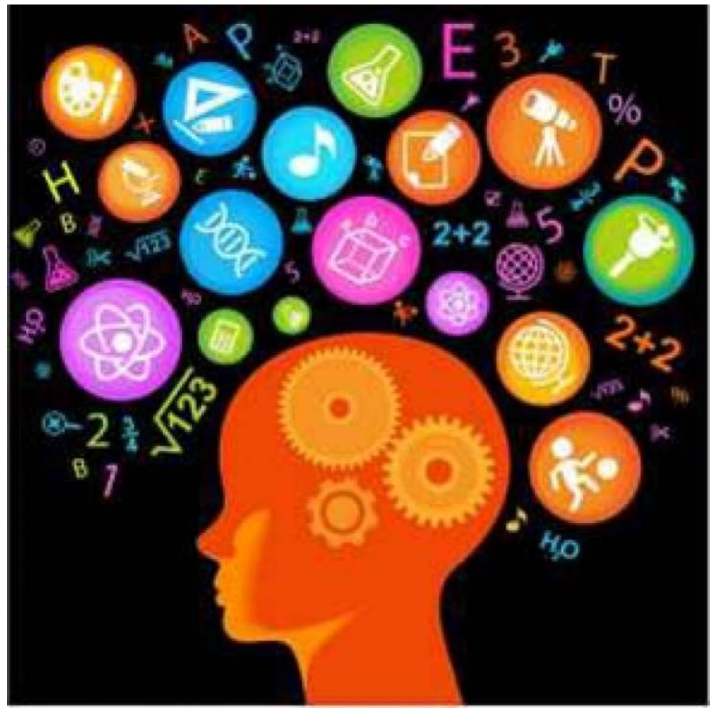
# Memory Management
## Part -3




Prachi Pandey

C-DAC Bangalore

prachip@cdac.in

# Topics

- Virtual Memory
  - Demand Paging
  - Page Faults
  - Page Replacement algorithms
    - FIFO
    - LRU
    - Optimal

# Memory Limitation

- Although code needs to be in memory to be executed, the entire program does not need to be
  - Only small sections execute in any small window of time, and
  - Error code, unusual routines, large data structures do not need to be in memory for the entire execution of the program
- What if we do not load the entire program into memory?
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running implies more programs run at the same time
  - Increased CPU utilization and throughput with no increase in response time or turnaround time
  - Less I/O needed to load or swap programs into memory -> each user program runs faster

# Virtual Memory – contd.

- Separation of user logical memory from physical memory
    - Only part of the program and its data needs to be in memory for execution
    - Logical address space can therefore be much larger than physical address space
    - Also allows address spaces to be shared by several processes
    - Allows for more efficient process creation
    - More programs running concurrently
    - Less I/O needed to load or swap processes

# Virtual Memory – contd.

- Virtual address space – logical view of how process is stored in memory
  - Usually starts at address 0, contiguous addresses until end of space
  - 48-bit virtual addresses implies 2^48 bytes of virtual memory
  - Physical memory is still organized into page frames
  - MMU must map virtual to physical
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

# Demand Paging

- **Demand paging** (as opposed to **anticipatory** paging) is a method of virtual memory management.
- In this method, the operating system copies a disk page into physical memory only if an attempt is made to access it and that page is not already in memory (*i.e.*, if a **page fault** occurs).
- To implement Demand paging we must develop
  - Frame allocation algorithm
  - Page replacement algorithm
- It follows that a process begins execution with none of its pages in physical memory, and many page faults will occur until most of a process's working set of pages is located in physical memory. This is an example of a **lazy loading** technique.

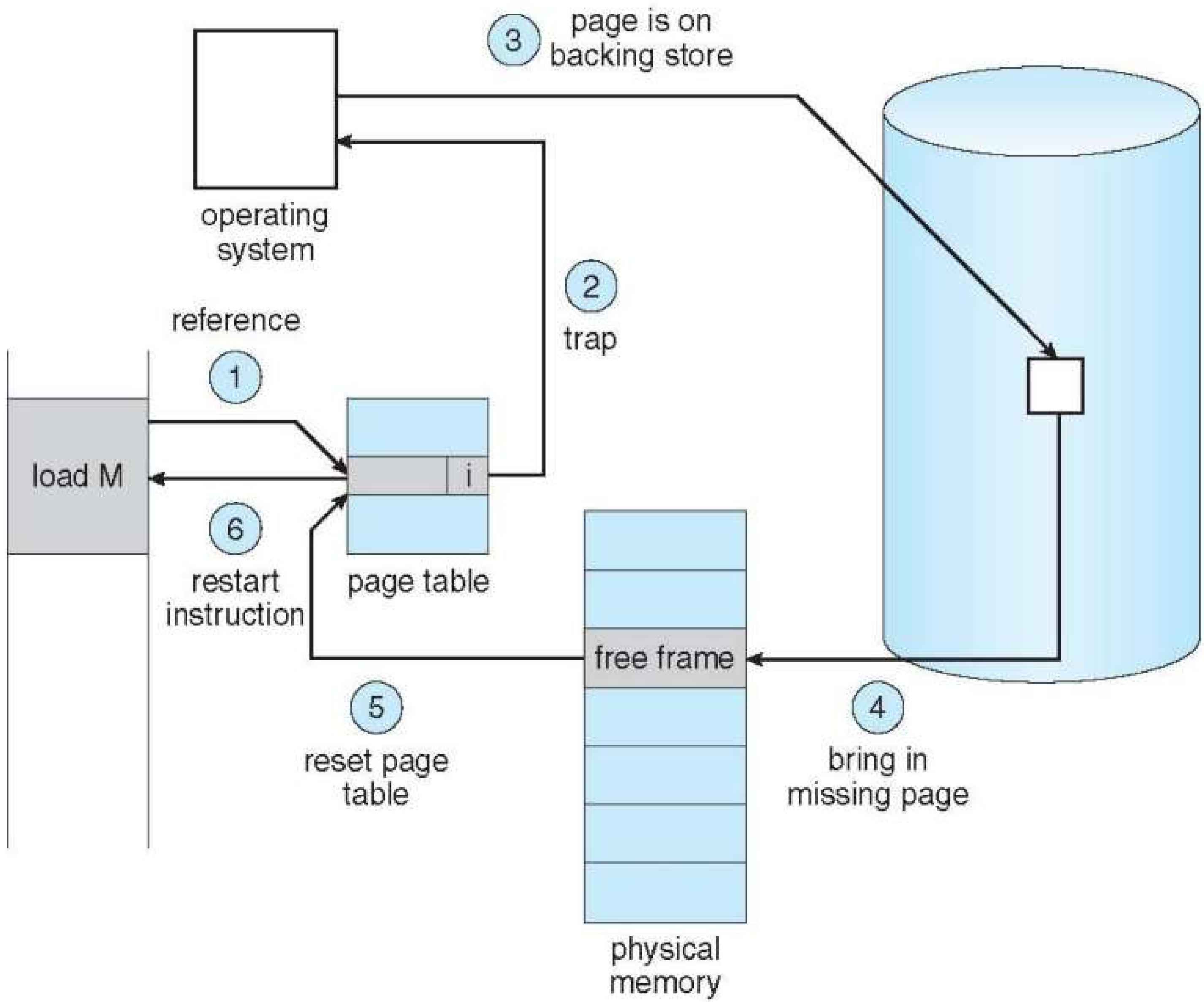
# Demand Paging

- In demand paging pages are brought into memory only when needed:
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
- With each page table entry a valid–invalid bit is associated (v -> in-memory – memory resident, i -> not-in-memory)
  - Initially valid–invalid bit is set to i on all entries
- During MMU address translation, if valid–invalid bit in page table entry is i -> page fault

# Handling page fault

- If there is a reference to a page, the first reference to that page will trap to operating system, i.,e. it is a
  - Page fault
- Operating system looks at another table to decide:
  - Invalid reference -> abort
  - Just not in memory
- Find free frame
- Swap page into frame via scheduled disk operation
- Reset tables to indicate page now in memory Set validation bit = v
- Restart the instruction that caused the page fault

# Steps in handling page fault

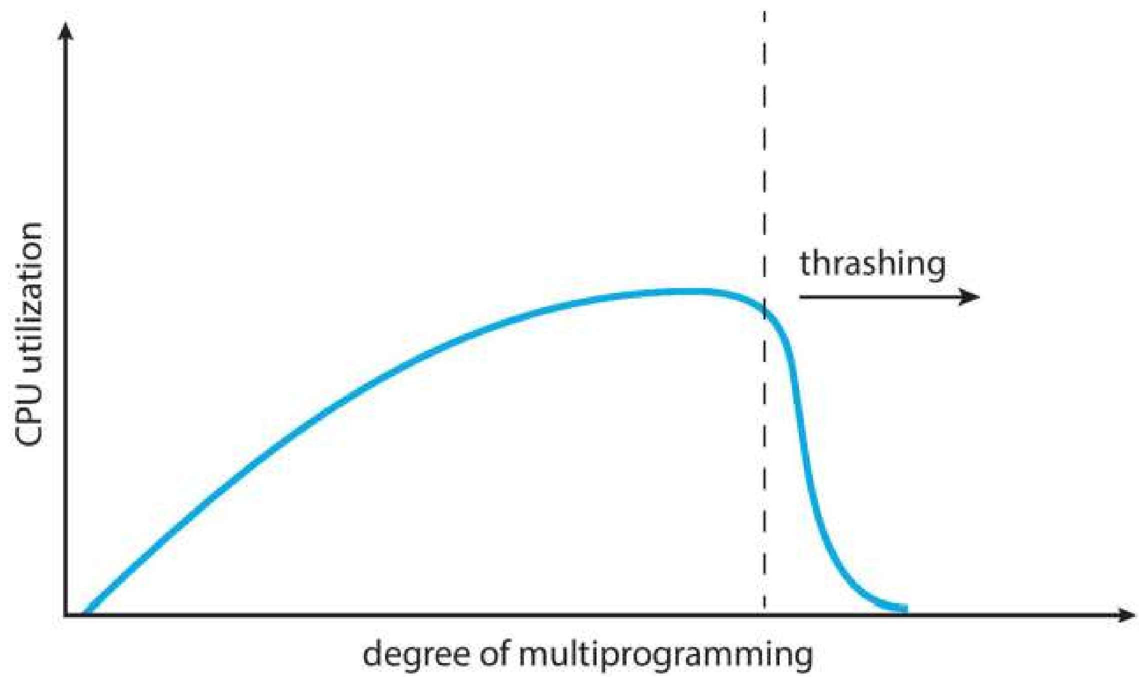


# Thrashing

- If a process does not have "enough" pages in the main memory, 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

- A process that is spending more time paging than executing is said to be ***thrashing.***



# Page Replacement

**Basic Scheme**

1. Find the location of the desired page on the 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 the victim page to the disk; change the page and frame tables accordingly
3. Read the desired page into the (newly) free frame; change the page and frame tables
4. Restart the user process

- Main objective of a good replacement algorithm is to achieve a low *page fault rate*

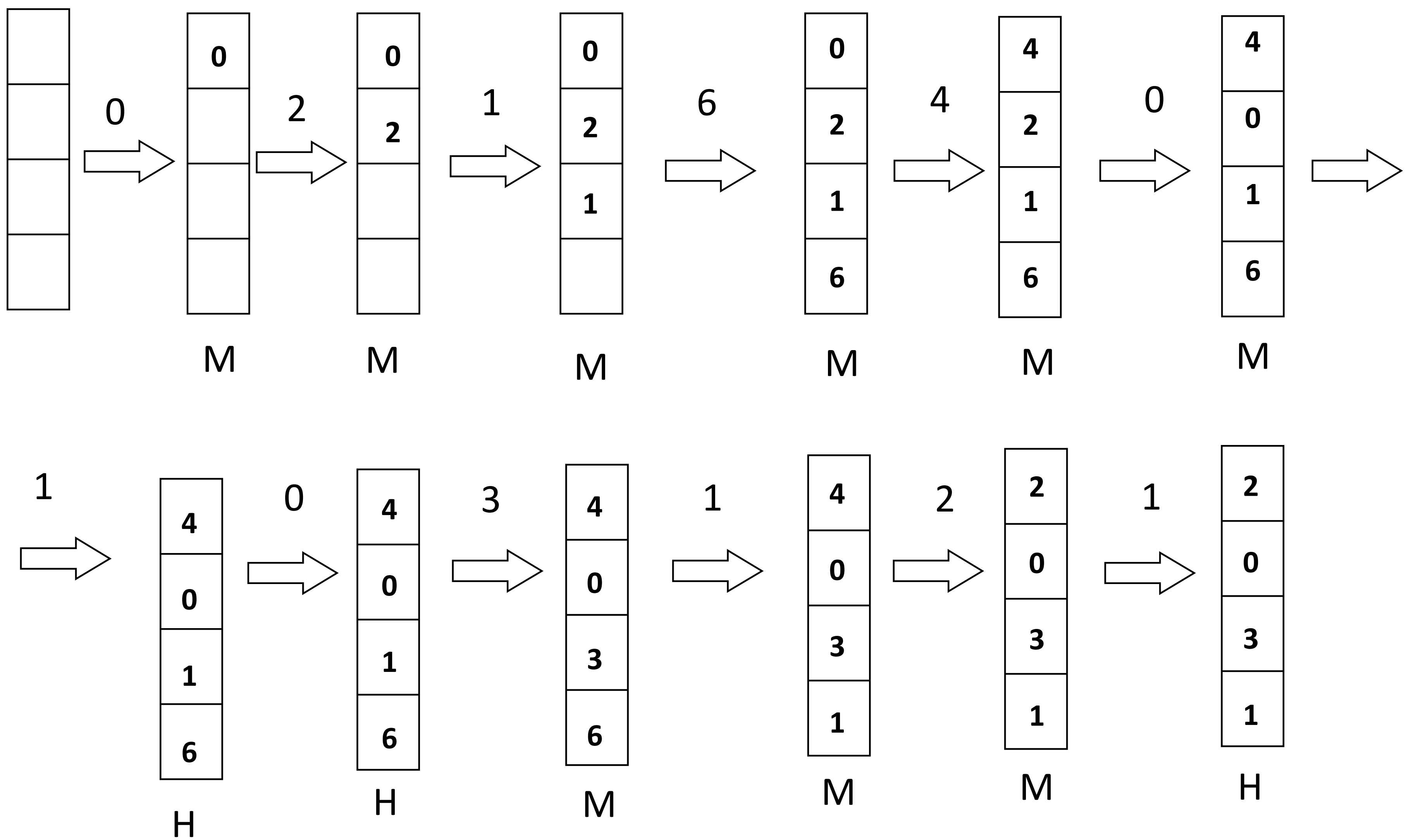
# Terminologies

- The string of memory references is called ***Reference String***
- The page size is generally fixed (say 4K), so we need to consider only the page number (***p***)
- To determine the ***number of page faults***, for a given ***reference string,*** we need to know the number of ***page frames*** (memory blocks) available

# First-In, First-Out (FIFO)

- The oldest page in physical memory is the one selected for replacement
- Very simple to implement
  - keep a list
    - victims are chosen from the tail
    - new pages in are placed at the head

Reference string: 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1



- Reference string: 7 0 1 2 0 3 0 4 2 3 0 3 1 2 0

| | 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 1 | 2 | 0 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| f3 | | | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 3 | 3 | 3 | 3 | 2 | 2 | | | |
| f2 | | 0 | 0 | 0 | 0 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | | | |
| f1 | 7 | 7 | 7 | 2 | 2 | 2 | 2 | 4 | 4 | 4 | 0 | 0 | 0 | 0 | 0 | | | |
| | M | M | M | M | H | M | M | M | M | M | M | H | M | M | H | | | |

Hit ratio = 3/15 = .20

Miss ratio = 12/15 = .80

# Solve

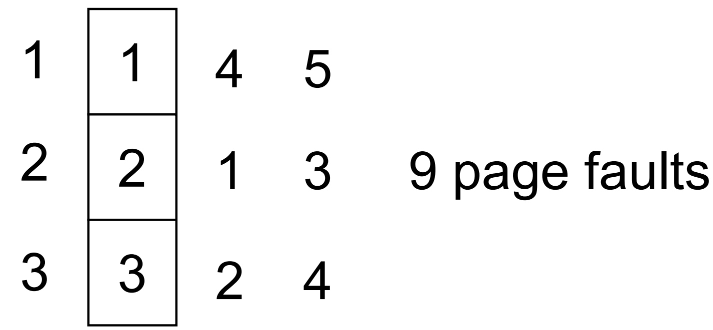
- Ref string 1 2 3 4 1 2 5 1 2 3 4 5
- With 3 frames
- With 4 frames
- Observe the result

# *Belady's Anomaly*

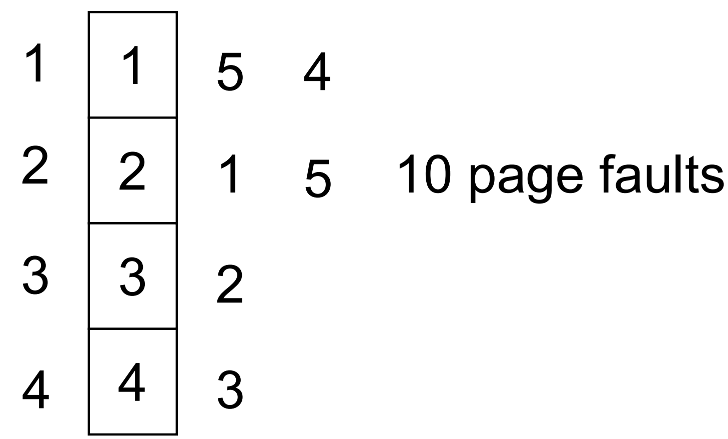
- Normally Increasing Number of frames should reduce page faults
- But the number of page faults for four frames is 10 is greater than the number of faults for three frames (9)!
- ***Belady's anomaly***: is unexpected result - in which for some page replacement algorithms, the page fault rate may ***increase*** as the number of allocated frames increases!
- FIFO exhibits Belady’s Anomaly

# 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):




- 4 frames:




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

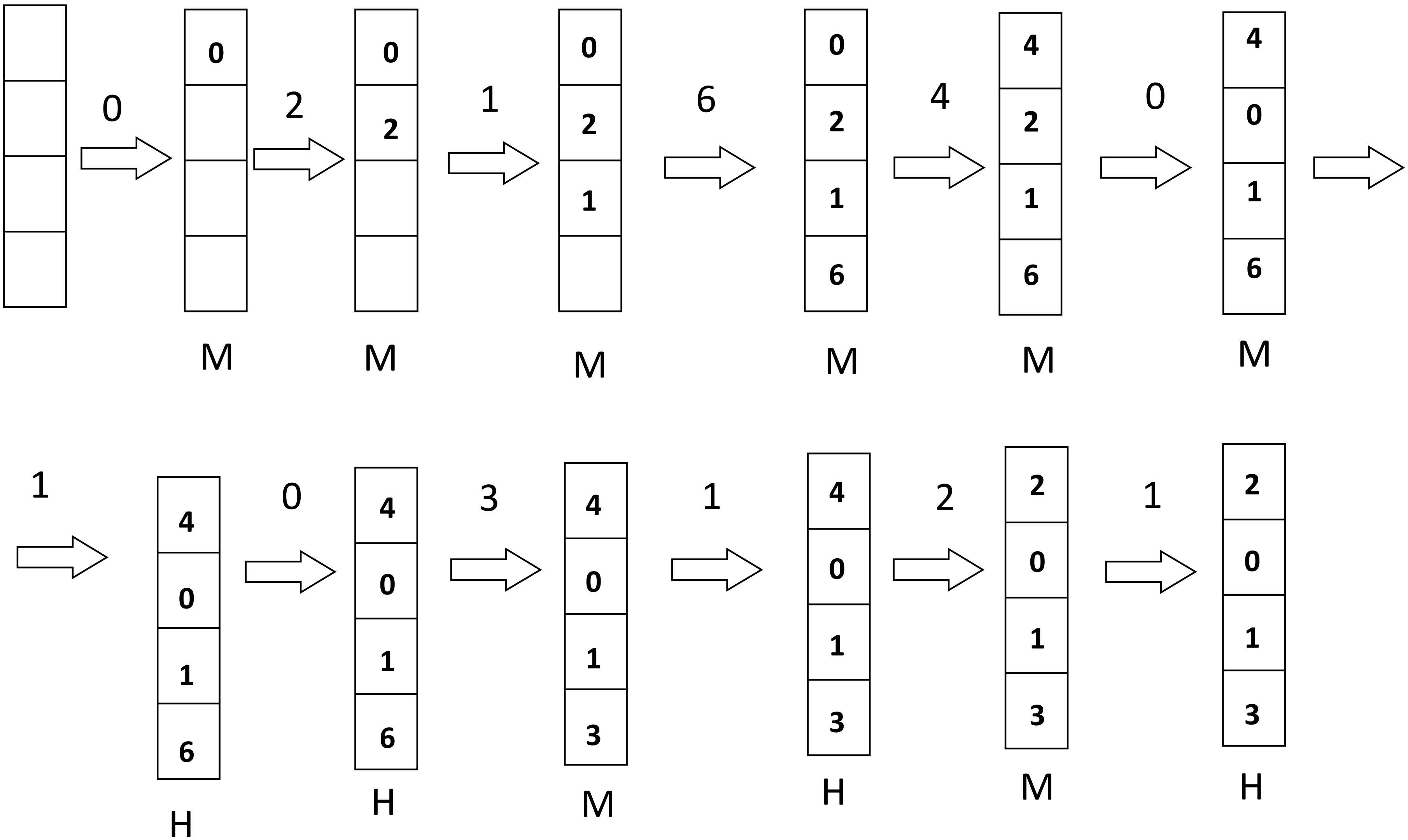
# FIFO Issues

- Poor replacement policy
- Evicts the oldest page in the system
  - usually a heavily used variable should be around for a long time
  - FIFO replaces the oldest page - perhaps the one with the heavily used variable
- FIFO does not consider page usage

# Least Recently Used (LRU)

- Basic idea
  - replace the page in memory that has not been accessed for the longest time
- Optimal policy looking back in time
  - as opposed to forward in time
  - fortunately, programs tend to follow similar behavior

Reference string: 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1



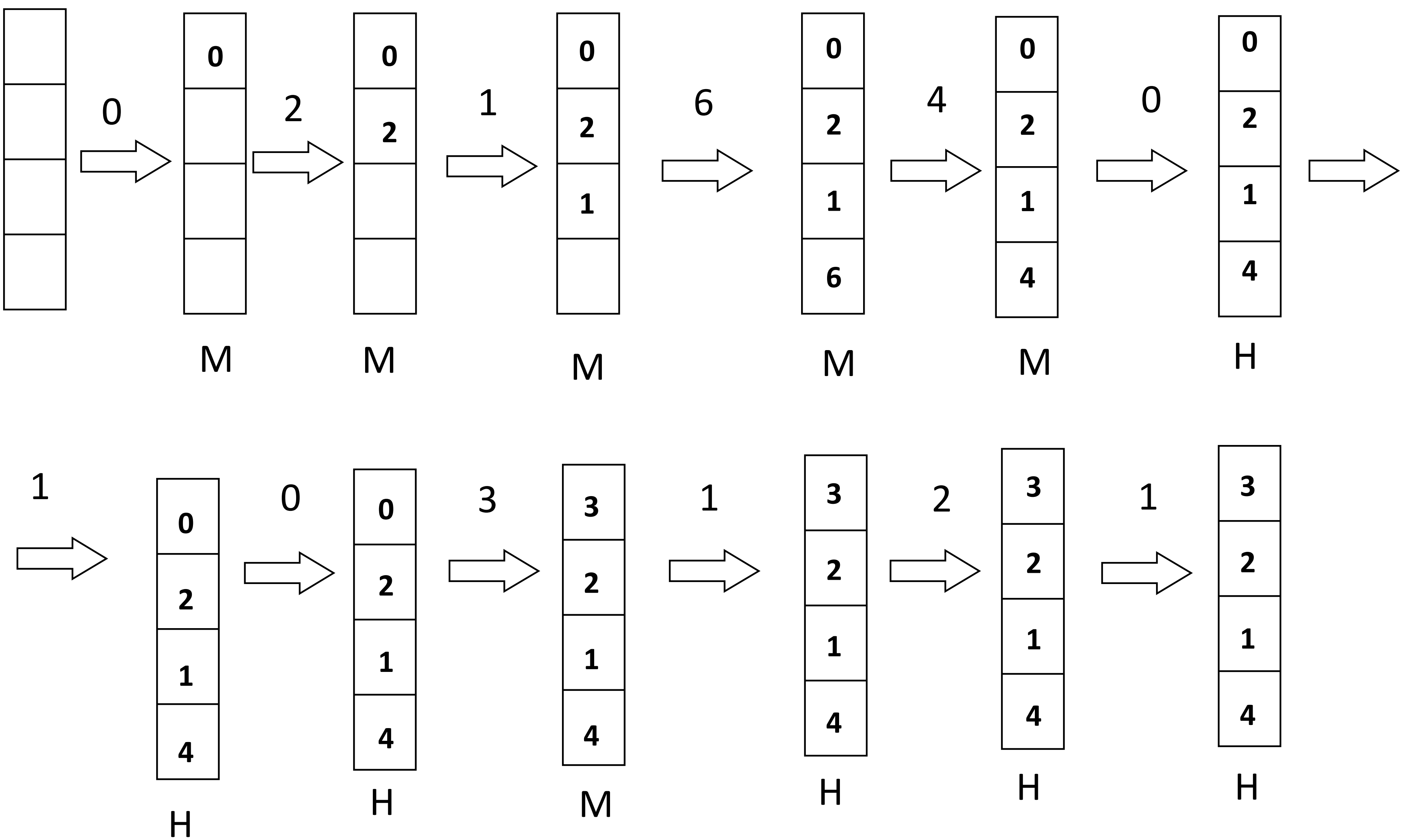
# LRU Issues

- How to keep track of last page access?
  - requires special hardware support
- 2 major solutions
  - counters
    - hardware clock “ticks” on every memory reference
    - the page referenced is marked with this “time”
    - the page with the smallest “time” value is replaced
  - stack
    - keep a stack of references
    - on every reference to a page, move it to top of stack
    - page at bottom of stack is next one to be replaced

# Optimal Page Replacement

- Basic idea
  - replace the page that will not be referenced for the longest time
- This gives the lowest possible fault rate
- Impossible to implement
- Does provide a good measure for other techniques

Reference string: 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1



# Questions ??