# Sistemas Operativos: VM Paging
# Page Faults
## Page Faults

Pedro F. Souto (pfs@fe.up.pt)

May 6, 2020

# Memory Virtualization

## Idea

**Goal** Illusion that each process has its own memory (address space)

**Mechanism** Address translation

- On every memory access, the VM subsystem maps the virtual address to a physical address

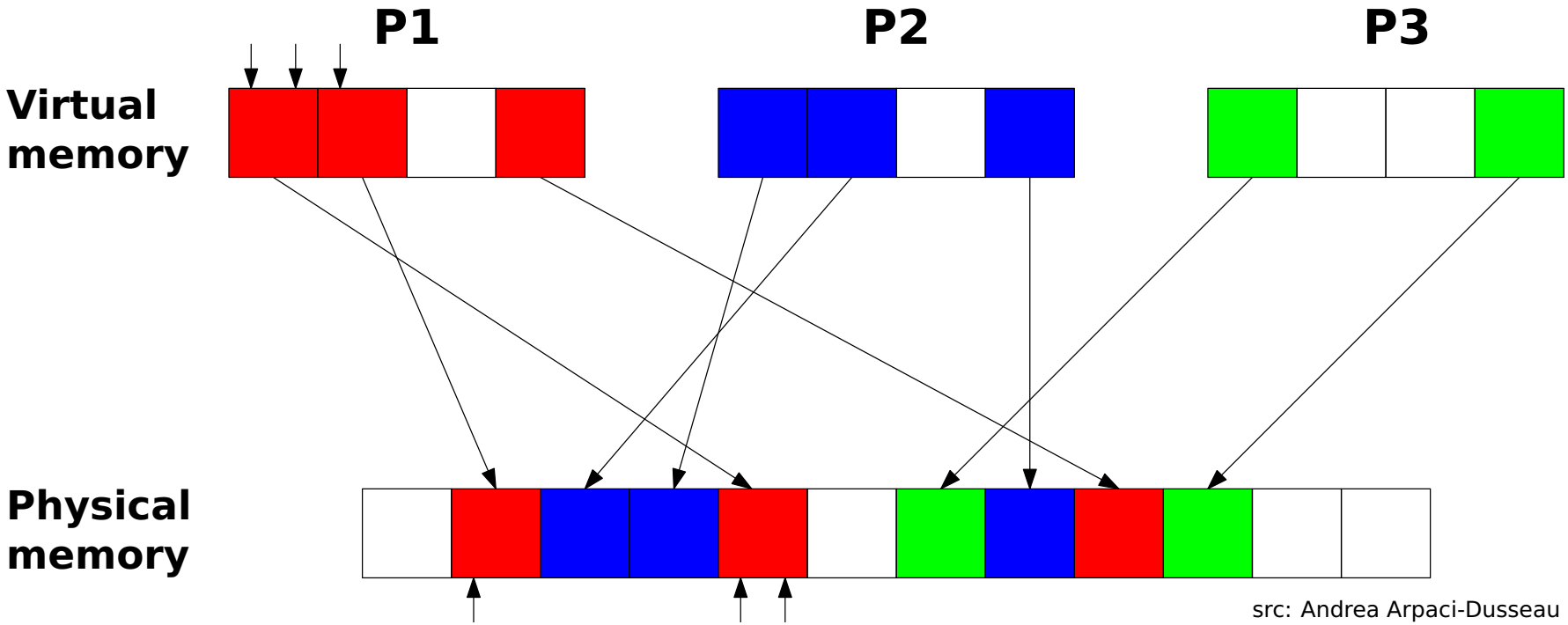
**Issue** What if the total memory used by all the processes in the system is larger than the physical memory?

- A process has lots of data
- Many processes executing simultaneously

# Paging

## Idea

1. Divide address space into *fixed-size* ($2^n$) units: *pages*
   - Typically 4KiB, but also 8KiB or even 1 MiB (super-pages)
2. Divide physical memory in same size units: *physical pages* or *page frames*
3. Map virtual pages to page frames
   - Relocate each page independently in memory.




src: Andrea Arpaci-Dusseau

# Paging: Too Small Physical Memory

**Issue** What if the total memory used by all the processes in the system is larger than the physical memory?

**Solution** Keep in:

- **Physical memory** only the pages that are being *actively* accessed;
- **Disk** the remaining pages
  - **Code** including shared libraries, is already stored on files
  - **Heap and Stack** stored in *swap space*

**Requirements**

- **Mechanisms** to keep track of where is each valid page of the address space, i.e. in main memory or in disk
- **Policies** to determine which pages are in main memory (and which are in disk)

# How to keep track of where is each valid page?

## Use the PT

- Add a **Present** bit, **P**, to each PTE

  P=1 page is present (in main memory)

  P=0 page is absent, i.e. is in disk

## Where on disk is the page, if P = 0

- The book suggests to use the field of the PTE used to store the PFN
  - Since the page is not in memory, that field is not used
- Linux uses another data structure: the address space map

  ```
  $cat /proc/<proc_no>/map
  ```

  or, more readable, but with less information:

  ```
  $pmap <proc_no>
  ```

# Virtual Address Translation: HW Implementation

1. Extract VPN from virtual address
2. Check TLB

   If TLB hit
   - Extract PFN from TLB entry (page is in memory)

   Else (TLB miss)
   - Read PTE from PT (in memory)

     If not valid raise SEGMENTATION FAULT
     If no permission raise PROTECTION FAULT
     If present
       2.1 Read PTE and insert it in TLB
       2.2 Retry instruction
     else (page absent) raise PAGE FAULT (should be PAGE MISS)

# Page Fault Handler (belongs to OS)

1. Find free page frame
   - If no page frame found (all page frames in use)
     - Select page in memory to replace
     - Update PTE of victim page (reset present bit)
     - Write victim page out to disk, if modified (need **dirty bit** to PTE)
2. Read referenced page from disk into page frame found
3. Update PTE of referenced page with PFN
   - Set present bit
4. Resume instruction execution

IMP. Handling a page fault requires going to disk. Scheduler:

1. Changes the state of faulting process to WAITING
2. Picks a READY process to run

# HW Support for Resuming Faulting Instruction

- Want page fault to be transparent to user
- Page fault may have occurred in middle of instruction
  - When instruction is being fetched
  - When data is being loaded or stored
- Requires HW support

  **Precise interrupts** stop CPU pipeline such that instructions before faulting instruction have completed, and those after can be restarted. Complexity depends on:

  **Instruction set** imagine the instruction faults after changing some memory positions, e.g. IA32/x64 "move string instructions"

  **Processor implementation**

  - E.g. on IA32/x64 these instructions change registers, so that it is fairly easy to resume them

  **Note** This is not a requirement unique to page fault handling

  - This is important for handling other exceptions as well as interrupts, as hinted by the name of the mechanism

# Virtual Memory: Policies

**Goal** Minimize the number of page faults

- Page faults require milliseconds to handle (reading and possibly writing to disk)
- There is plenty of time for OS to make a good decision

**OS has two decisions** to make

Which resident page/s in memory should be moved to disk?
When should a page/s on disk be brought into memory?

# When to bring a page from disk into memory?

**Demand paging** Only upon a page fault

**Intuition** Wait until page must absolutely be in memory

- When a process starts, no pages are loaded in memory

**Problem** Pay cost of page fault for every newly accessed page

**Prepaging (prefetching)** In anticipation of a reference to a page

- OS predicts future accesses and brings pages into memory early "in background"
  - Processes do not suffer disk access delays
- Works well for some access patterns (e.g., sequential)
- Costs of guessing wrong?

**Hints** Combine above with "program"-supplied hints about page references

- "Program" indicates: "may need page in future", "don't need this page anymore", or "sequential access pattern", ...
- Example: `madvise()` in Unix (and Linux)

# Which page to move out to disk (replace)?

OPT (Optimal) Replace page not used for longest time **in future**

Pros Guaranteed to minimize number of page faults

Cons Requires OS to predict the future!!!

LRU (Least-Recently used) Replace page not used for longest time **in past**

Rationale Use the past to predict the future

Pros With locality, LRU approximates OPT

Cons Hard to implement

- Action required on every memory access

FIFO Replace page that has been in memory the longest

Rationale First referenced a long time ago, done with it by now

Pros Fair (all pages receive equal treatment) and easy to implement

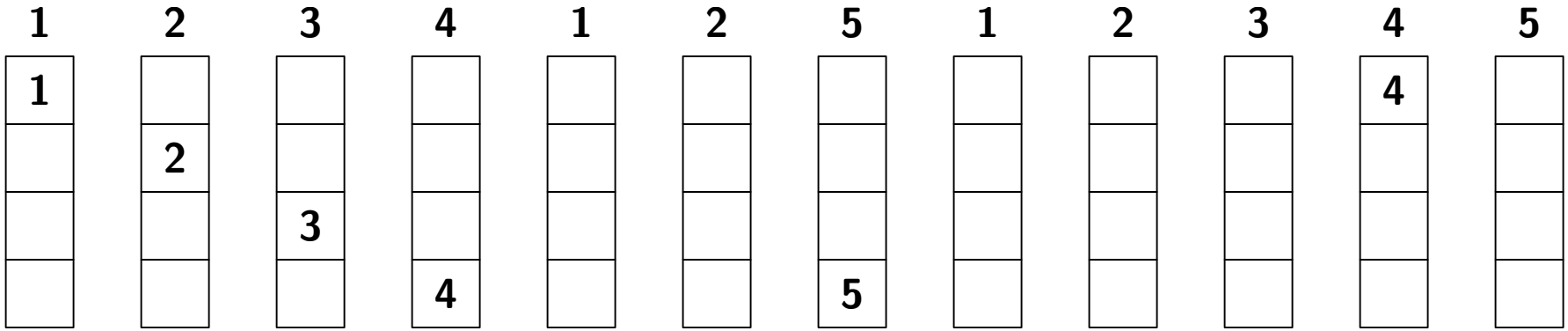
Cons Some pages may always be needed

Random Pick any page in memory and evict it to disk

# OPT

Page access sequence 1, 2, 3, 4, 1, 2 , 5, 1, 2, 3, 4, 5

# of frames 4




# of page faults 4 + 2 = 6

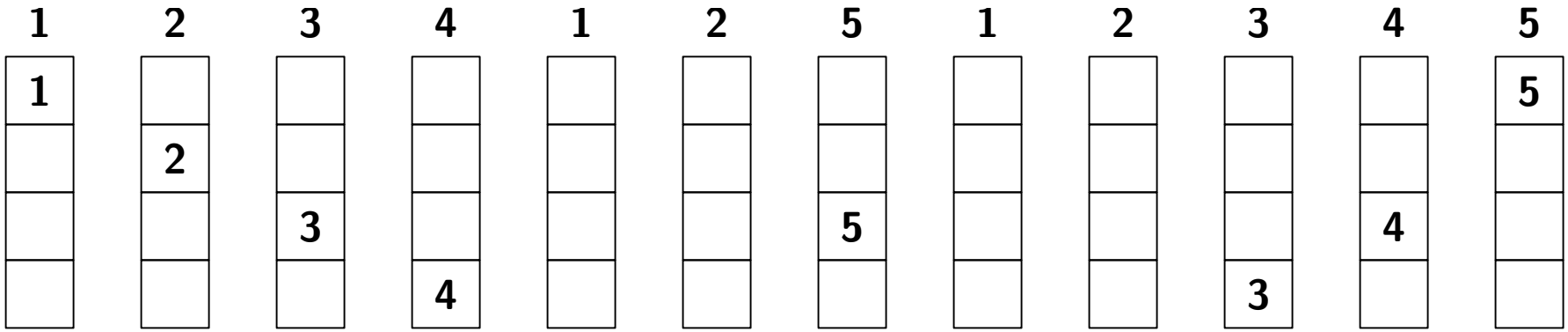
Usefulness Provides optimal results, i.e. minimum number of page-faults

- Used as a benchmark for the evaluation of other page replacement algorithms

# LRU

Page access sequence 1, 2, 3, 4, 1, 2 , 5, 1, 2, 3, 4, 5

# of frames 4




# page faults 4 + 4 = 8

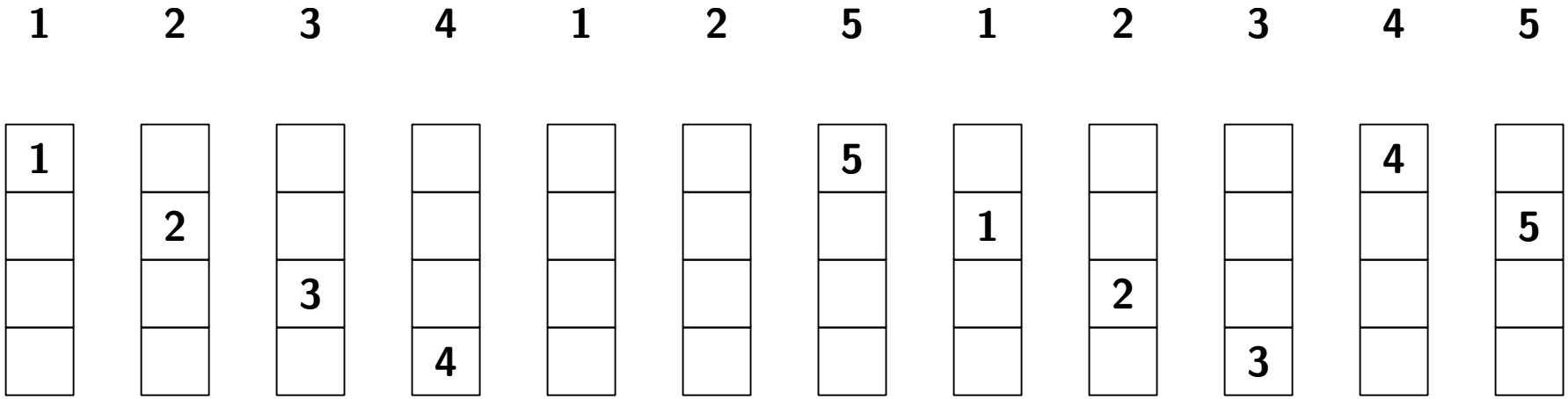
Hard to implement

- HW implementation would require too many resources
- SW implementation would require too much time
- In practice, use an approximation (see clock algorithm, below)

# FIFO

Page access sequence 1, 2, 3, 4, 1, 2 , 5, 1, 2, 3, 4, 5

# of frames 4



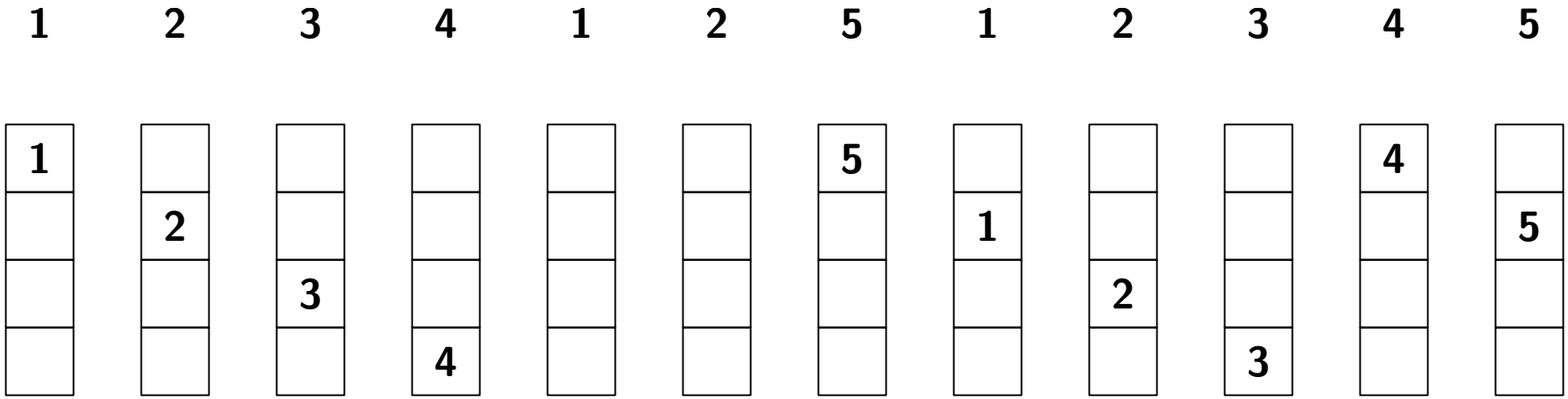

# page faults 4 + 6 = 10

What if the # of frames is 3?

# FIFO

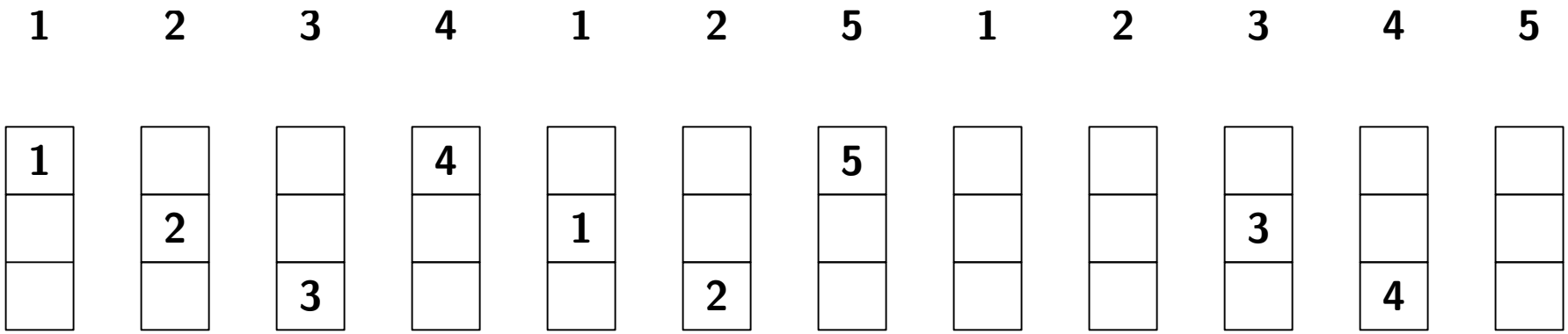
Page access sequence 1, 2, 3, 4, 1, 2 , 5, 1, 2, 3, 4, 5

# of frames 4




# page faults 4 + 6 = 10

What if the # of frames is 3?




# page faults 3 + 6 = 9, i.e. less than for 4 frames

# Belady's Anomaly

Intuitively the larger the number of frames, the less page faults a system should experience

- Hence, the larger the main memory, faster the system

However some algorithms, like FIFO, incur more page faults when the number of frames increases – this is known as the **Belady's anomaly**

Stack algorithms are page replacement algorithms where:

$$M(n, r) \subseteq M(n + 1, r), \forall n, r$$

where $M(f, r)$ is the memory contents with $f$ *frames* after $r$ memory references.

- Thus, algorithms of this class, e.g. LRU, do not suffer from the Belady's anomaly

# LRU Implementation

## Software

- OS maintains list of physical pages ordered by reference time
- When page is referenced, move it to the head of the list
- When need a victim: pick the page in the last position

## Hardware

- Associate a timestamp with each page
- When page is referenced, update page timestamp (fast)
- When need a victim: scan through the timestamps to find oldest (slow)

## In practice Approximate LRU

- After all, LRU is an approximation of OPT

# LRU Approximation: The Clock Algorithm

## Hardware

- **use/reference** bit in the PTE
- Upon memory reference to page, HW sets use/reference bit

## OS

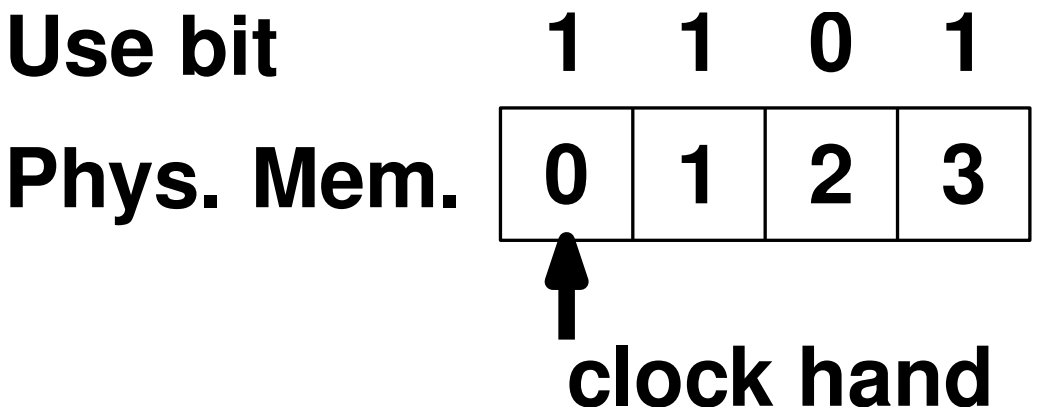
### Idea

replace pages with use/reference bit cleared

### Implementation

- Keep a pointer to the last valid PTE examined
- Scan PTEs circularly
- If use bit is set, clear it
- If use bit is cleared, replace page

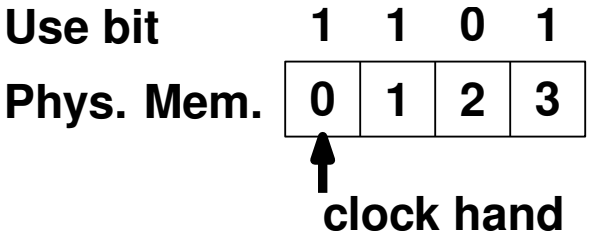
# Clock: Looking for a page (1/3)



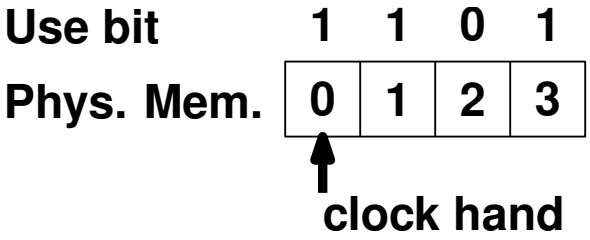

Event running process has a page fault

OS which page to evict?

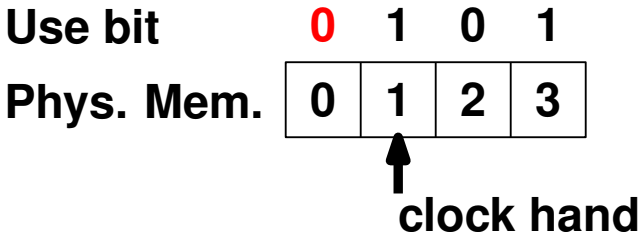
# Clock: Looking for a page (2/3)



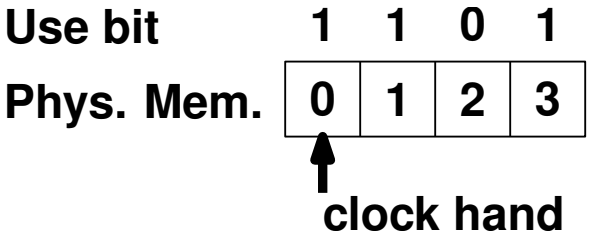
# Clock: Looking for a page (2/3)



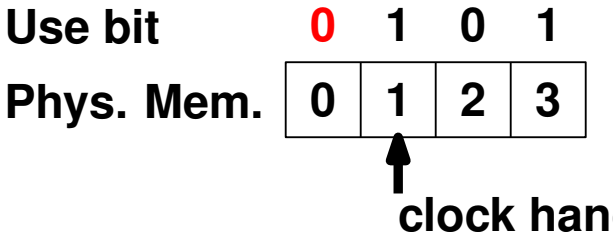

1. Clear bit for frame 0 and advance clock hand



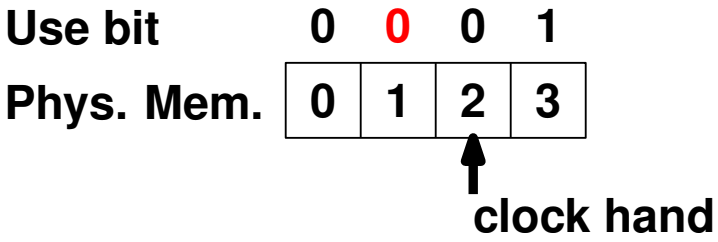
# Clock: Looking for a page (2/3)



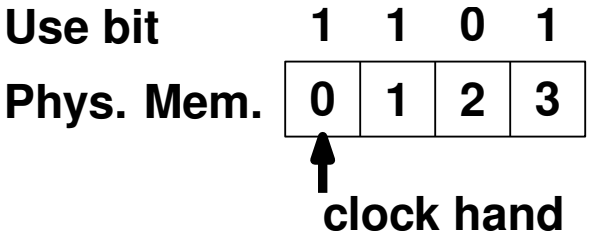

1. Clear bit for frame 0 and advance clock hand



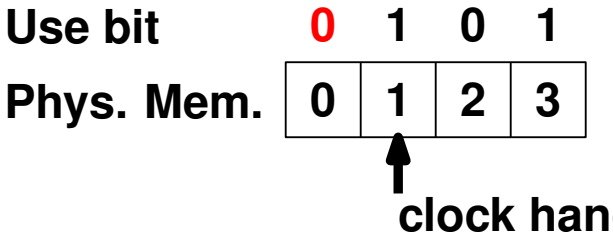

2. Clear bit for frame 1 and advance clock hand



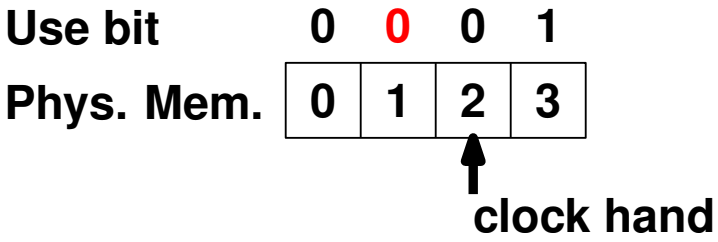
# Clock: Looking for a page (2/3)



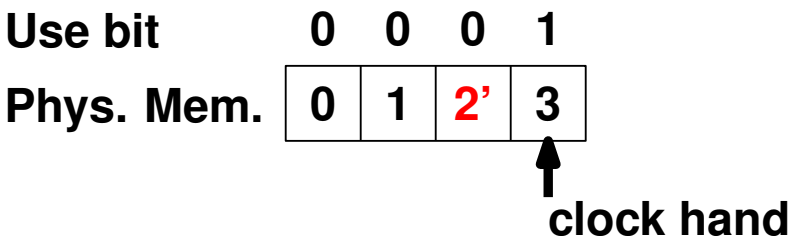

1. Clear bit for frame 0 and advance clock hand



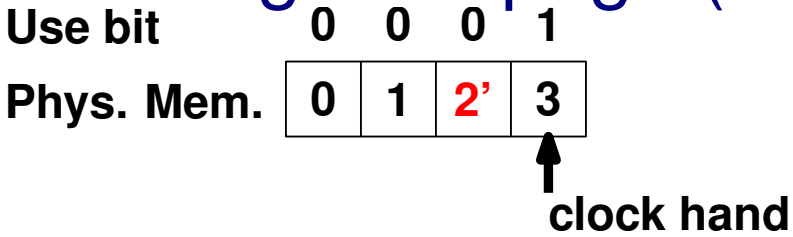

2. Clear bit for frame 1 and advance clock hand



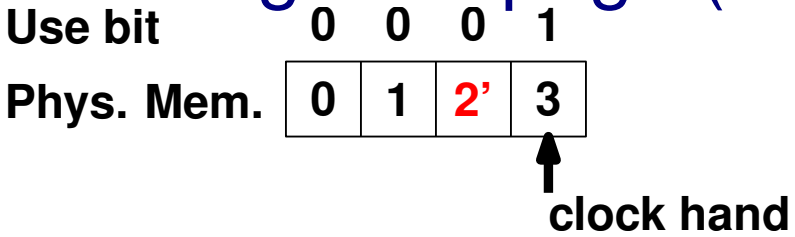

3. Evict page in frame **2**: its use bit is cleared
4. Load the faulting page to frame **2** and advance clock hand



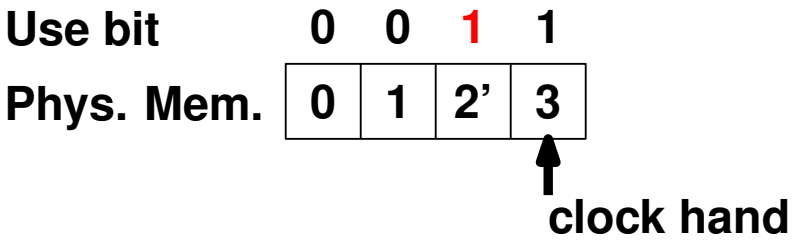
# Clock: Looking for a page (3/3)



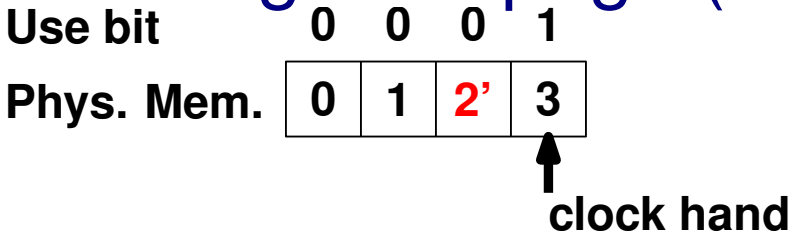
# Clock: Looking for a page (3/3)



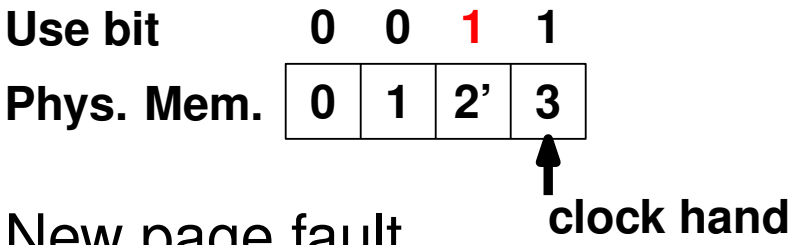

5. Access to paged in page (HW sets use bit)



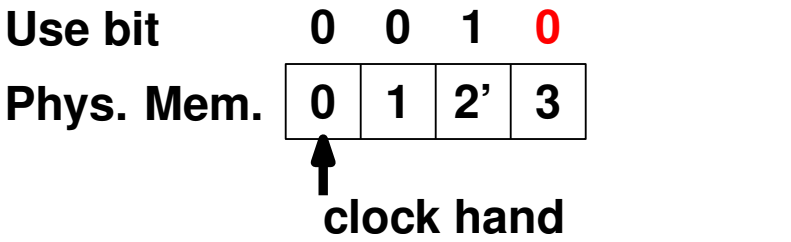
# Clock: Looking for a page (3/3)



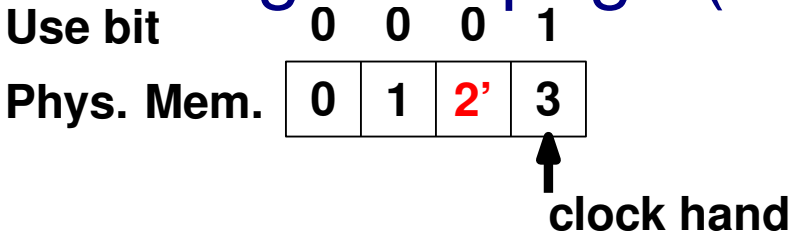

5. Access to paged in page (HW sets use bit)



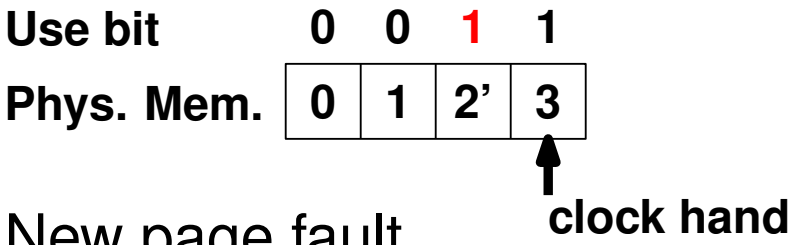

6. New page fault
7. Clear bit 1 of page in frame 3 and advance clock hand



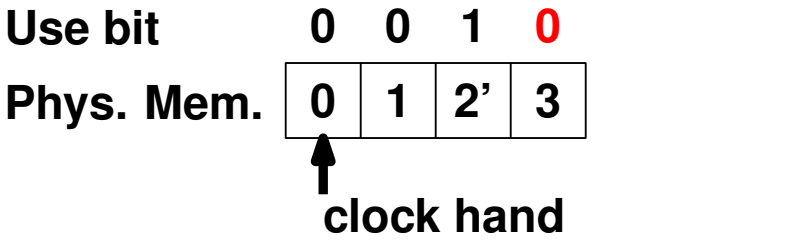
# Clock: Looking for a page (3/3)



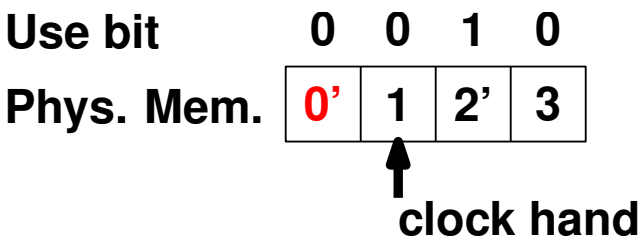

5. Access to paged in page (HW sets use bit)




6. New page fault
7. Clear bit 1 of page in frame 3 and advance clock hand




1. Evict page in frame **0**: its use bit is cleared
2. Load the faulting page to frame **0** and advance clock hand



# The Clock Algorithm: Extensions

## Replace multiple pages at once

**Issue** Expensive to run replacement algorithm and to write single page to disk

**Fix** Evict multiple victims each time and track free list

## Second/third/... chance

**Issue** Use bit can be crude

**Fix** Use a sofware counter (`chance`)

**If use bit is cleared** increment the `chance` counter

**If the `chance` counter exceeds bound** replace page

## Use dirty bit

**Issue** Dirty pages are more expensive to replace

- They must be written to disk

**Fix** Replace first pages that have both the use and the dirty bits cleared

# Thrashing

**Working-set** set of pages that a process actively accesses

- When the working set is in memory, there are no page-faults
- Initially, the page-fault rate of a process is relatively high, since its working set is not in memory
- The working set changes (slowly) with time, thus, even with lots of memory, a process has a residual page-fault

**When there is not enough memory** i.e. the sum of the working sets of all processes exceeds the size of physical memory, the system starts **thrashing**, and processes slow down or even halt

- To get a free frame for a page to bring into memory, the OS pages out a page that it will have to bring in soon
  - It will be accessed by the process to which it belongs

**Fixes**

**Buy more memory** as long as your page replacement algorithm does not suffer from the Belady's anomaly

**Reduce the multiprogramming degree** i.e. the number of running processes