process to run = number of processes in run queue × time slice $\approx 1800\ \mu secs$

## Performance of Demand Paging

- Effective access time:
  - $t_{eff} = (1 - p) \times 0.5 + p \times 17{,}000.5\ \mu secs \approx 17\ msec$

# Performance of Demand Paging

Some scenarios:

- Assume, hit rate (1 page fault out of 1000 pages)
- $p = 1/1000$, $t_{eff}$ = 17.5 μsecs (compared to 0.5 μsecs)
  - almost 35 time slower!! $\Rightarrow$ NOT GOOD
  - several things can be improved:
    - page size can be increased
    - compiler can optimize for paging

## Performance of Demand Paging

Another way to view this situation:

- What kind of page fault rate do we need if we want to keep the degradation to no more than 10 percent?
  - i.e., if we want $t_{\text{eff}} = 0.5 + 0.1 \times 0.5 = 0.55\mu sec$
- If we plug these numbers into the $t_{eff}$ formula, we get approximately 1 page fault out of 300,000 accesses.
- These are more typical and realistic page fault rates.
  - This would be our target for page fault rates!

## Placement Strategies

- When a page or segment is brought into primary storage, where is it to be put?
  - Paging – trivial: any free page frame will do.
  - Segmentation – significant problem
    - may not have room, fragmentation problems
    - may have to compact memory

## Replacement Strategies

- When some other page or segment is to be brought in and there is not enough room:
  - Which page or segment now in primary storage is to be removed from primary storage?
    - Heuristic: replace the one you don’t need.
    - How to implement? Many different algorithms.

## 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. Write the selected victim page to the disk if necessary and update any necessary tables
4. Read the desired page into the (newly) freed frame. Update the page and frame tables.
5. Restart the process.

## Page Replacement considerations

- Page fault forces choice
  - which page must be removed
  - make room for incoming page
- Modified page must first be saved
  - unmodified just overwritten
- Better not to choose an often used page
  - will probably need to be brought back in soon

## 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.
- Reference string consists of the virtual page numbers generated by memory references.
- In all our examples, the reference string is

  A, B, C, D, A, B, E, A, B, C, D, E

  or

  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 (same thing numerically.

## Page Replacement Issues

- If no frames are free, a disk read and write of page is required
- If the page to be replaced has not been changed since it was read in, it can be discarded and does not need to be copied out to secondary storage
- Read only pages are never written but may be shared!
- A dirty bit can be set in the page table by hardware to indicate that the page has been modified

## Page Replacement Strategies

- The principle of optimality
  - Replace the page that will not be used again the farthest time in the future

<u>Non-usage based replacement strategies:</u>

- Random page replacement (easiest)
  - Choose a page randomly
- FIFO - first in first out
  - Replace the page that has been in primary memory the longest

# Page Replacement Strategies

Usage based replacement strategies:

- LRU - least recently used
  - Replace the page that has not been used for the longest time
- LFU - least frequently used
  - Replace the page that is used least often
- NRU – not recently used (page classes)
- NFU – not frequently used (aging)
  - An approximation to LRU
- Second chance, clock algorithm, and variations.
- Working set (actually used in real computers)
  - Keep in memory those pages that the process is actively using

## Principle of Optimality

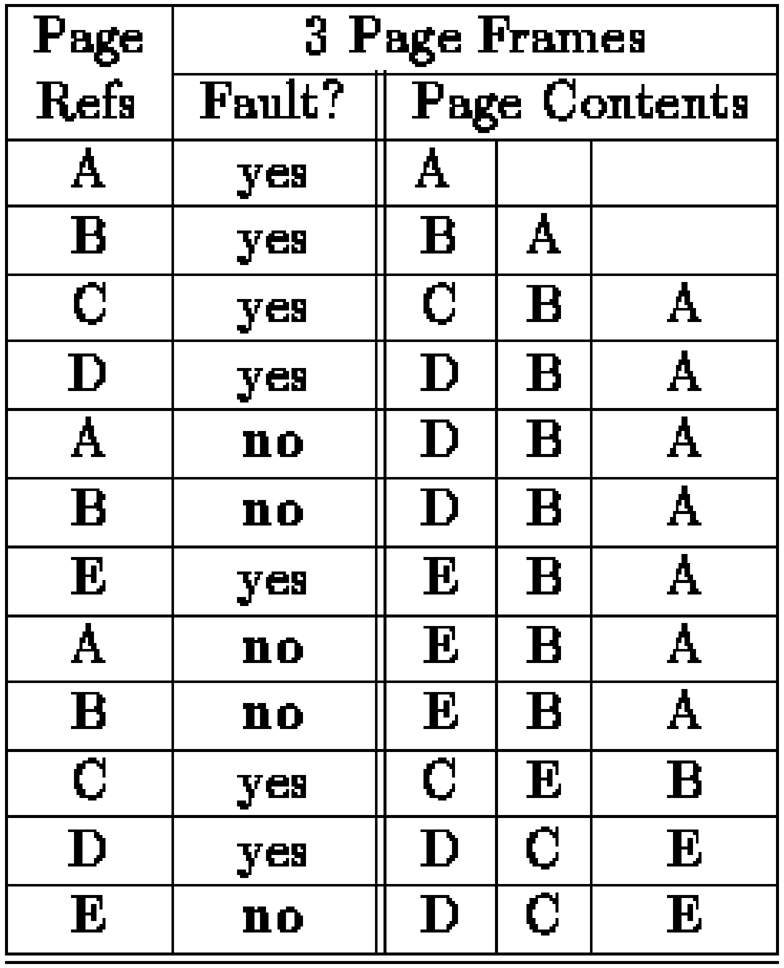
- Replace the page that will not be used again the farthest time in the future
  - Not realizable, but…
  - Provides a basis for comparison with other schemes (similar idea to SJF scheduling algorithm)
- Is difficult to implement because of trying to predict the reference string for a program
- Compiler technology can help by providing hints

## Principle of Optimality

- If the reference string could be predicted accurately, then we shouldn't use demand paging but we should use pre-paging (anticipatory paging) instead.
  - This would allow paging activity of pages needed in the future to be overlapped with computation

# Optimal Example

12 references,
7 faults total
(optimal)



| Page Refs | 3 Page Frames | | | |
|---|---|---|---|---|
| | Fault? | Page Contents | | |
| A | yes | A | | |
| B | yes | B | A | |
| C | yes | C | B | A |
| D | yes | D | B | A |
| A | no | D | B | A |
| B | no | D | B | A |
| E | yes | E | B | A |
| A | no | E | B | A |
| B | no | E | B | A |
| C | yes | C | E | B |
| D | yes | D | C | E |
| E | no | D | C | E |

3 page frames

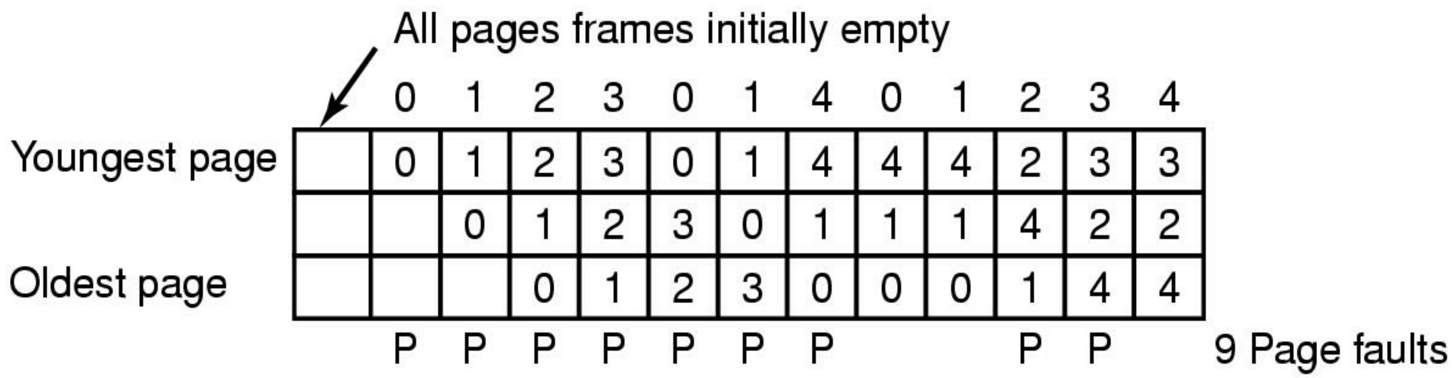
Reference string
column 1

$\leftarrow$ Which page
to replace?
Look into
future.
C not
referenced
for
a long time.

## FIFO Page Replacement Algorithm

- Non-usage-based algorithm.
- Replace the page that has been in primary memory the longest
- Maintain a FIFO queue of all pages
- Page at the head of queue replaced
- <u>Disadvantage</u>
  - page in memory the longest (head of the queue) may be being used heavily.

# FIFO example



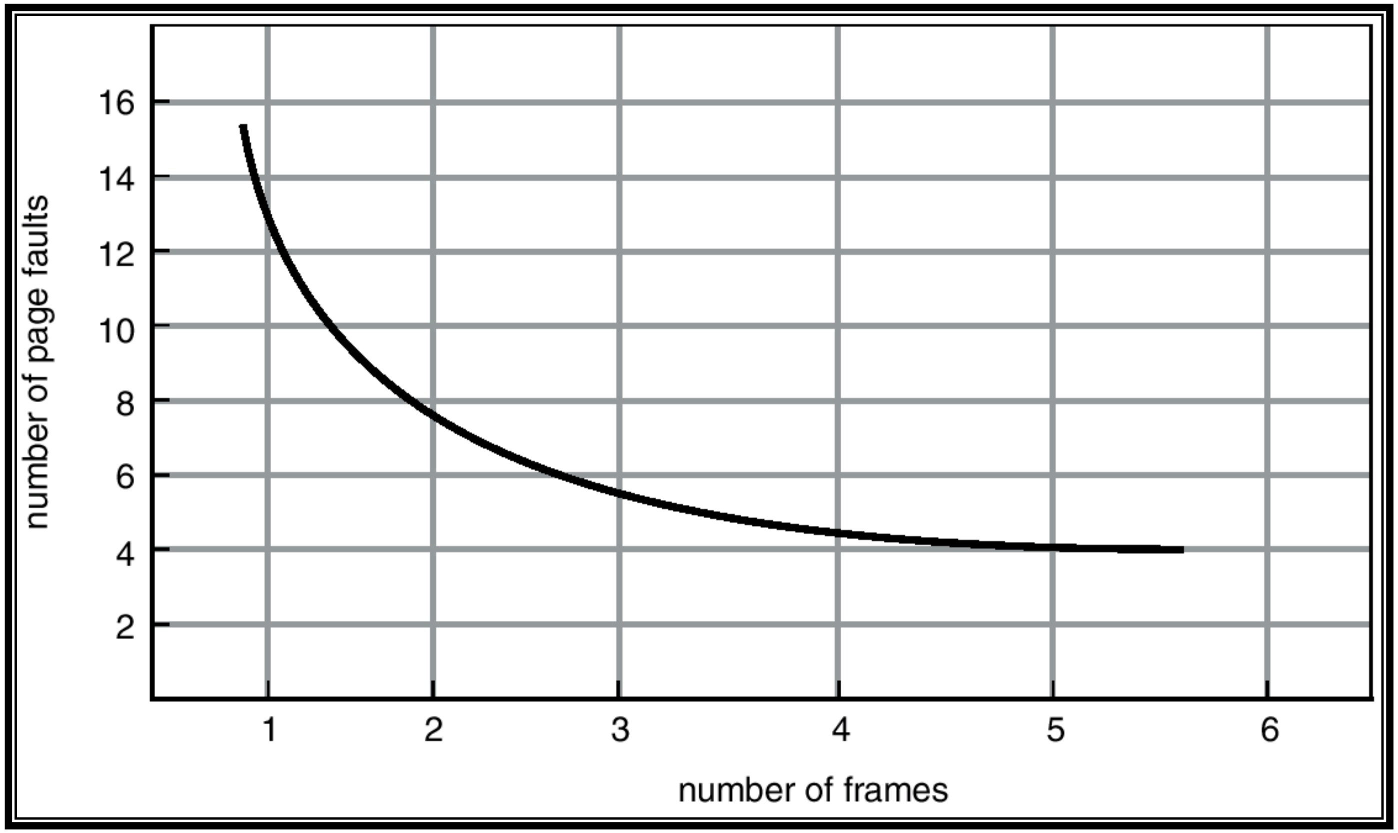
All pages frames initially empty

| | | 0 | 1 | 2 | 3 | 0 | 1 | 4 | 0 | 1 | 2 | 3 | 4 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Youngest page | | 0 | 1 | 2 | 3 | 0 | 1 | 4 | 4 | 4 | 2 | 3 | 3 | |
| | | | 0 | 1 | 2 | 3 | 0 | 1 | 1 | 1 | 4 | 2 | 2 | |
| Oldest page | | | | 0 | 1 | 2 | 3 | 0 | 0 | 0 | 1 | 4 | 4 | |
| | | P | P | P | P | P | P | P | | | P | P | | 9 Page faults |

- 12 references, 9 faults
- often used page (e.g., library) might generate too many page faults.

## Modeling Page Replacement Algorithms: Belady's Anomaly

Graph of Page Faults Versus The Number of Frames



- Normally, paging schemes behave this way.
- But, some algorithms don’t.

# Belady's Anomaly (for FIFO)

All pages frames initially empty

| | | 0 | 1 | 2 | 3 | 0 | 1 | 4 | 0 | 1 | 2 | 3 | 4 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Youngest page | | 0 | 1 | 2 | 3 | 0 | 1 | 4 | 4 | 4 | 2 | 3 | 3 | |
| | | | 0 | 1 | 2 | 3 | 0 | 1 | 1 | 1 | 4 | 2 | 2 | |
| Oldest page | | | | 0 | 1 | 2 | 3 | 0 | 0 | 0 | 1 | 4 | 4 | |
| | | P | P | P | P | P | P | P | | | P | P | | 9 Page faults |

(a)

| | | 0 | 1 | 2 | 3 | 0 | 1 | 4 | 0 | 1 | 2 | 3 | 4 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Youngest page | | 0 | 1 | 2 | 3 | 3 | 3 | 4 | 0 | 1 | 2 | 3 | 4 | |
| | | | 0 | 1 | 2 | 2 | 2 | 3 | 4 | 0 | 1 | 2 | 3 | |
| Oldest page | | | | 0 | 1 | 1 | 1 | 2 | 3 | 4 | 0 | 1 | 2 | |
| | | | | | 0 | 0 | 0 | 1 | 2 | 3 | 4 | 0 | 1 | |
| | | P | P | P | P | | | P | P | P | P | P | P | 10 Page faults |

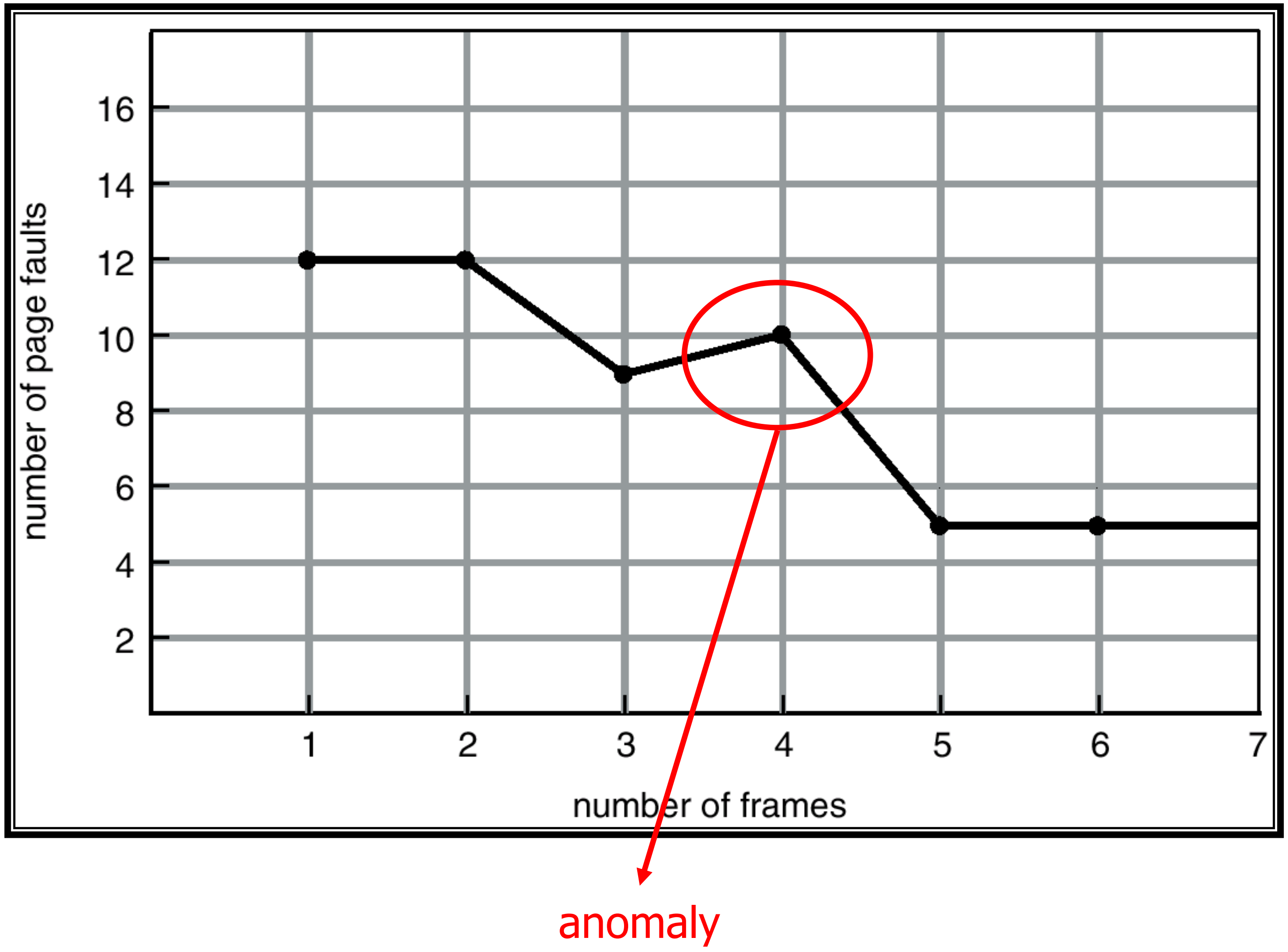
(b)

- FIFO with 3 page frames (9 page faults)
- FIFO with 4 page frames (10 page faults)
- P's show which page references show page faults

## Belady’s anomaly

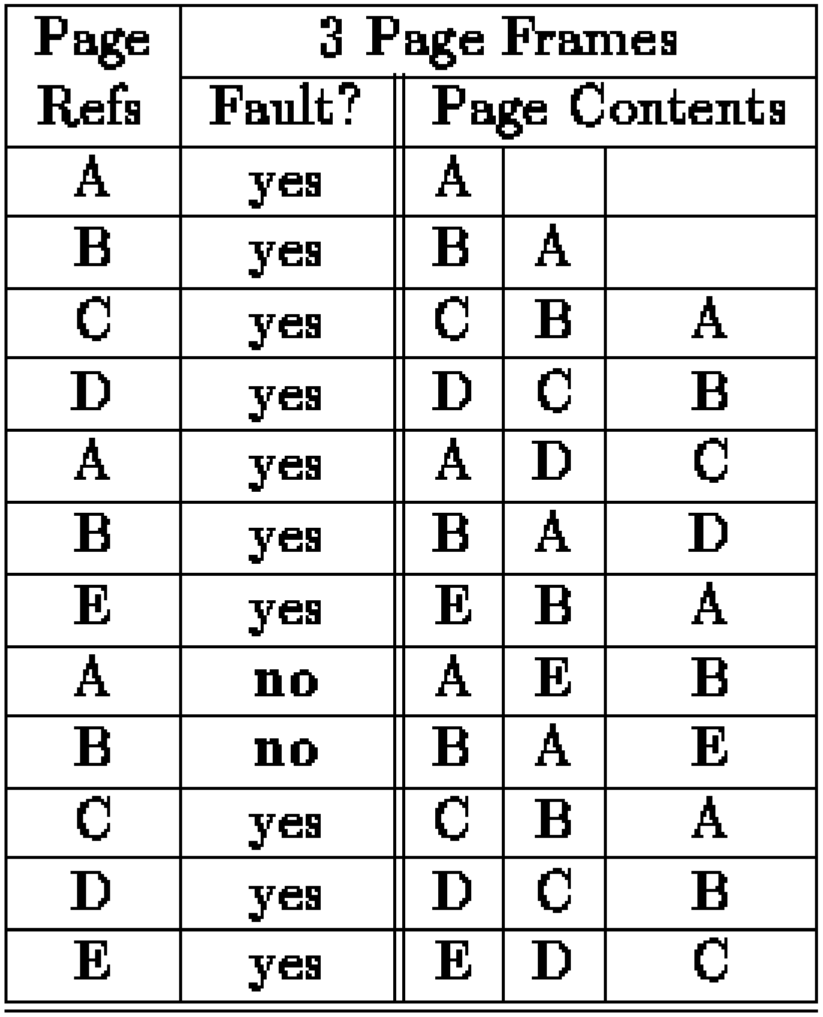
- Buying and adding more memory (i.e., adding new page frames) should not decrease performance!
- When reference strings contain strings as long as the no. of page frames, the this anomaly is seen for FIFO replacement scheme.

## FIFO Illustrating Belady's Anomaly



# LRU Page Replacement

12 references,
10 faults



| Page Refs | 3 Page Frames | | | |
|---|---|---|---|---|
| | Fault? | Page Contents | | |
| A | yes | A | | |
| B | yes | B | A | |
| C | yes | C | B | A |
| D | yes | D | C | B |
| A | yes | A | D | C |
| B | yes | B | A | D |
| E | yes | E | B | A |
| A | no | A | E | B |
| B | no | B | A | E |
| C | yes | C | B | A |
| D | yes | D | C | B |
| E | yes | E | D | C |

# LRU and Anomalies

Anomalies do not occur.

12 references, 8 faults

| Page | 4 Page Frames | | | | |
|---|---|---|---|---|---|
| Refs | Fault? | Page Contents | | | |
| A | yes | A | | | |
| B | yes | B | A | | |
| C | yes | C | B | A | |
| D | yes | D | C | B | A |
| A | no | A | D | C | B |
| B | no | B | A | D | C |
| E | yes | E | B | A | D |
| A | no | A | E | B | D |
| B | no | B | A | E | D |
| C | yes | C | B | A | E |
| D | yes | D | C | B | A |
| E | yes | E | D | C | B |

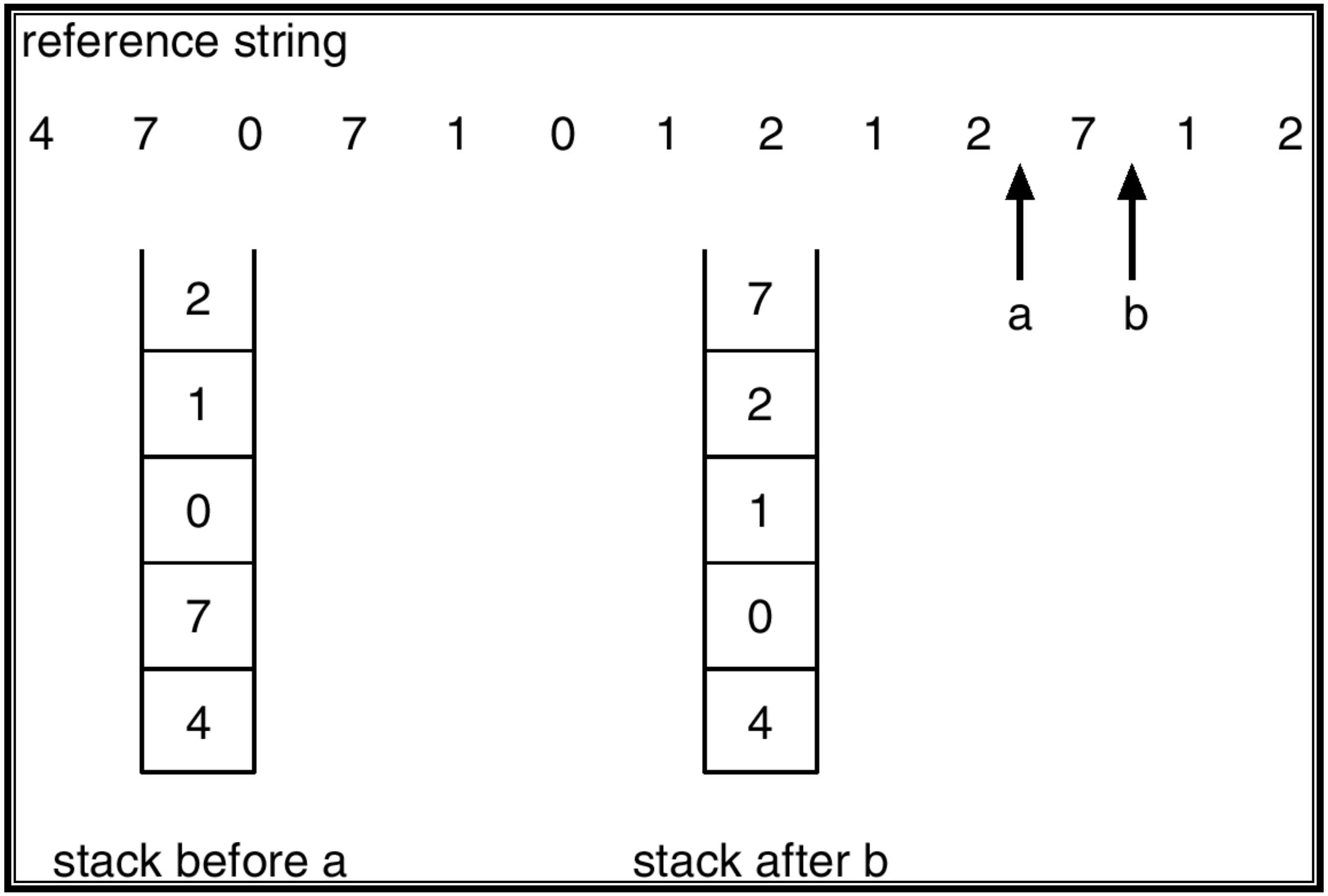
## LRU Issues

- Does not suffer from Belady's anomaly
- Not perfect: no knowledge that A’s and B’s are used a lot.
- Implementation strategies:
  - Use time
    - Record time of reference with page table entry
    - or use counter as clock
    - Search for smallest time
  - Use stack
    - Remove reference of page from stack (implemented as linked list)
    - Push it on top of stack
- Both approaches require large processing overhead, more space, and hardware support

## LRU Algorithm (Cont.)

- 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

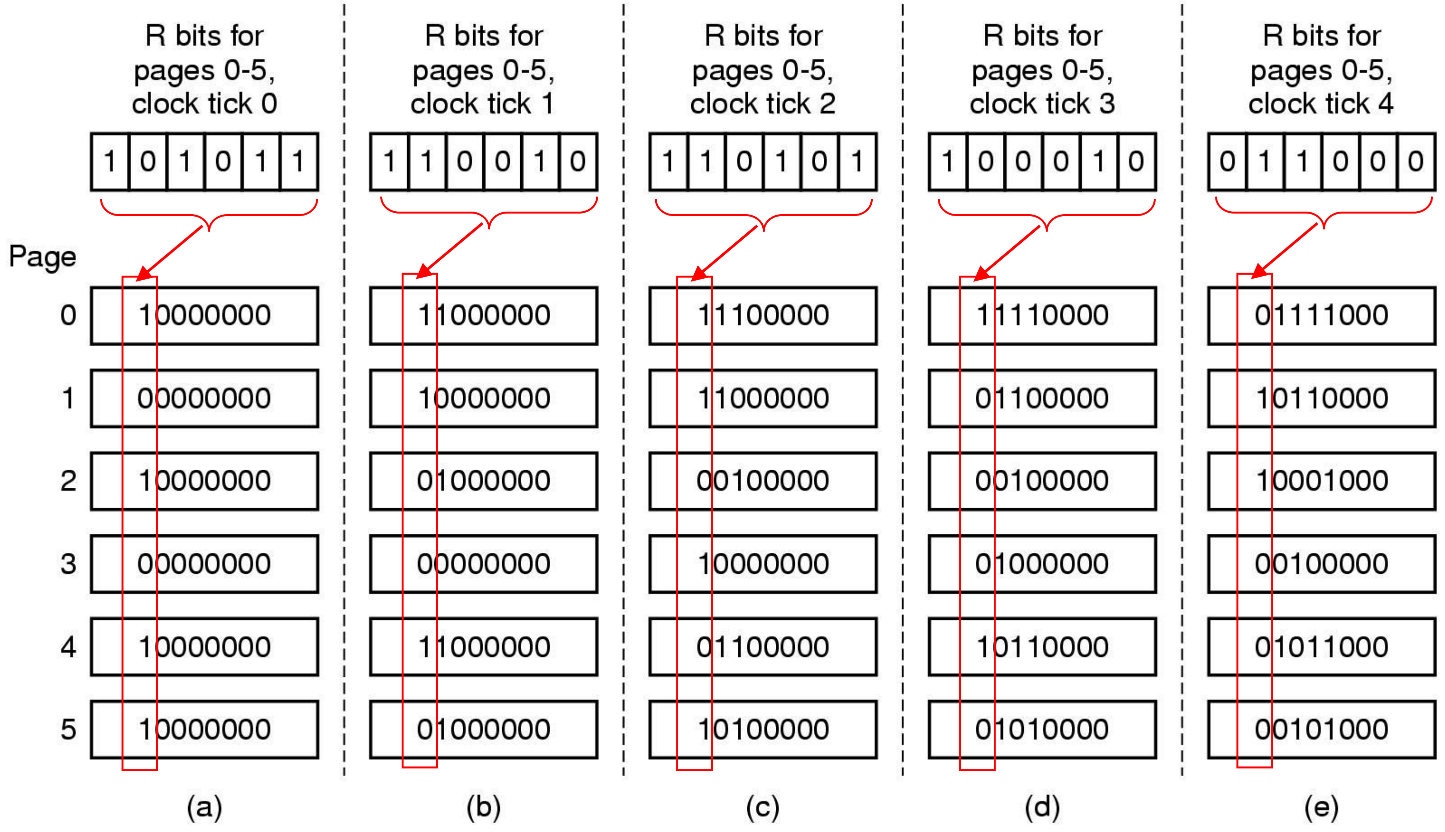
## Use Of A Stack to Record The Most Recent Page References



## An LRU Approximation (aging) also called Not Frequently Used (NFU)

- Additional reference bits
- A number of reference bits are kept per page (say, 8 bits)
- If the page is referenced, the reference bit is set
- At regular intervals (e.g., 100msec), a timer interrupt is generated, and the OS shifts the reference bits to the right, putting the current reference bit into MSB.
  - At the same time it clears the R bit for the next time slice.
- The page with register holding the lowest number is the least recently used
- e.g., 00110011 would be accessed more recently than 00010111
- The value may not be unique: use FCFS to  resolve conflicts

# LRU approximation in action



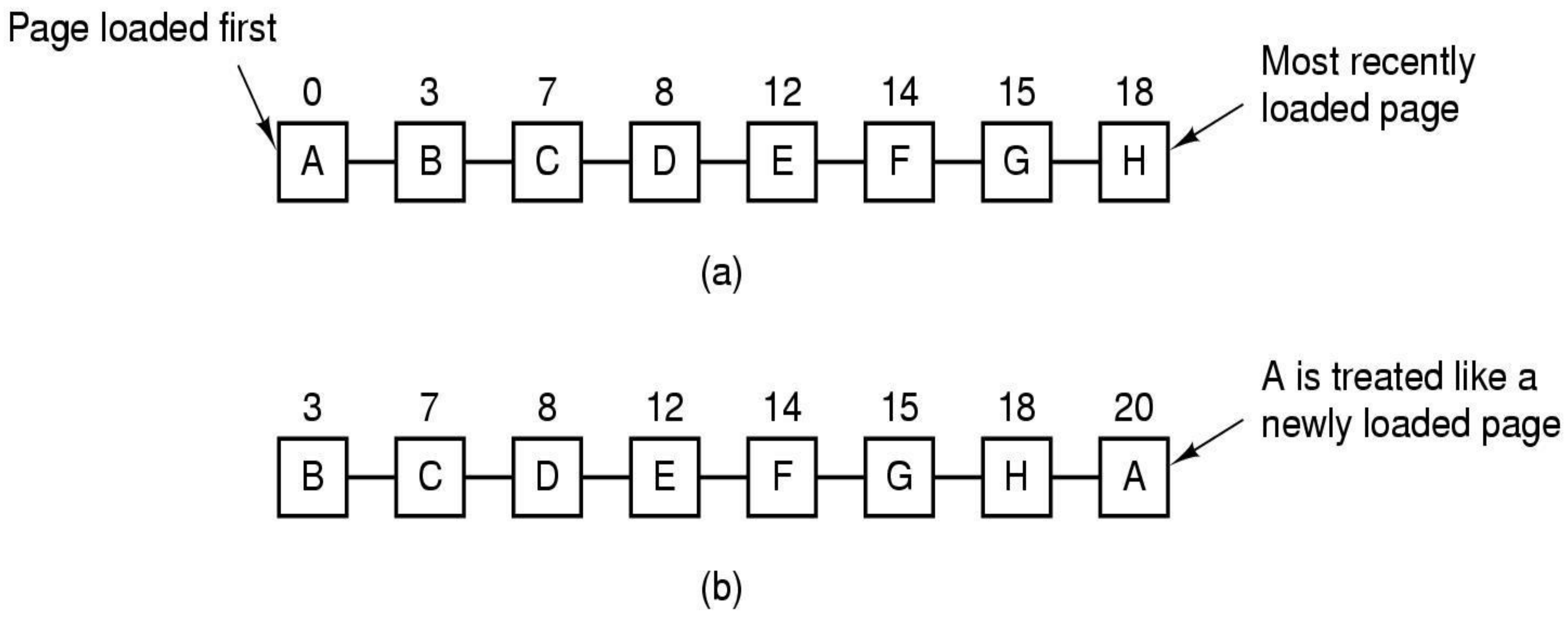
R bits for pages 0-5, clock tick 0
1 0 1 0 1 1
R bits for pages 0-5, clock tick 1
1 1 0 0 1 0
R bits for pages 0-5, clock tick 2
1 1 0 1 0 1
R bits for pages 0-5, clock tick 3
1 0 0 0 1 0
R bits for pages 0-5, clock tick 4
0 1 1 0 0 0
Page
0
1
2
3
4
5
(a) 10000000 00000000 10000000 00000000 10000000 10000000
(b) 11000000 10000000 01000000 00000000 11000000 01000000
(c) 11100000 11000000 00100000 10000000 01100000 10100000
(d) 11110000 01100000 00100000 01000000 10110000 01010000
(e) 01111000 10110000 10001000 00100000 01011000 00101000

- The aging algorithm simulates LRU in software
- Note 6 pages for 5 clock ticks, (a) – (e)

# LRU Approximation Algorithms

- Second chance
  - Need reference bit.
  - Pages are kept in FIFO order using a circular list
  - If next page to be replaced (in FIFO order) has reference bit = 1.  then:
    - set reference bit 0.
    - leave page in memory. ➔ this is where the page is given a second chance.
    - Move the page to the tail of the list
    - Continue looking for a page to replace, subject to same rules.
- System V, R4 uses a variant of second chance
  - first time a page is visited in queue it’s kept if bit=1
  - 2nd time around it gets paged out
  - That is, a page at the head of FIFO gets a second chance.

## Second Chance Page Replacement Algorithm

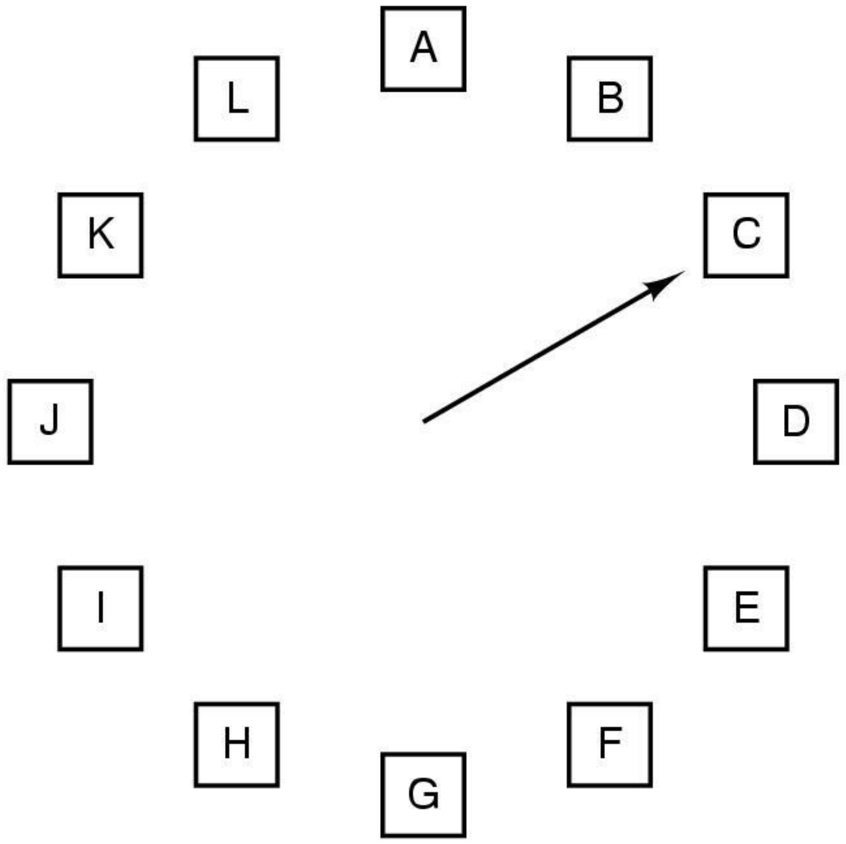


- Operation of a second chance
  - pages sorted in FIFO order
  - Page list if fault occurs at time 20, A has R bit set (numbers above pages are loading times)

## Second chance

- Second chance tends to be costly because the pages are moved around the FIFO queue all the time.
- Instead use a modification called the Clock algorithm.

# The Clock Page Replacement Algorithm

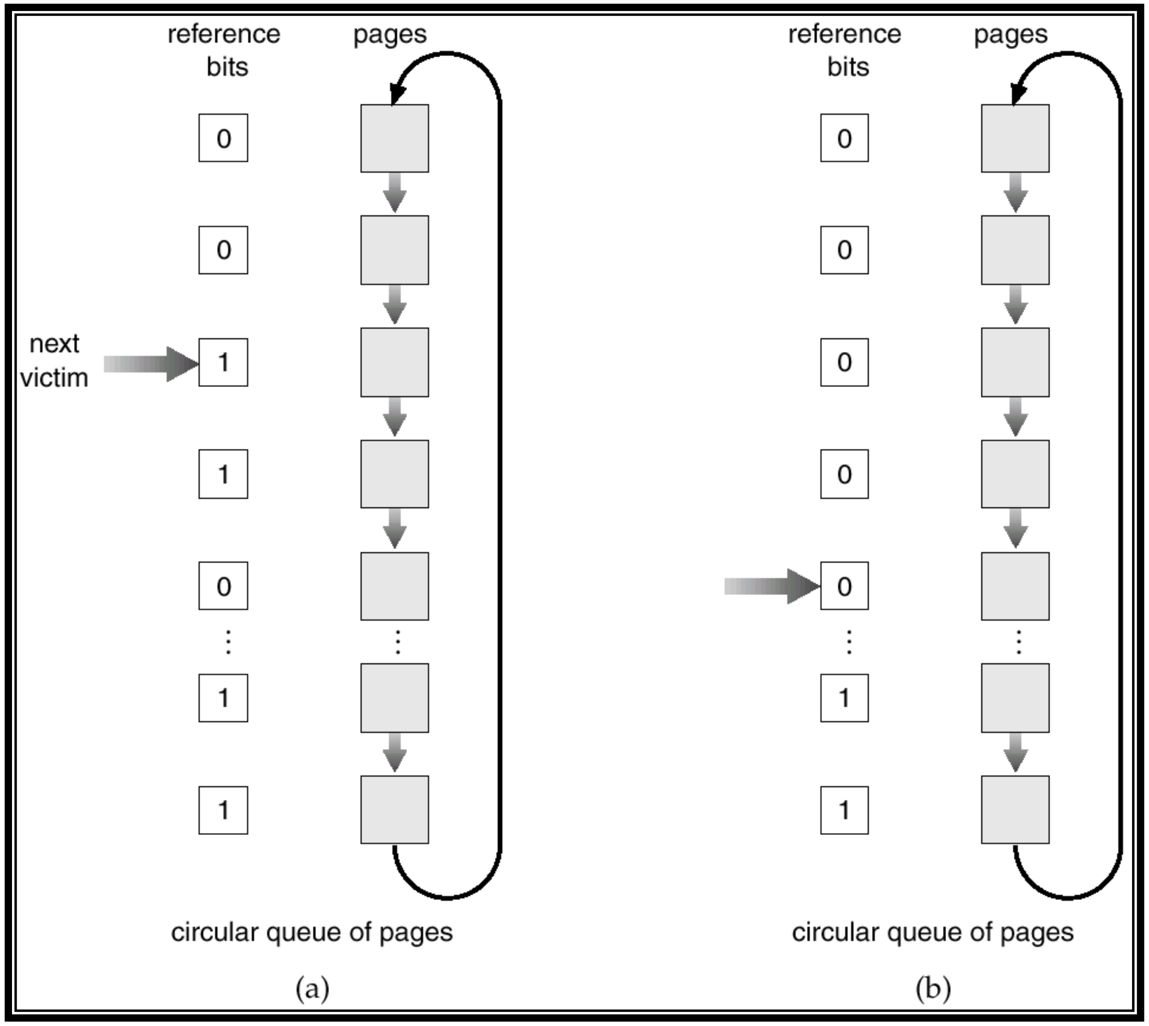


When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:

R = 0: Evict the page
R = 1: Clear R and advance hand

Advancing the hand is equivalent to putting the page C in this example at the tail of the queue.

# Second-Chance (Clock) Page-Replacement Algorithm with circular queues



## Page Classes (Enhanced Second Chance)

- Take into account in replacement if page is dirty (dirty means a lot of work writing page onto disk)
- Each page has Reference bit, Modified bit
  - bits are set when page is referenced, modified
  - Periodically (e.g., on timer interrupts), the R bit is cleared
- At the time of the page fault, R = 0 means the page has not been referenced in the last time slice and R = 1 means the page has been referenced in the last slice.
- At page fault time, then, pages are classified into classes as follows:
  1. (0,0): not referenced, not modified
  2. (0,1): not referenced, modified
  3. (1,0): referenced, not modified
  4. (1,1): referenced, modified
- NRU removes page at random
  - from lowest numbered non empty class

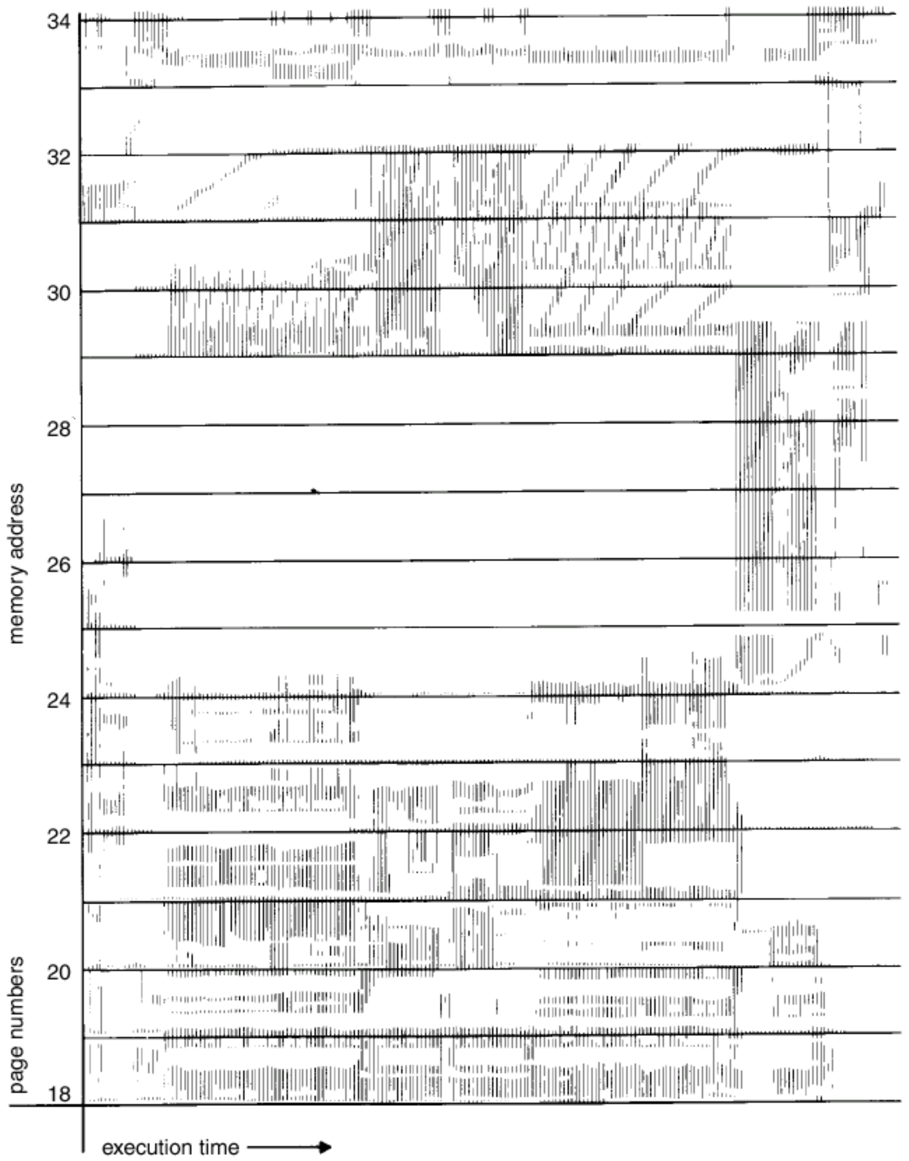
## Counting Algorithms

- Keep a counter of the number of references that have been made to each page.
- LFU Algorithm: (Least frequently used) replaces page with smallest count.
- MFU Algorithm: (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.

## Working Set

- The working set model assumes locality
- **The principle of locality states that a program clusters its access to data and text temporally**
  - Many empirical studies done → all support this assumption
- Thus, as the number of <u>page frames</u> increases above some threshold, the page fault rate will drop dramatically

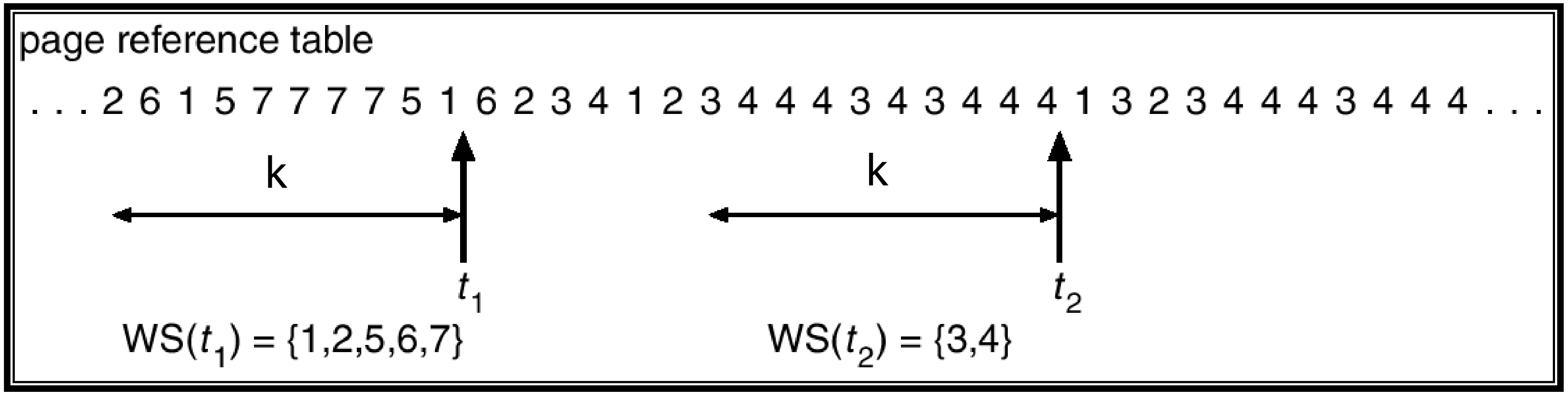
# Locality In A Memory-Reference Pattern



## Working Set

- Developed in 1960’s and 1970’s (Denning)
- Working set parameter–window size: *k*.
- The set of pages in the most recent *k* pages is the working set of the process.
  - If a page is actively used, it will be in the working set.
  - If it’s no longer used, it will be dropped from the working set, *k* time units after its last reference.

# Working-set model

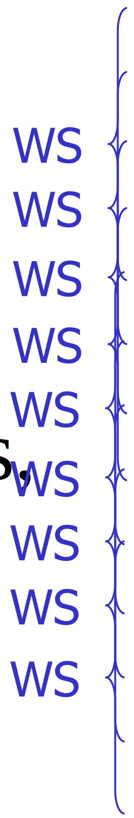


# Working Set Example

Window size is k

12 references, 8 faults

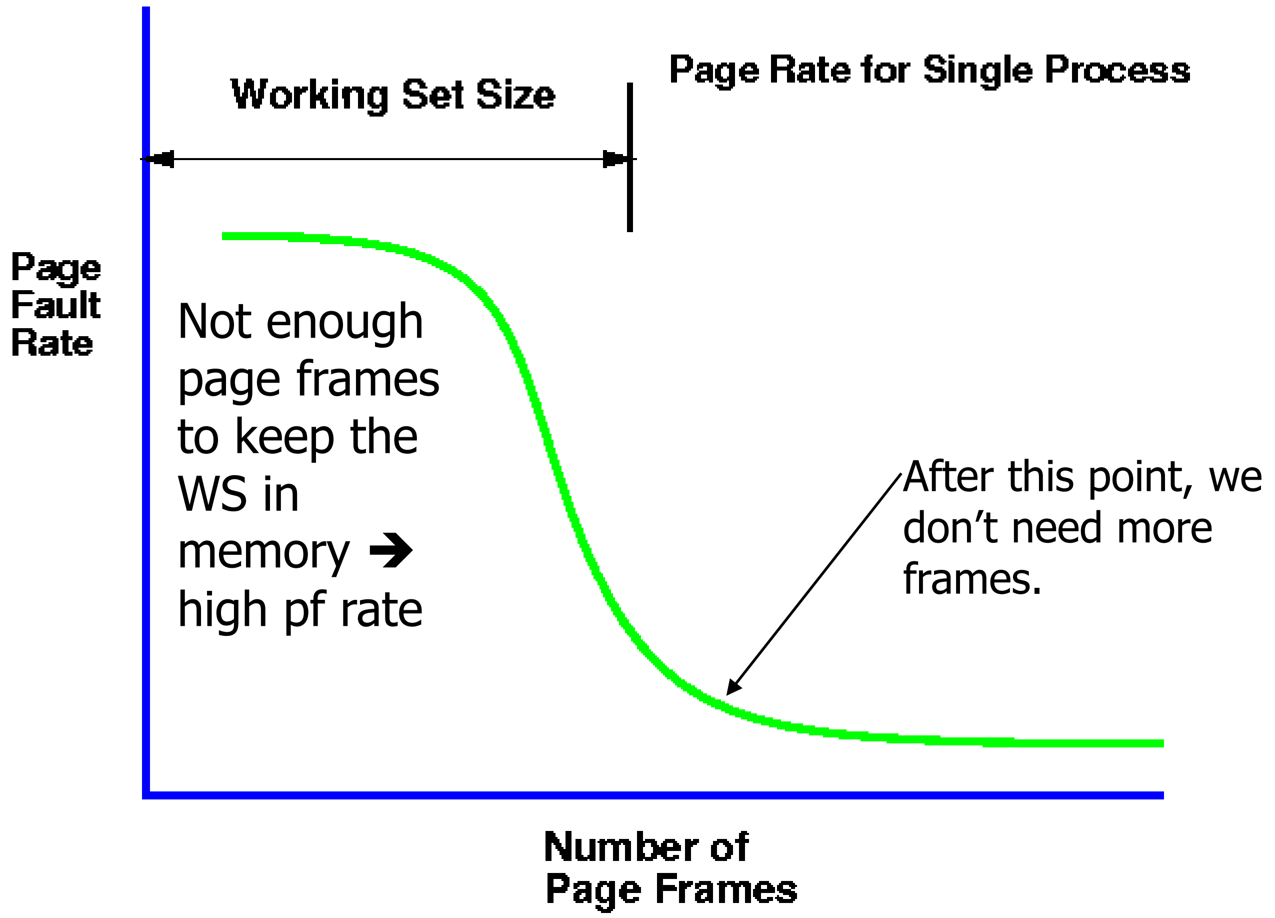
| Page Refs | k= 4 references | | | | |
|---|---|---|---|---|---|
| | Fault? | Page Contents | | | |
| A | yes | A | | | |
| B | yes | A | B | | |
| C | yes | A | B | C | |
| D | yes | A | B | C | D |
| A | no | A | B | C | D |
| B | no | A | B | C | D |
| E | yes | A | B | D | E |
| A | no | A | B | E | |
| B | no | A | B | E | |
| C | yes | A | B | C | E |
| D | yes | A | B | C | D |
| E | yes | B | C | D | E |



Replace C

Working set drops below window size.

# Working Set and # of page frames



## Working Set Issue

- We saw the principle of the "working set." Now the question is how to implement it for page replacement.
- How do we keep track of working set?
  - Actual WS algorithm is too expensive to implement.
  - Instead, we implement approximations.

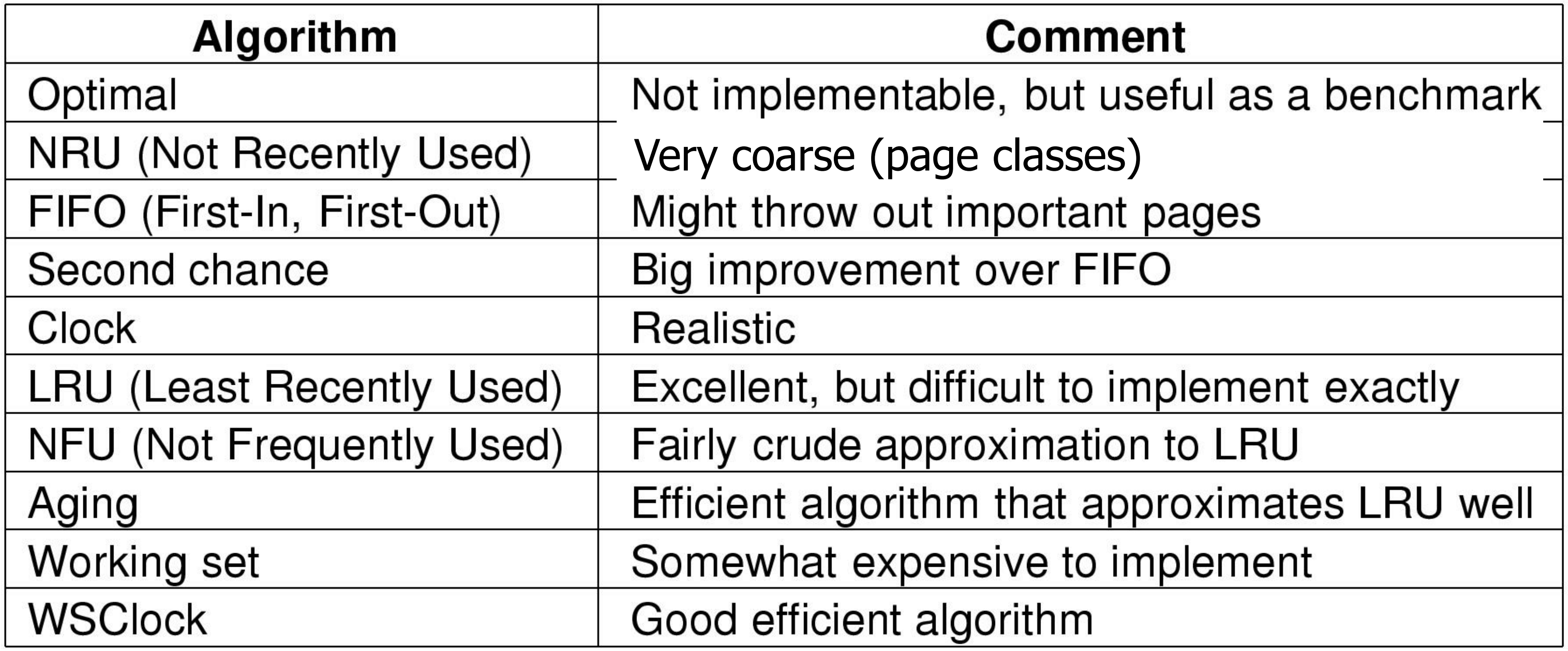
## Practical implementation of the WS page replacement

- Approximate working set model using timer and reference bit [a discrete evaluation]
- Last *k* page references (e.g., 10 million) → pages references in the last $\tau$ time units (e.g., 100 msec).
- Set timer to interrupt every $\tau$ time units.
- When timer interrupt goes off, find pages in the working set (using R bit in PT); [no info about the order of references…]
- Remove pages that have not been referenced from the WS and reset reference bit for the next time slice.
- How to do the last two bullets?
  - We don't keep explicit WS data. Instead, we decide what to do at page fault time.

## Another WS algorithm

- WSClock: combination of Clock algorithm and WS algorithm
- We avoid scanning the entire page table each time.
- Circular list of page frames (as in Clock). Each entry has time of last use and R and M bits.
- Current page is tracked with a pointer (which is the clock hand).

# Review of Page Replacement Algorithms



| Algorithm | Comment |
|---|---|
| Optimal | Not implementable, but useful as a benchmark |
| NRU (Not Recently Used) | Very coarse (page classes) |
| FIFO (First-In, First-Out) | Might throw out important pages |
| Second chance | Big improvement over FIFO |
| Clock | Realistic |
| LRU (Least Recently Used) | Excellent, but difficult to implement exactly |
| NFU (Not Frequently Used) | Fairly crude approximation to LRU |
| Aging | Efficient algorithm that approximates LRU well |
| Working set | Somewhat expensive to implement |
| WSClock | Good efficient algorithm |

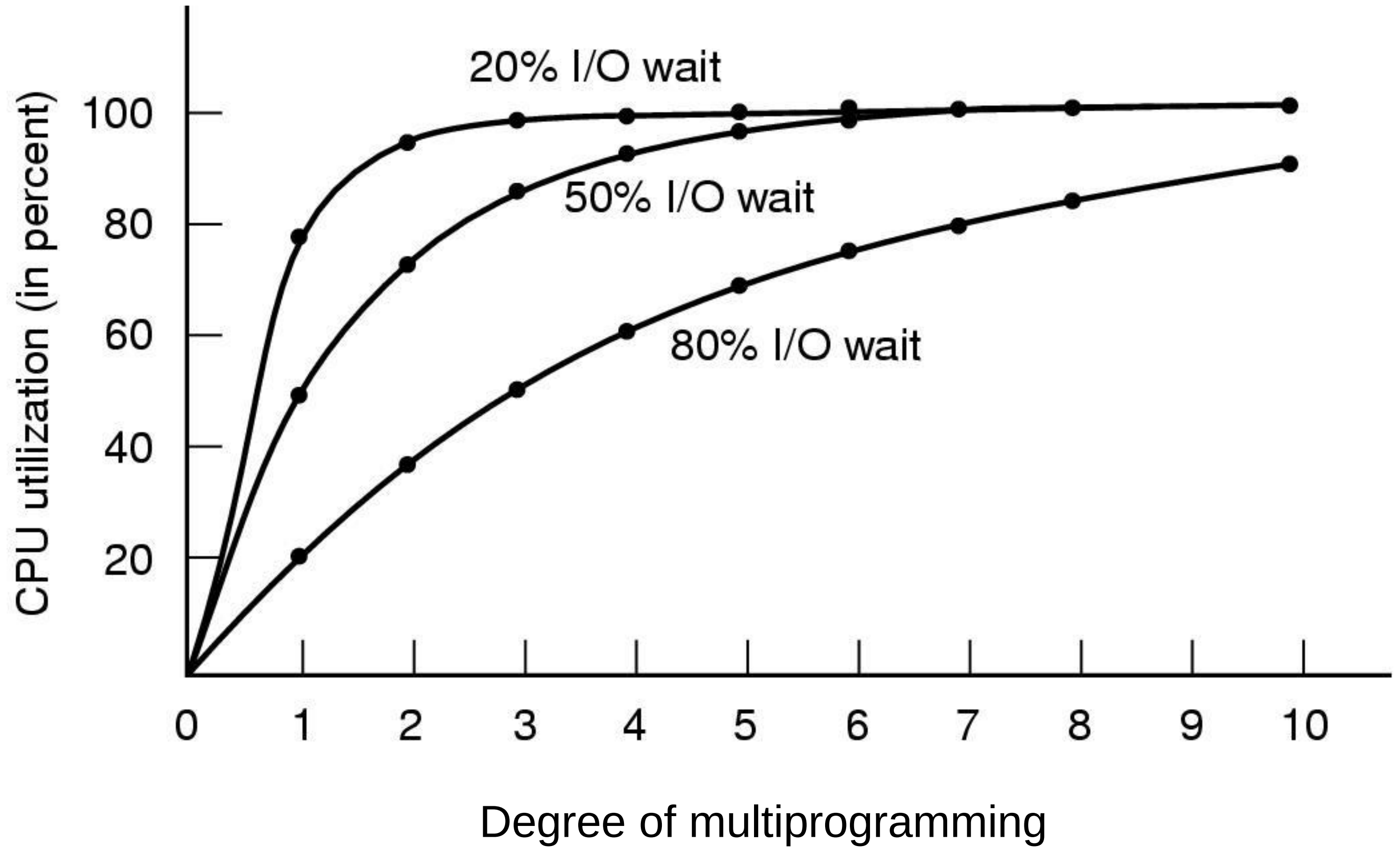
## Other issues (read textbook)

- Policy issues
  - Equal vs. proportional allocation
  - Local vs. global page replacement

## Thrashing

- Thrashing ≡ a process is busy swapping pages in and out (disk activity) rather than doing useful computation (CPU activity).
- It has been observed that as page frames per VM space decrease, the page fault rate increases
- If a process does not have "enough" page frames, 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.

# Multiprogramming model we saw

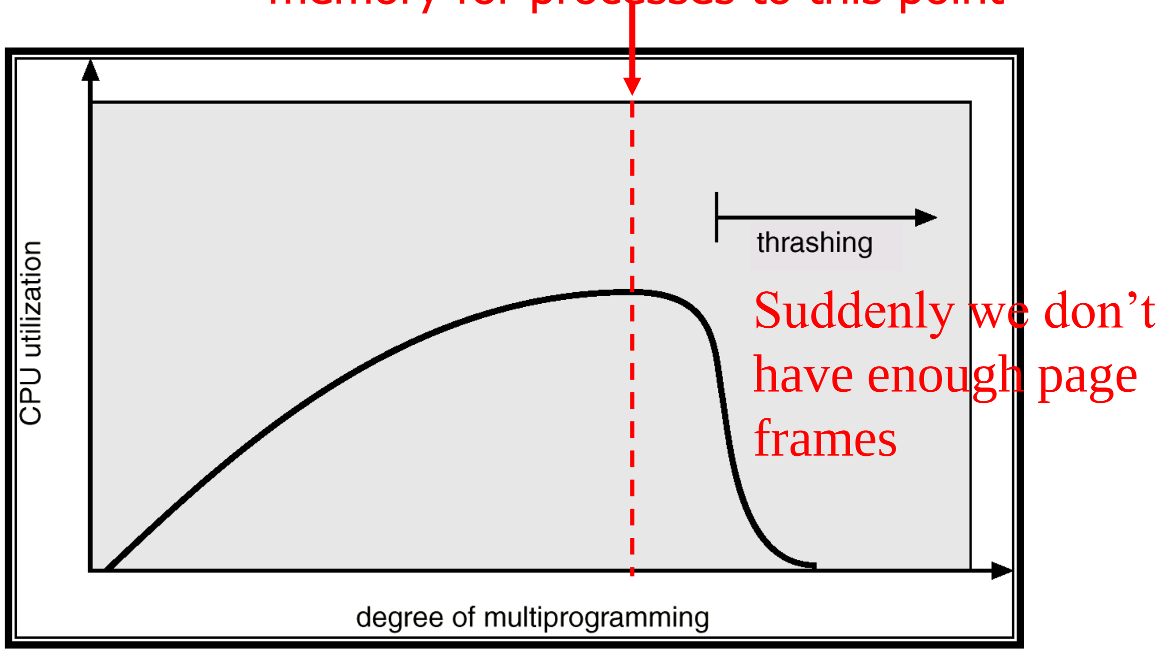


- CPU utilization as a function of number of processes in memory

# Thrashing

The model in previous slide captures reality only if there is enough memory for processes to this point

Manage this thrashing by suspending processes in scheduling



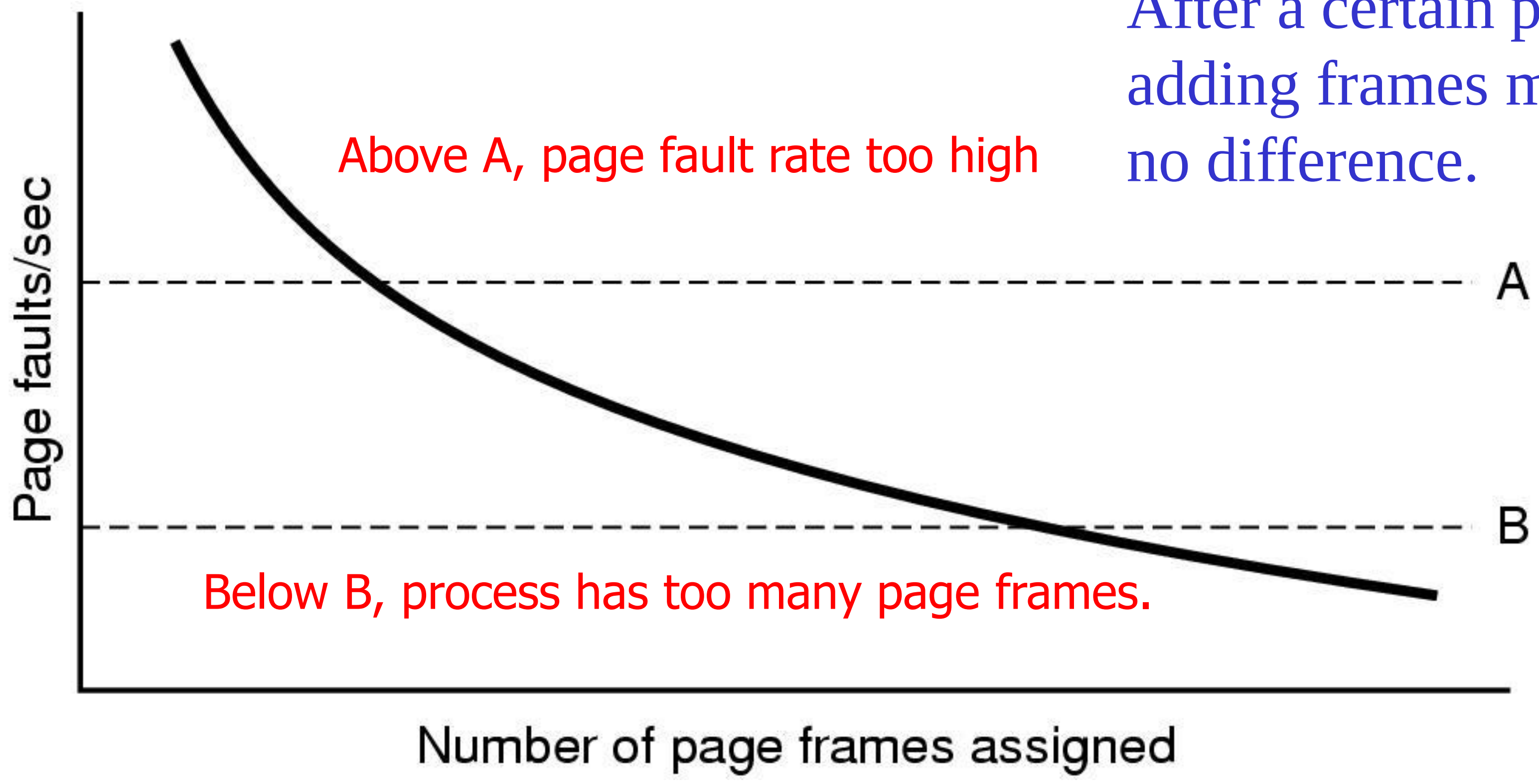
- Why does paging work?
  Locality model
  - Process migrates from one locality to another.
  - Localities may overlap.
- Why does thrashing occur?
  Σ size of locality > total memory size (# of frames)

# Thrashing and CPU Utilization

- As the page rate goes up, processes get suspended on pageout queues for the disk
- The system may try to optimize performance by starting NEW jobs
- Starting new jobs will reduce the number of page frames available to each process, increasing the page fault requests
- System throughput plunges
- Before starting new jobs, ensure there aren’t too many jobs already suspended due to paging.

## Typical page fault rate behavior

- Page fault rate as a function of the number of page frames assigned



After a certain point adding frames makes no difference.

## Ad Hoc Techniques to improve performance

- Separate page out from page in (buffer a page being faulted out)
- Keep a pool of free frames (in memory for each process)
- When a page is to be replaced, use a free frame
- Read the faulting page and restart the faulting process while page out is occurring
- useful in recovering if we made a mistake paging out (as in the FIFO example we saw).

# Ad Hoc Techniques

- Write dirty pages to disk whenever the paging device is free and reset the dirty bit. This allows page replacement algorithms to replace clean pages.
- Cache paged out pages in primary memory
  - Page out dirty pages as before
  - Return pages to a free pool but remember which page frame they are
  - If system needs to bring same page in again, reuse page (avoids disk activity at page fault time)
  - If system needs to page in data, choose any page in free pool
  - System V, R4 implements this strategy
    - tries to clean pages
    - it has a page-out queue

## Page Size Considerations

- Small pages require large page tables
- Large pages imply significant amounts of page may not be referenced
- Locality of reference tends to be small (256), implying small pages [empirical studies]
- I/O transfers have high seek time, implying larger pages. (More data per seek.)
  [This is the most important factor in determining page size in real systems--trading off memory utilization vs. speed of data transfer from disk].
- Internal fragmentation minimized with small page size

## Other Considerations

- **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 TLB misses.

## Other Considerations

- Program structure, assume 4KB page size.

  **int A[][] = new int[1024][1024];**

  Each row is stored in one page (row-major storage):
  1024 ints=4096 bytes.

  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

# Summary

- Paging policies
  - There are a lot of them.

Non-Usage based algorithms

- Random
  - sometimes may work surprisingly well.
- FIFO

Usage based algorithms

- Optimal (not realizable — only for comparison)
- LRU
  - Too much overhead; too expensive.
- NFU (aging) -- approach approximates LRU using a shift register

# Summary

- Second chance modification of FIFO to include usage infor--uses just the reference bit and a round robin selection
  - Variations include Clock algorithm.
- Page classes, find replacement page from lowest class to highest.
  - could be combined with second chance algorithm.
- Working set
  - Used most commonly in real systems!
- Ad Hoc methods and Pool algorithms
  - may make a real difference in real systems.

## Summary

<u>Policy issues</u>

- Fixed allocation vs. proportional allocation vs priority allocation
- Local vs. global allocation/replacement
- Page size issues and other considerations
- Thrashing
- Page fault frequency and load control (memory scheduling)