# Page replacement algorithms

## When a page fault occurs

- OS has to choose a page to evict from memory
- If the page has been modified, the OS has to schedule a disk write of the page
- The page just read overwrites a page in memory (e.g. 4Kbytes)
- Clearly, it's better not to pick a page at random
- Same problem applies to memory caches

# Benchmarking

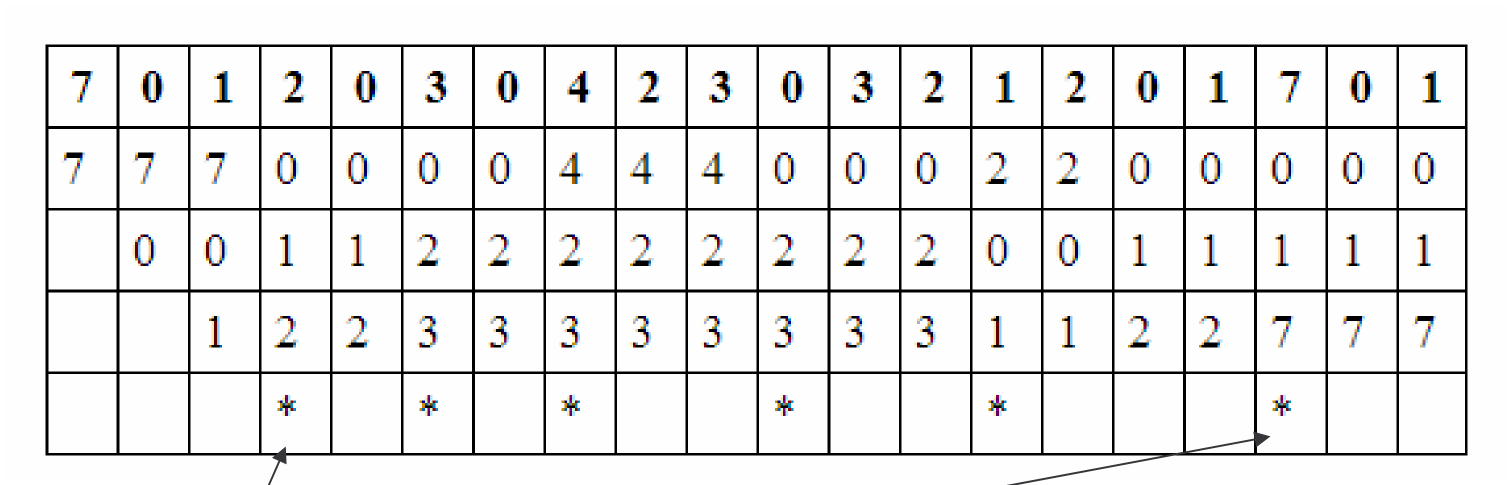
- Tests are done by generating page references (either from real code or random)
- Sequences of page numbers (no real address, no offset)
- Example:

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |

# Optimal page replacement

- At the moment of page fault:
  - Label each page in memory is labeled with the number of instructions that will be executed before that page is first referenced
  - Replace the page with the highest number: i.e. postpone as much as possible the next page fault
- Nice, optimal, but unrealizable
  - The OS can't look into the future to know how long it'll take to reference every page again

# Example: optimal

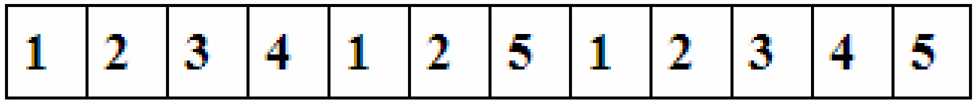


| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Sequence | 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |
| Phys mem | 7 | 7 | 7 | 0 | 0 | 0 | 0 | 4 | 4 | 4 | 0 | 0 | 0 | 2 | 2 | 0 | 0 | 0 | 0 | 0 |
| | | 0 | 0 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| | | | 1 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 1 | 1 | 2 | 2 | 7 | 7 | 7 |
| PF | | | | * | | * | | * | | | * | | | * | | | | * | | |

6 page faults

## Belady's anomaly

Try this sequence



| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 1 | 2 | 5 | 1 | 2 | 3 | 4 | 5 |

With 3 page frames

With 4 page frames

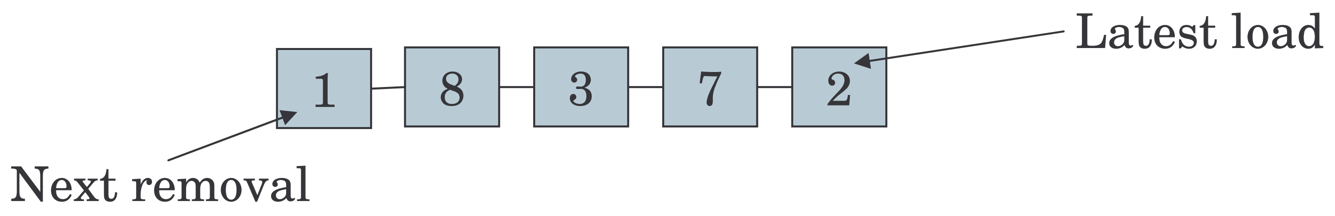
With FIFO, with the optimal algorithm, (later) with the LRU

# “Not recently used” algorithm

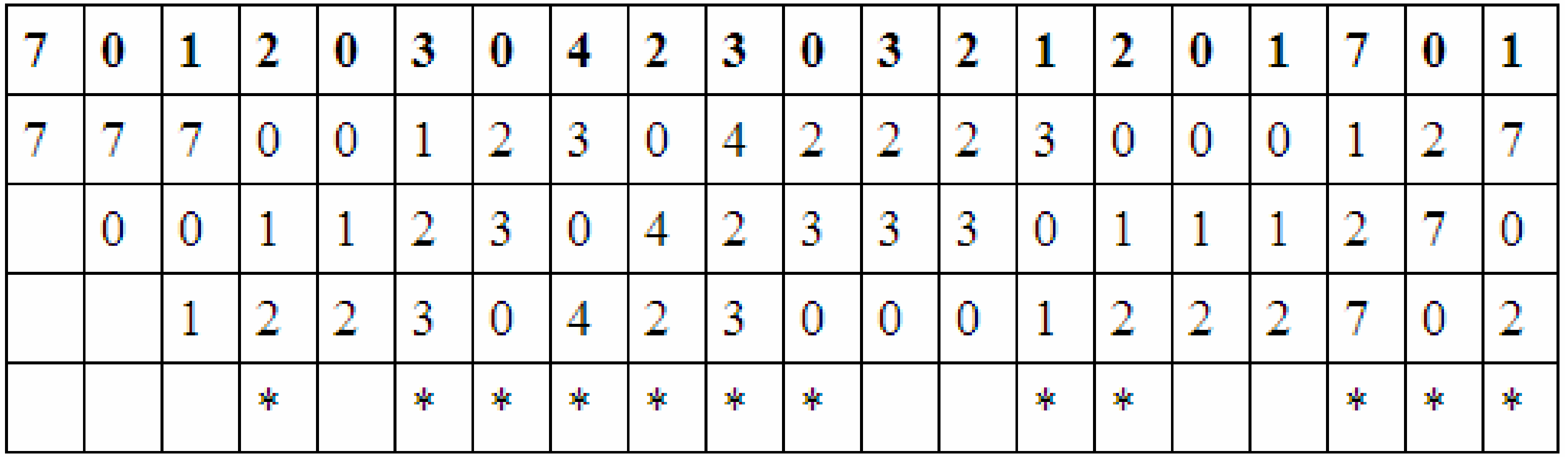
- Use Referenced and Modified bits
- R&M are in hardware, potentially changed at each reference to memory
  - R&M are zero when process is started
- On clock interrupt the R bit is cleared
- On page fault, to decide which page to evict:
  - Classify:
    - Class 0 – R=0,M=0
    - Class 1 – R=0,M=1
    - Class 2 – R=1,M=0
    - Class 3 – R=1,M=1
  - Replace a page at random from the lowest class

# FIFO replacement

- FIFO, first in first out for pages
- Clearly not particularly optimal
- It might end up removing a page that is still referenced since it only looks at the page’s age
- Rarely used in pure form…



# Example (FIFO)

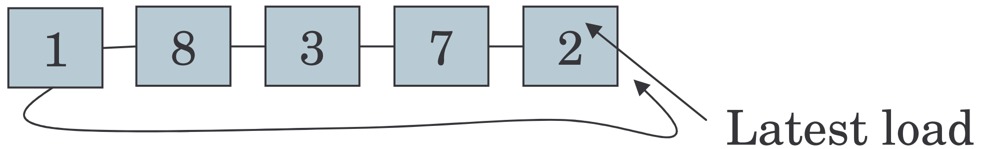


| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Sequence | **7** | **0** | **1** | **2** | **0** | **3** | **0** | **4** | **2** | **3** | **0** | **3** | **2** | **1** | **2** | **0** | **1** | **7** | **0** | **1** |
| Phys mem | 7 | 7 | 7 | 0 | 0 | 1 | 2 | 3 | 0 | 4 | 2 | 2 | 2 | 3 | 0 | 0 | 0 | 1 | 2 | 7 |
| | | 0 | 0 | 1 | 1 | 2 | 3 | 0 | 4 | 2 | 3 | 3 | 3 | 0 | 1 | 1 | 1 | 2 | 7 | 0 |
| | | | 1 | 2 | 2 | 3 | 0 | 4 | 2 | 3 | 0 | 0 | 0 | 1 | 2 | 2 | 2 | 7 | 0 | 2 |
| PF | | | | * | | * | * | * | * | * | * | | | * | * | | | * | * | * |

12 page faults

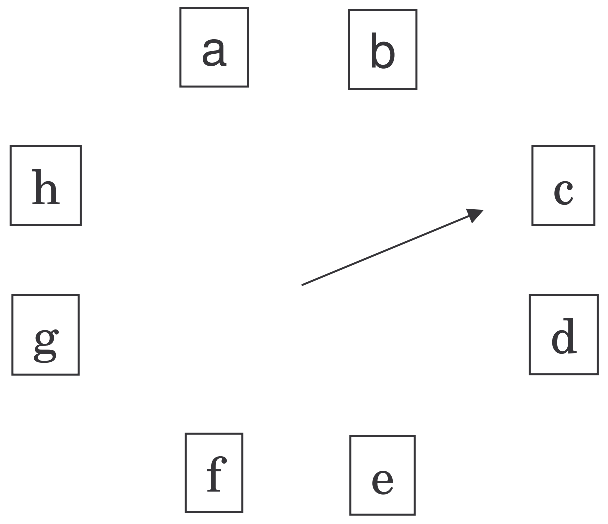
## “Second chance” algorithm

- Like FIFO but…
- Before throwing out a page checks the R bit:
  - If 0 remove it
  - If 1 clear it and move the page to the end of the list (as it were just been loaded)
  - If all pages have R=1, eventually the algorithm degenerates to FIFO (why?)



## Clock page algorithm

- Like “second chance” but…
- …implemented differently:
  - Check starting from the latest visited page
  - More efficient: doesn’t have to move list’s entries all the time

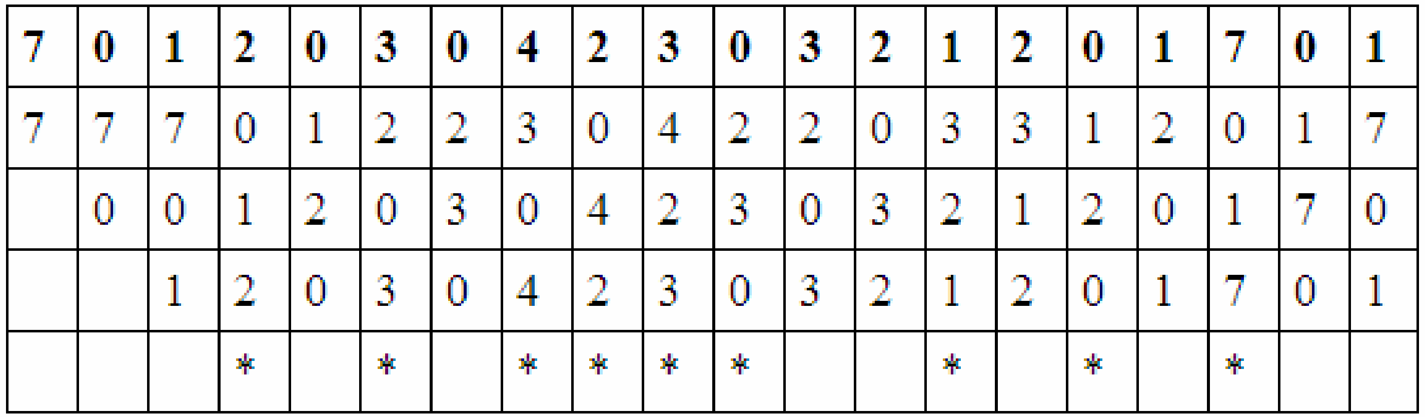


# Least recently used (LRU)

- Pages recently used tend to be used again soon (on average)
- Idea! Get a counter, maybe a 64bit counter
- Store the value of the counter in each entry of the page table (last access time to the page)
- When is time to remove a page, find the lowest counter value (this is the LRU page)

- Nice & good but expensive: it requires dedicated hardware

# Example LRU



| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Sequence | **7** | **0** | **1** | **2** | **0** | **3** | **0** | **4** | **2** | **3** | **0** | **3** | **2** | **1** | **2** | **0** | **1** | **7** | **0** | **1** |
| Phys mem | 7 | 7 | 7 | 0 | 1 | 2 | 2 | 3 | 0 | 4 | 2 | 2 | 0 | 3 | 3 | 1 | 2 | 0 | 1 | 7 |
| | | 0 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 |
| | | | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |
| PF | | | | * | | * | | * | * | * | * | | | * | | * | | * | | |

9 page faults

## NFU algorithm

- Since LRU is expensive
- NFU: “Not Frequently Used” algorithm
- At each clock interrupt add the R bit to a counter for each page: i.e. count how often a page is referenced
- Remove page with lowest counter value
- Unfortunately, this version tends not to forget anything

## Aging (NFU + forgetting)

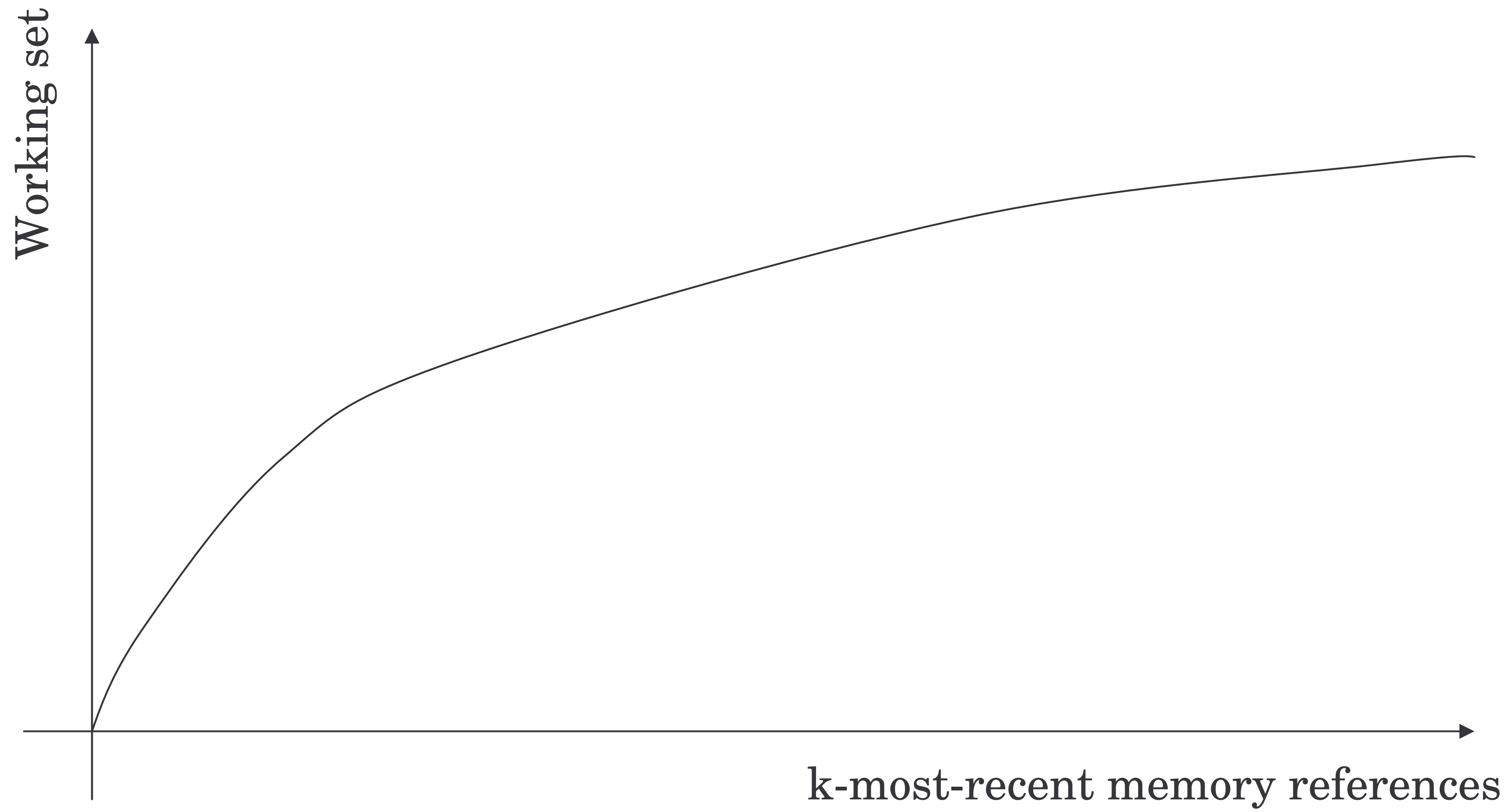
- Take NFU but…
- At each clock interrupt:
  - Right shift the counters (divide by 2)
  - Add the R bit to the left (MSB)
- As for NFU remove pages with lowest counter

- Note: this is different from LRU since the time granularity is a clock tick and not every memory reference!

## Process' behavior

- Locality of reference: most of the time the last *k* references are within a finite set of pages < a large address space
- The set of pages a process is currently using is called the *working set* of the process
- Knowing the working set of processes we can do very sophisticate things (e.g. pre-paging)

# Working set



## WS based algorithm

- Store time information in the table entries
- At clock interrupt handle R bits as usual (clear them)
- At page fault, scan entries:
  - If R=1 just store current time in the entry
  - If R=0 compute "current-last time page was referenced" and if > *threshold* the page can be removed since it's no longer in the working set (not used for *threshold* time)
- **Note**: we're using time rather than actual memory references

# WSClock algorithm

- Use the circular structure (as seen earlier)
- R=1, page in the WS – don’t remove it
- R=0, M=0 no problem (as before)
- M=1, schedule disk write appropriately to procrastinate as long as possible a process switch
  - No write is schedulable (R=1 always), just choose a clean page

# Summary

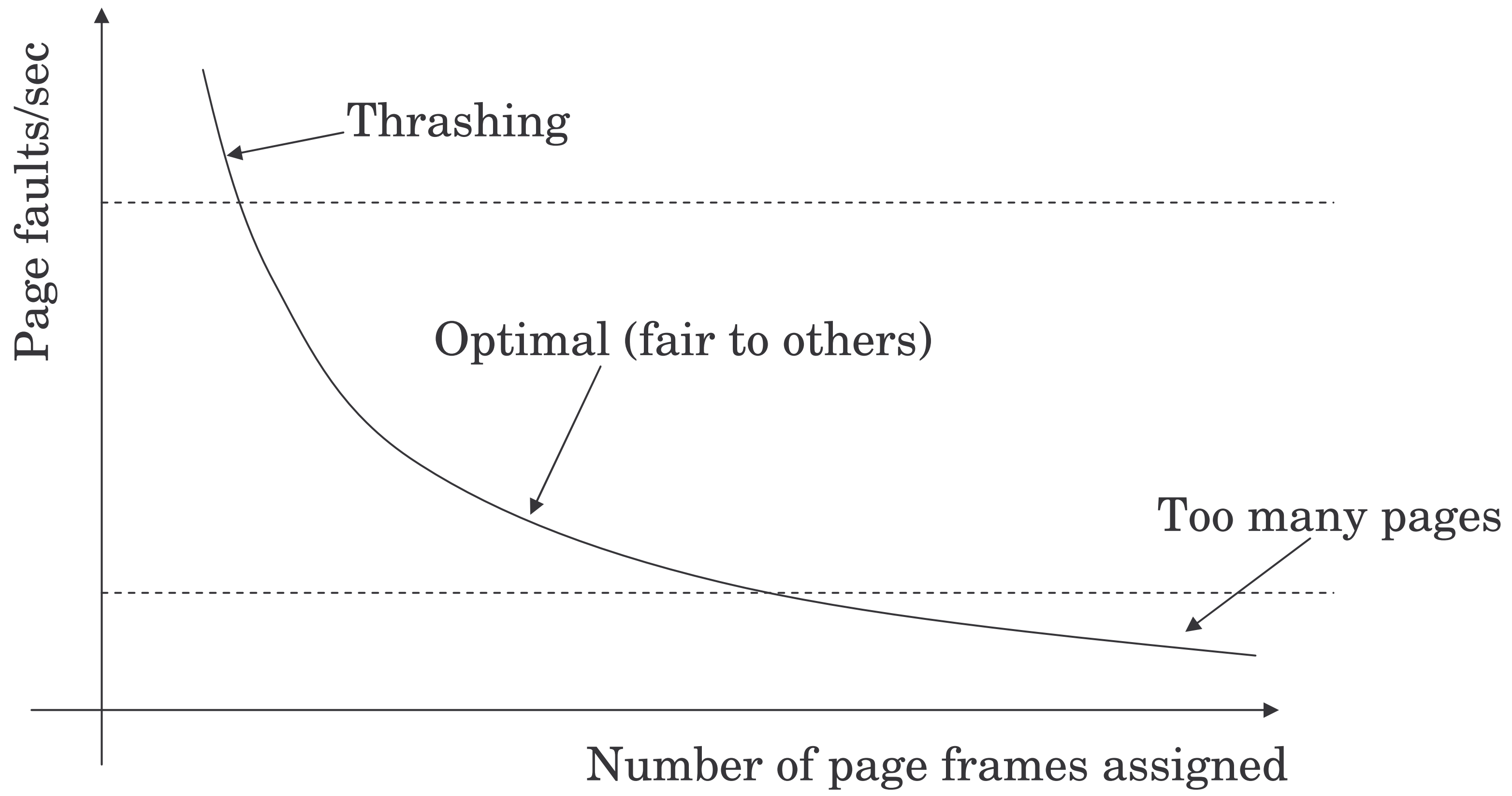
| Algorithm | Comment |
|---|---|
| Optimal | Not implementable, useful for benchmarking |
| NRU (Not recently used) | Very crude |
| FIFO | Might throw out important pages |
| Second chance | Big improvement over FIFO |
| Clock | Realistic (better implementation) |
| LRU (Least Recently Used) | Excellent but difficult to implement |
| NFU | Crude approx to LRU |
| Aging | Efficient in approximating LRU |
| Working set | Expensive to implement |
| WSClock | Good and efficient |

## Design issues

## Design issues

- Local vs. global allocation policy
  - When a page fault occurs, whose page should the OS evict?
- Which process should get more or less pages?
  - Monitor the number of page faults for every process (PFF – page fault frequency)
  - For many page replacement algorithms, the more pages the less page faults

# Page fault behavior



## Load control

- If the WS of all processes > memory, there's *thrashing*
- E.g. the PFF says a process requires more memory but none require less
- Solution: swapping – swap a process out of memory and re-assign its pages to others

## Page size

- Page size $p$, $n$ pages of memory
- Average process size $s$, in pages $s/p$
- Each entry in the page table requires $e$ bytes
- On average $p/2$ is lost (fragmentation)
- Internal fragmentation: how much memory is not used within pages
- Wasted memory: $p/2 + se/p$
- Minimizing it yields the optimal page size (under simplifying assumptions)

## Two memories

- Separate data and program address spaces
- Two independent spaces, two paging systems
- The linker must know about the two address spaces

## Other issues

- Shared pages, handle shared pages (e.g. program code)
  - Sharing data (e.g. shared memory)
- Cleaning policy
  - Paging algorithms work better if there are a lot of free pages available
  - Pages need to be swapped out to disk
  - Paging daemon (write pages to disk during spare time and evict pages if there are to few)

## Page fault handling

1. Page fault, the HW traps to the kernel
    1. Perhaps registers are saved (e.g. stack)
2. Save general purpose microprocessor information (registers, PC, PSW, etc.)
3. The OS looks for which page caused the fault (sometimes this information is already somewhere within the MMU)
4. The system checks whether the process has access to the page (otherwise a protection fault is generated, and the process killed)
5. The OS looks for a free page frame, if none is found then the replacement algorithm is run
6. If the selected page is dirty (M=1) a disk write is scheduled (suspending the calling process)
7. When the page frame is clean, the OS schedules another transfer to read in the required page from disk
8. When the load is completed, the page table is updated consequently
9. The faulting instruction is backed up, the situation before the fault is restored, the process resumes execution