Virtual Memory

COMP 3361: Operating Systems I

Winter 2015

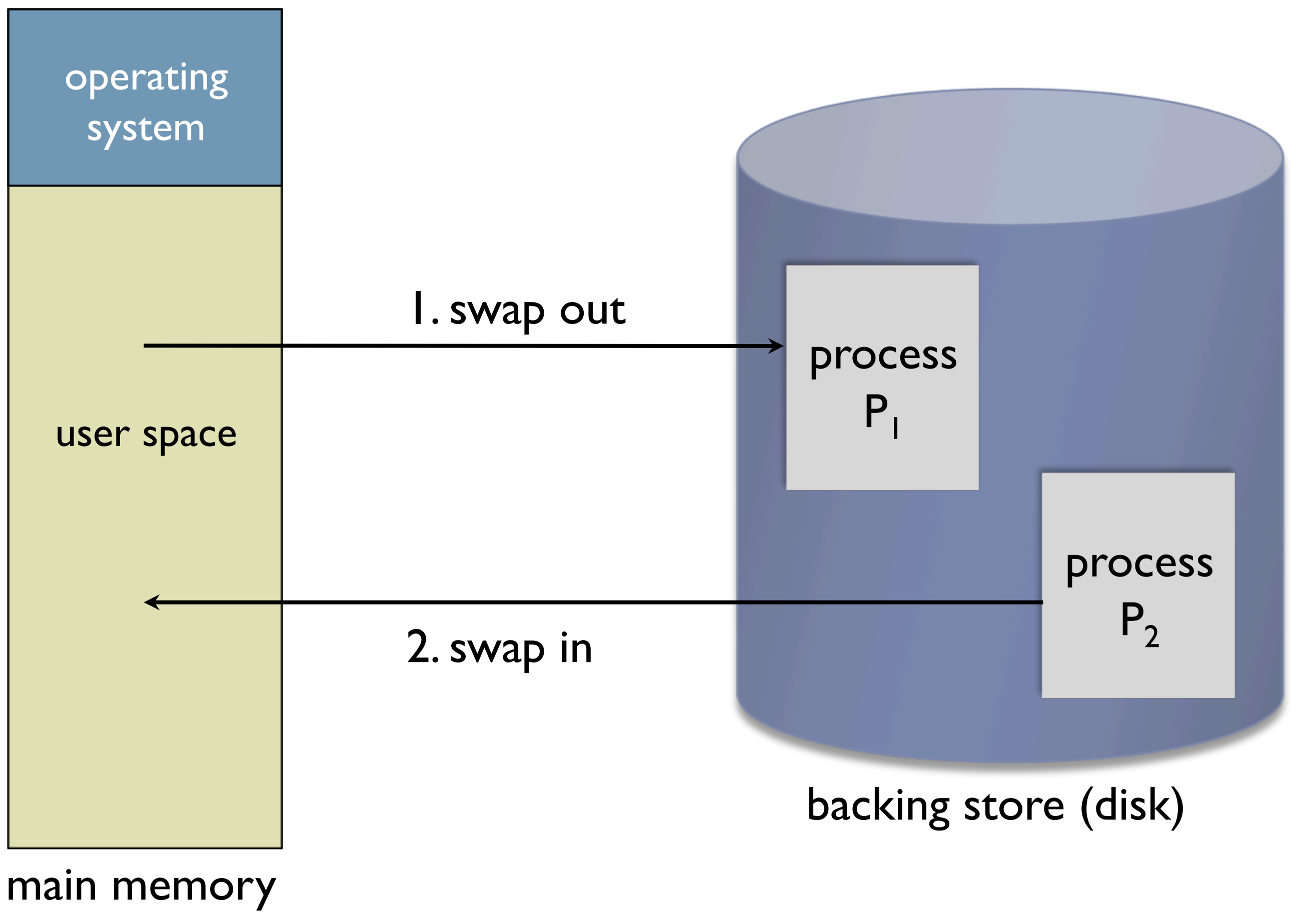
http://www.cs.du.edu/3361

Till Now…

- Process is unaware of amount of physical memory installed
  - we are using logical address spaces
- We assume that there is enough physical memory to hold all running processes
- What if there is not enough physical memory for all processes?
- Can we run a process without requiring it to be in physical memory in its entirety?
  - swapping
  - virtual memory

Swapping

- Swap processes in and out of the disk

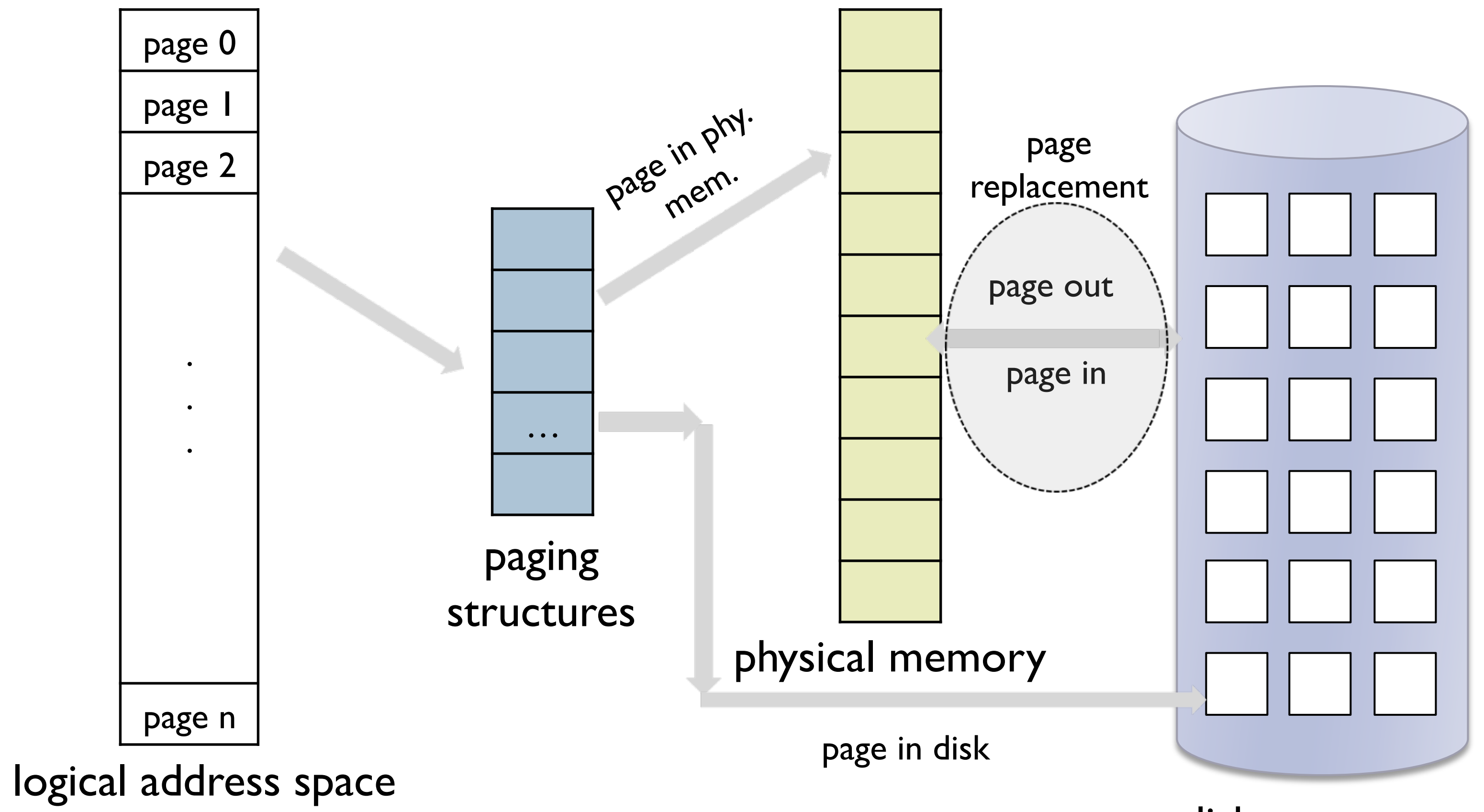


Virtual Memory

- Only a part of the program needs to be in memory for execution
  - keep only some pages of the process in physical memory
  - others kept in a special disk space (called **swap space**)
- Use paging structures to indicate that a certain page is not in physical memory
- Bring page in from swap space when needed
- Swap page out to swap space when low in memory
  - process used up maximum number of allowed physical frames

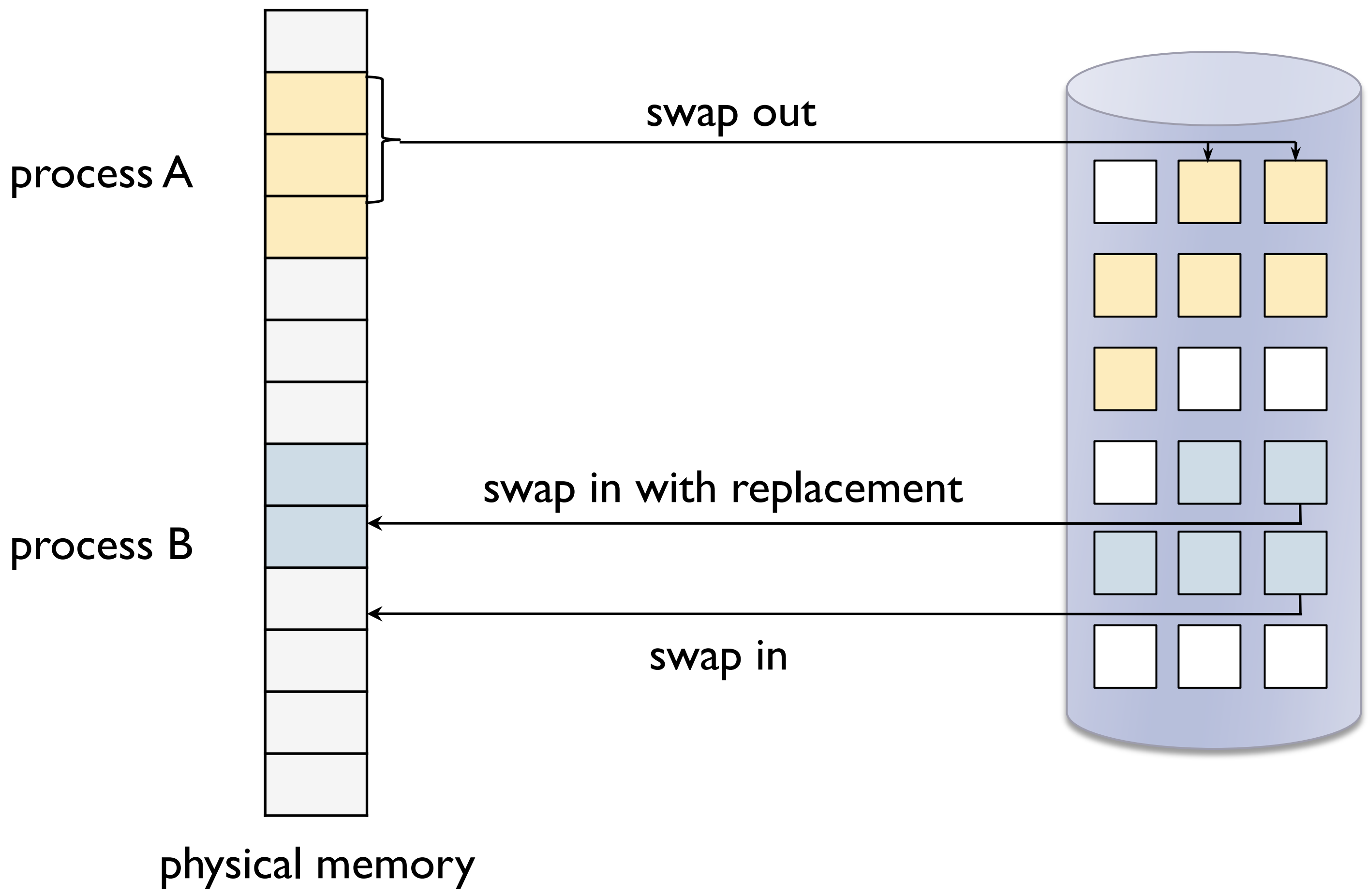
4

Simulating a Larger Physical Memory



Swapping Pages In and Out

Similar to swapping entire processes



Page Faults

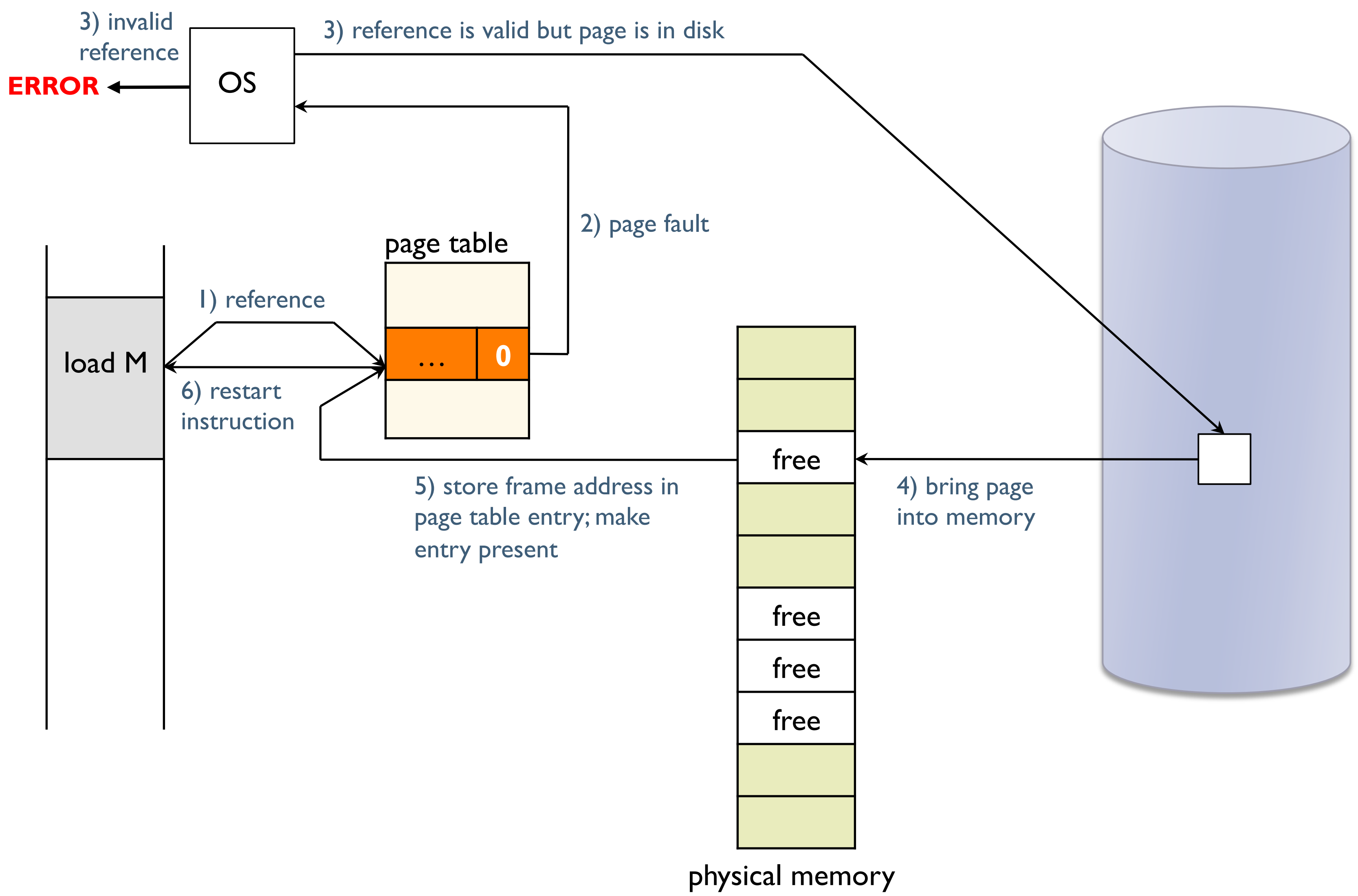
- A **Present** bit is associated with each page table entry
  - **1** $\Rightarrow$ in-memory, **0** $\Rightarrow$ not-in-memory
- During address translation, if present bit in page table entry is
  - **1** $\Rightarrow$ access frame
  - **0** $\Rightarrow$ **page fault**
- Page fault means you are trying to access a page that is not in memory
  - could be because page is not mapped
  - could be because page is in swap space
- An exception is thrown if a page fault happens

What To Do On A Page Fault?

- Operating system must decide if
  - page fault address is unmapped ⇒ abort (segmentation fault)
  - page is in swap space
- OS can store swap space location in the page table entry itself when page is swapped out

- Find a free frame (with or without replacement)
- Read in desired page into allocated frame
- Modify page table entry
- Restart the instruction that caused the page fault
  - program counter points to faulting instruction (*hardware must do this automatically*)

Handling A Page Fault


3) invalid reference
ERROR
OS
3) reference is valid but page is in disk
2) page fault
page table
1) reference
load M
...
0
6) restart instruction
free
free
free
free
5) store frame address in page table entry; make entry present
4) bring page into memory
physical memory

What If There Are No Free Frames?

- Find some frame in memory that is not really in use, and swap it out
  - need a **page replacement** algorithm
  - should result in minimum number of page faults
- Same page may be brought into memory several times

Basic Page Replacement

- Find the location of the desired page on disk
- Find a free frame
  - if there is a free frame, use it
  - if there are no free frames, use a page replacement algorithm to select a **victim** page
- Write victim page to disk if page has been modified
- Update victim page's page table entry to indicate that page is in swap space (can store location here as well)
- Bring the desired page into the newly freed frame
  - update the page tables
- Restart the user process

Reference String

- A string of page numbers accessed by a process during execution
- Useful in computing how many page faults will occur when a particular replacement algorithm is used

Optimal Page Replacement Algorithm

- Replace page that will not be used for longest period of time

- How do we know this?
  - we don't!! not really a practical algorithm
  - useful for measuring how well other algorithms perform

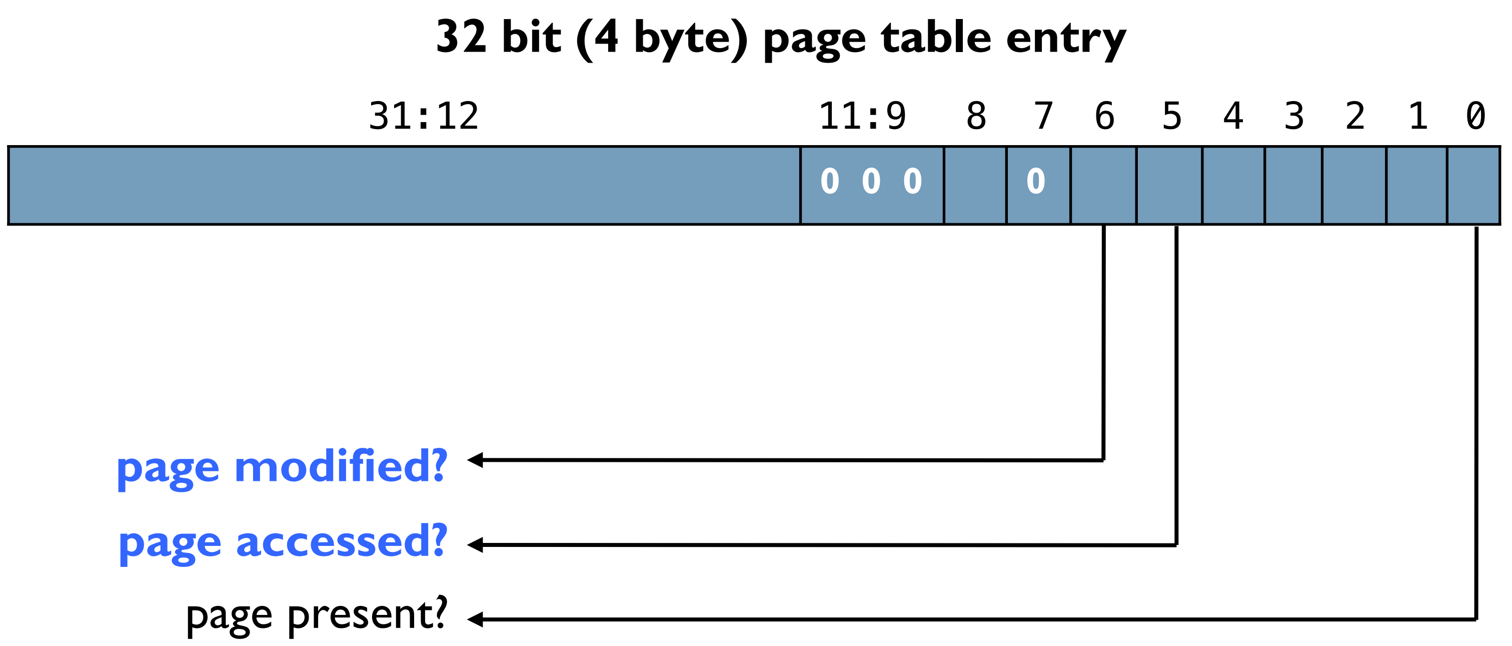
Optimal Algorithm Example

🟥 frame to be overwritten in case of a page fault

reference string

		7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
memory frames	🟥	7	7	🟥 7	2	2	2	2	2	2	2	2	2	2	2	🟥 2	🟥 2	🟥 2	🟥 7	7	7
		🟥	0	0	0	0	0	🟥 0	🟥 4	🟥 4	🟥 4	🟥 0	0	0	0	0	0	0	0	0	0
			🟥	1	🟥 1	🟥 1	🟥 3	3	3	3	3	3	🟥 3	🟥 3	🟥 1	1	1	1	1	1	1

Using Page Table Entry Flags



Modifed bit set by hardware when page is written to
Accessed (Referenced) bit set by hardware when page is read or written to

Once set, these bits are not cleared by hardware
OS can clear them (set to 0) during timer interrupts

Not Recently Used Algorithm

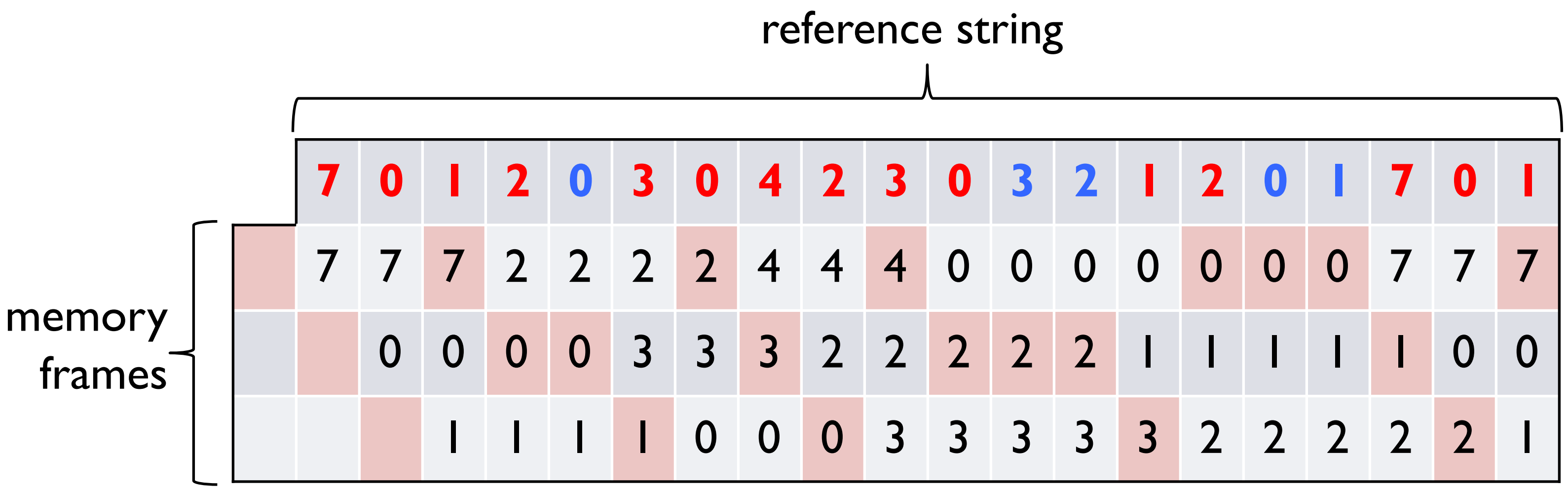
- OS periodically clears accessed (referenced) bit

- Class 0: Accessed = 0; Modified = 0
- Class 1: Accessed = 0; Modified = 1
- Class 2: Accessed = 1; Modified = 0
- Class 3: Accessed = 1; Modified = 1

- Choose Class 0 page (swap out not necessary)
- If none, choose class 1 page
- If none, choose class 2 page (swap out not necessary)
- If none, choose class 3 page

16 FIFO Algorithm

🟥 frame to be overwritten in case of a page fault


reference string
memory frames

		7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
memory frames	🟥	7	7	🟥 7	2	2	2	🟥 2	4	4	🟥 4	0	0	0	0	🟥 0	🟥 0	🟥 0	7	7	🟥 7
		🟥	0	0	🟥 0	🟥 0	3	3	🟥 3	2	2	🟥 2	🟥 2	🟥 2	1	1	1	1	🟥 1	0	0
			🟥	1	1	1	🟥 1	0	0	🟥 0	3	3	3	3	🟥 3	2	2	2	2	🟥 2	1

Is More Frames Better?

- 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)

	1	2	3	4	1	2	5	1	2	3	4	5
	1	1	1	4	4	4	5	5	5	5	5	5
		2	2	2	1	1	1	1	1	3	3	3
			3	3	3	2	2	2	2	2	4	4

9 page faults

- 4 frames

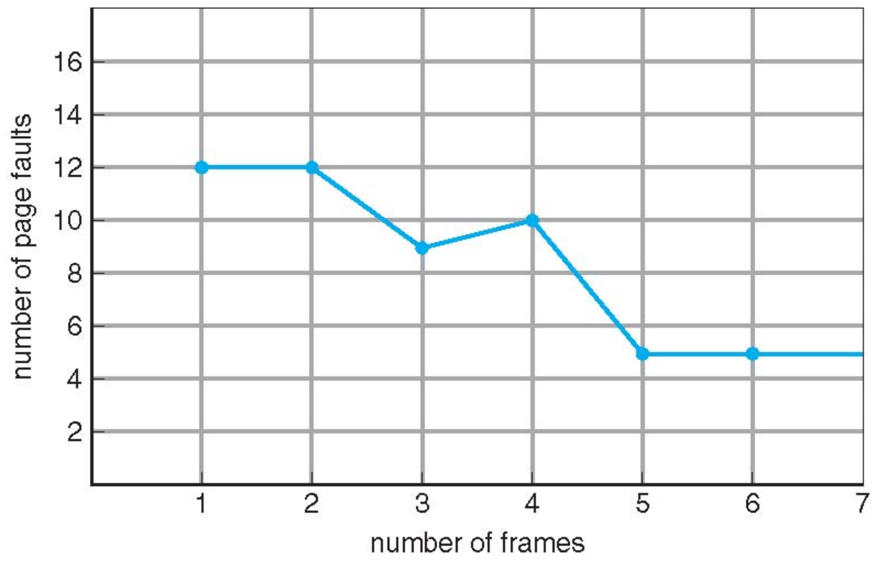
	1	2	3	4	1	2	5	1	2	3	4	5
	1	1	1	1	1	1	5	5	5	5	4	4
		2	2	2	2	2	2	1	1	1	1	5
			3	3	3	3	3	3	2	2	2	2
				4	4	4	4	4	4	3	3	3

10 page faults

Belady's Anomaly

- For some page replacement algorithms, increasing the number of frames increases the number of page faults
  - increasing memory does not always increase performance!!

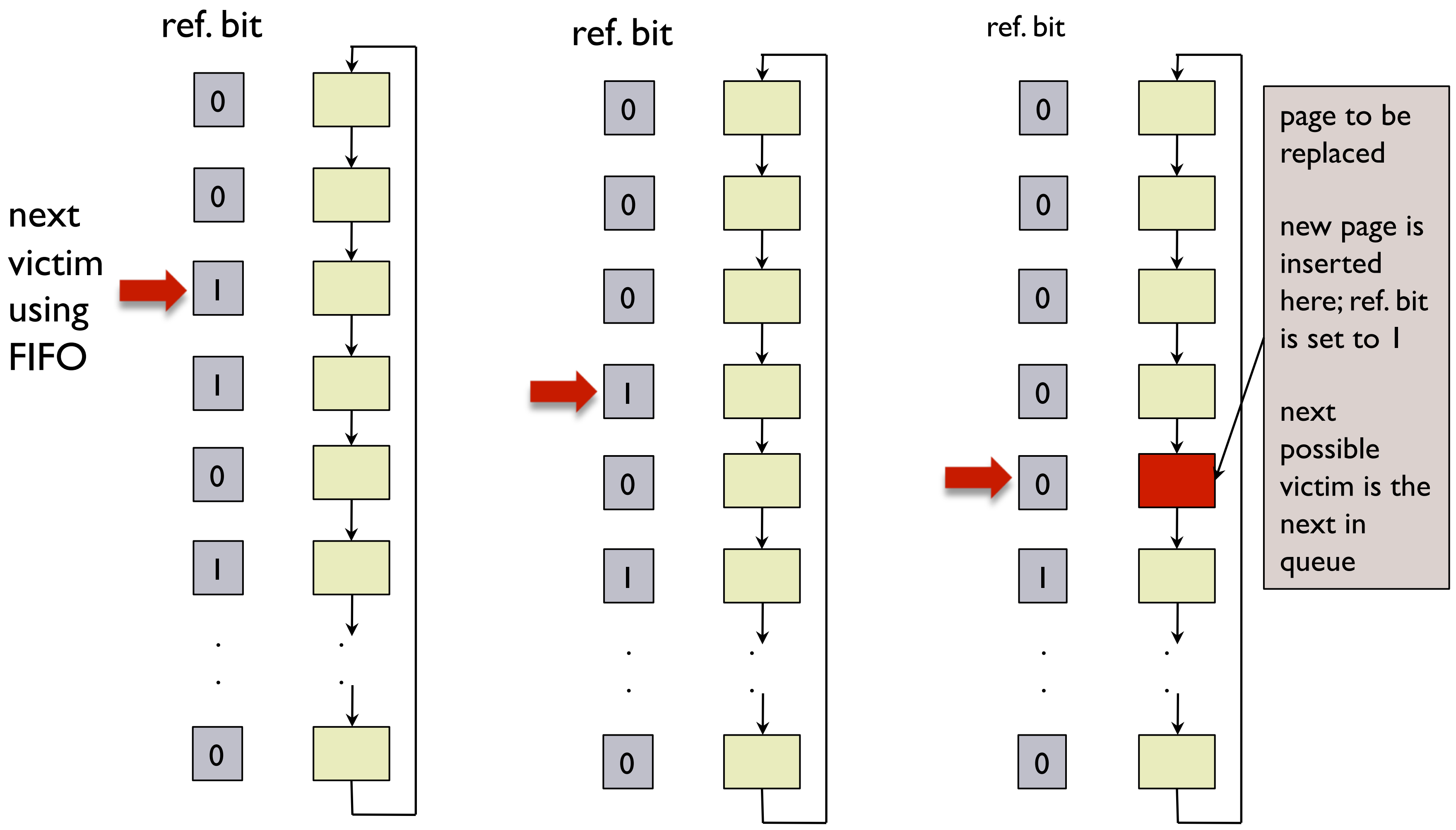
reference string:
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



Second-Chance Algorithm

- Second chance
  - same as FIFO but uses reference bit
- After a page has been selected using FIFO
  - if Accessed = 1
    - set reference bit to zero
    - leave the page
    - move over to next possibility according to FIFO

Clock Algorithm



ref. bit
next victim using FIFO
0
0
1
1
0
1
0
ref. bit
0
0
0
1
0
1
0
ref. bit
0
0
0
0
0
1
0
page to be replaced
new page is inserted here; ref. bit is set to 1
next possible victim is the next in queue

Second-Chance Using Circular Queue

LRU Page Replacement Algorithm

- Least Recently Used (LRU) page is chosen as the victim
- Counter implementation (not really possible)
  - every page table entry has a counter field
  - every time page is referenced through this entry, increment counter
  - when a page needs to be replaced, choose the one with the smallest counter value
  - requires a search to find the least recently used page

LRU Example

🟥 frame to be overwritten in case of a page fault

	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
🟥	7	7	7	2	2	2 🟥	2 🟥	4	4	4 🟥	0	0	0 🟥	1	1	1 🟥	1	1	1 🟥	1
	🟥	0	0	0 🟥	0	0	0	0	0 🟥	3	3	3	3	3 🟥	3 🟥	0	0	0 🟥	0	0
		🟥	1	1	1 🟥	3	3	3 🟥	2	2	2 🟥	2 🟥	2	2	2	2	2 🟥	7	7	7 🟥

LRU Approximation Algorithms

- Most systems do not provide any hardware support for a true LRU implementation
- **Not Frequently Used** algorithm: use the reference bit (accessed bit) provided in hardware
  - maintain software counter for each page, initially = 0
  - at each timer interrupt, add the page's reference bit value to counter
    - then clear reference bit
  - on page fault, choose page with lowest counter value
  - what happens if page is heavily used and then not used at all?

Aging Algorithm

- Maintain a byte (8 bits) for each page
- At timer interrupt, for each page
  - discard low-order bit of the byte
  - insert reference bit for the page at the left-most position

$$11001001 \xrightarrow{\text{ref. bit} = 0} 01100100$$

  - then, clear the reference bit
- Page with lowest value is the least referenced one
- Issues
  - do not know which page was accessed more recently between two timer interrupts
  - do not know which page was accessed nine ticks ago

Demand Paging

- Process starts with no pages in memory
- Pages are brought into memory as and when they are needed

- Why demand paging works?
  - programs tend to have *locality of reference*
  - programs heavily refer to a set of pages before moving on to another set

Working-Set Model

- Working Set: $w(k,t)$ = the pages used by a process in the last k memory references at time t
  - easier to work with "pages used by a process in the last Δ time units"
- If entire working set is not in memory, process will cause too many page faults (called **thrashing**)
- Reduce page faults by *prepaging* the working set

Working Set Based Page Replacement

- Approximate with interval timer and a reference bit
  - need to maintain timer tick count of last use for each page in memory
- Example: Δ = 10,000 timer ticks
  - reference bit is cleared on every timer tick (interrupt)
    - the bit is set by hardware every time the page is used
  - on page fault, examine reference bit and timer tick count to find page that has not been used in last Δ ticks

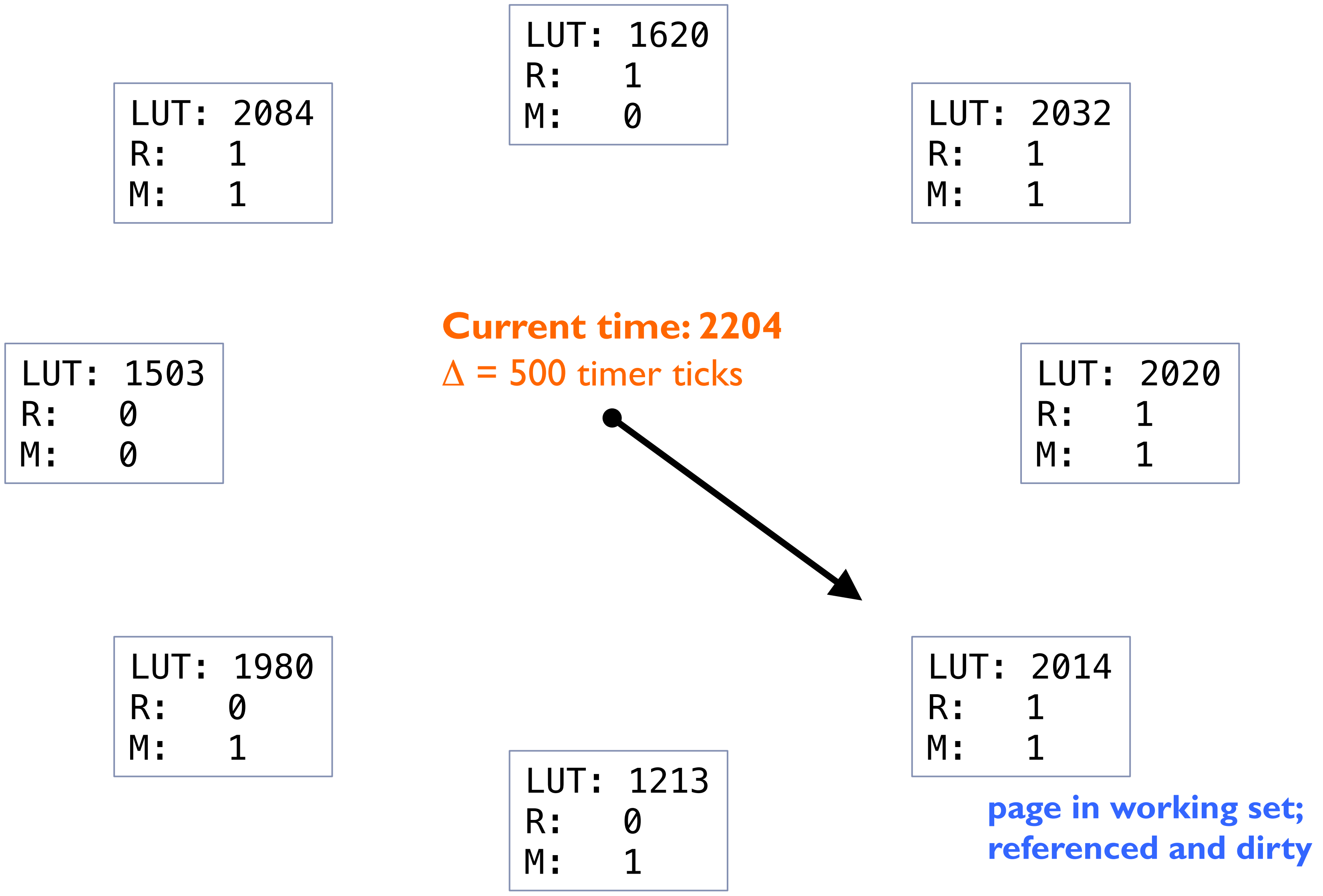
Using Working-Set on a Page Fault

LUT: timer tick when last used
R: reference bit
M: modified bit

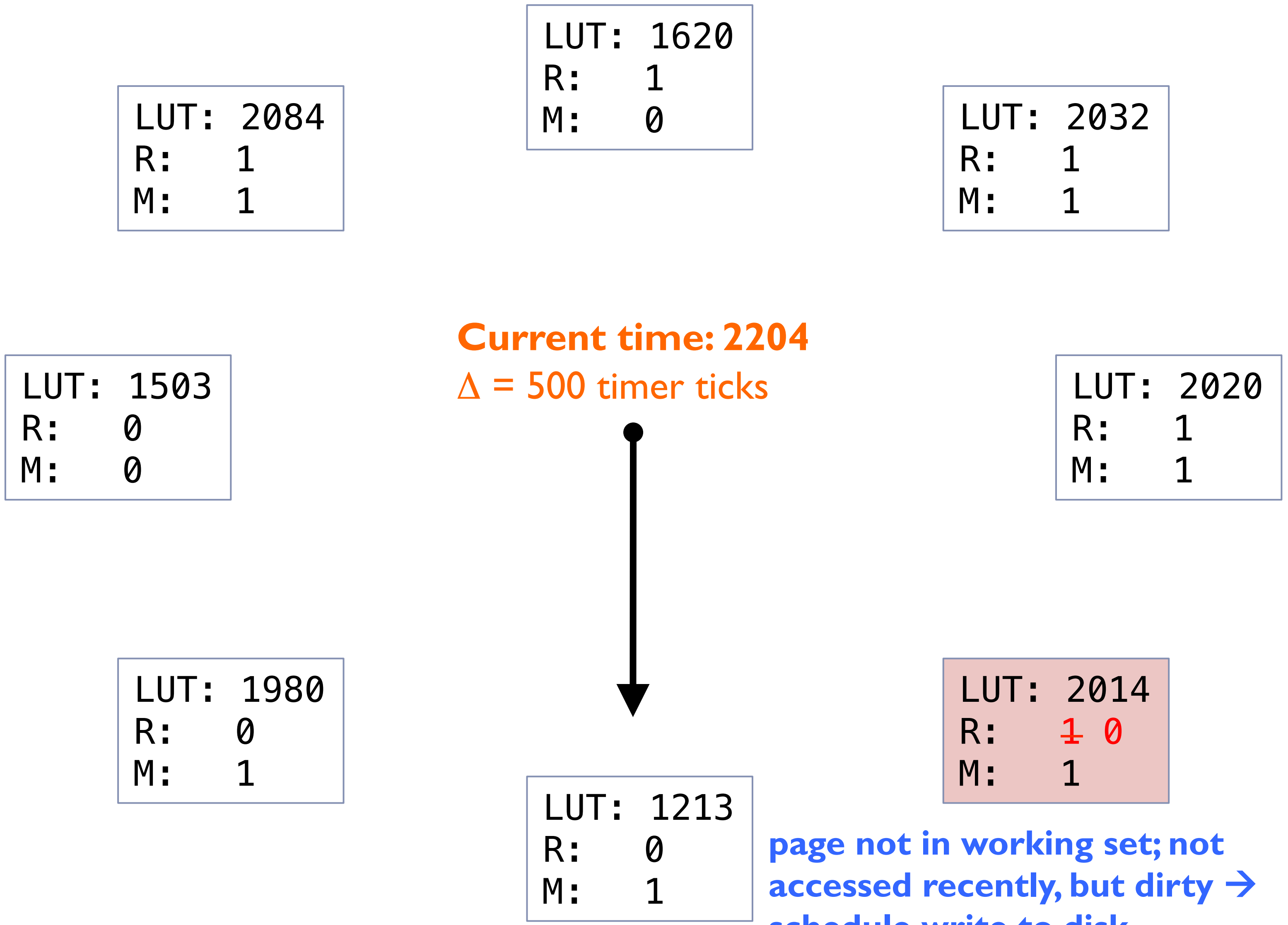
Current timer value = 103450
Δ = 10,000 timer ticks

LUT: 102084 R: 1	LUT: 93000 R: 0	LUT: 94003 R: 0	LUT: 92000 R: 1
a page used between tick 103450 and now; before that page was used near tick 102084	a page not used since tick 93000	a page not used since tick 94003	a page used between tick 103450 and now; before that page was used near tick 92000
age = 0	**age = 10450**	**age = 9447**	**age = 0**
LUT: 103450 R: 1	LUT: 93000 R: 0	LUT: 94003 R: 0	LUT: 103450 R: 1
in working set	***not in working set***	***in working set***	***in working set***

WSClock Algorithm



WSClock Algorithm

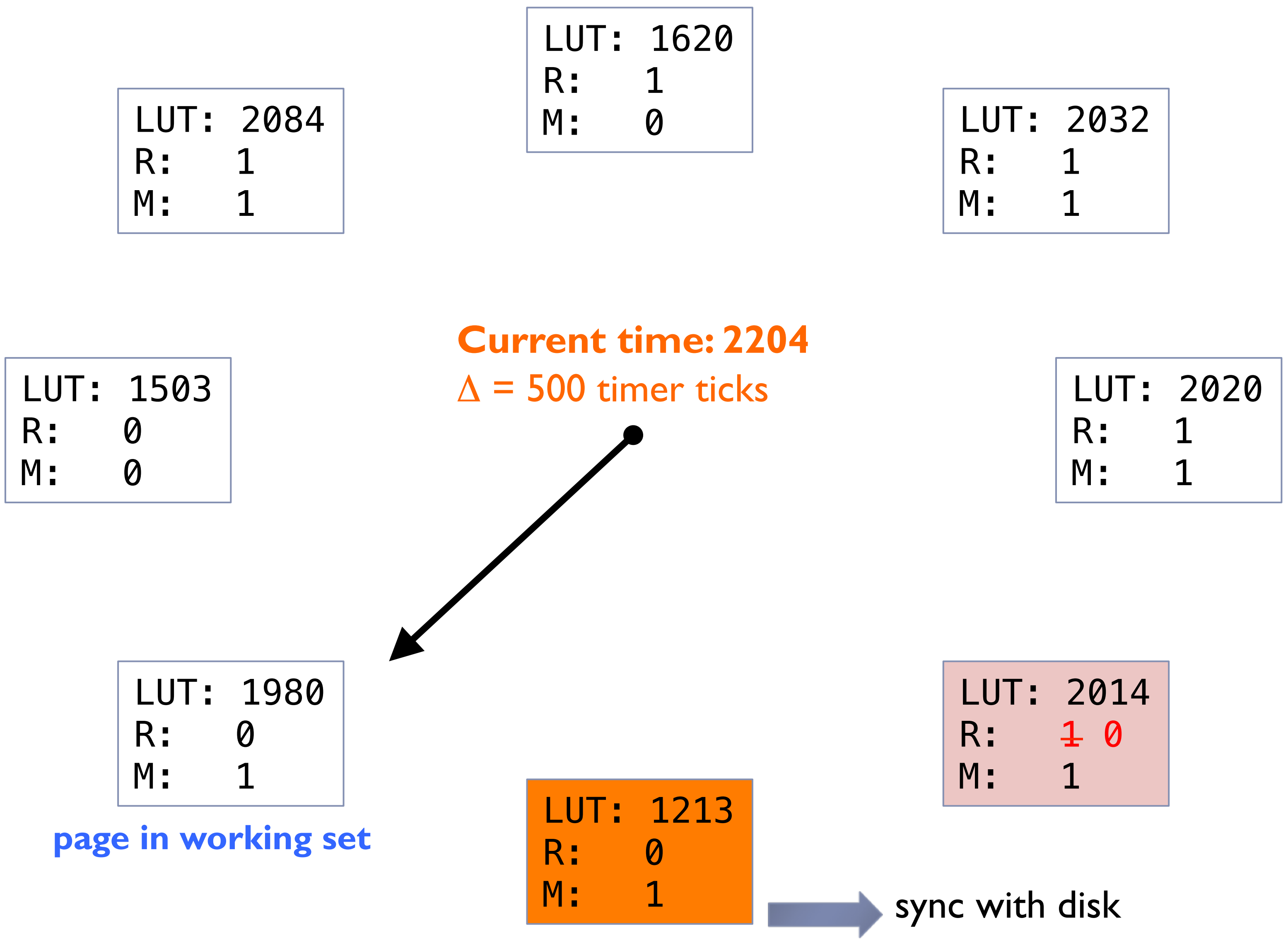


Current time: 2204

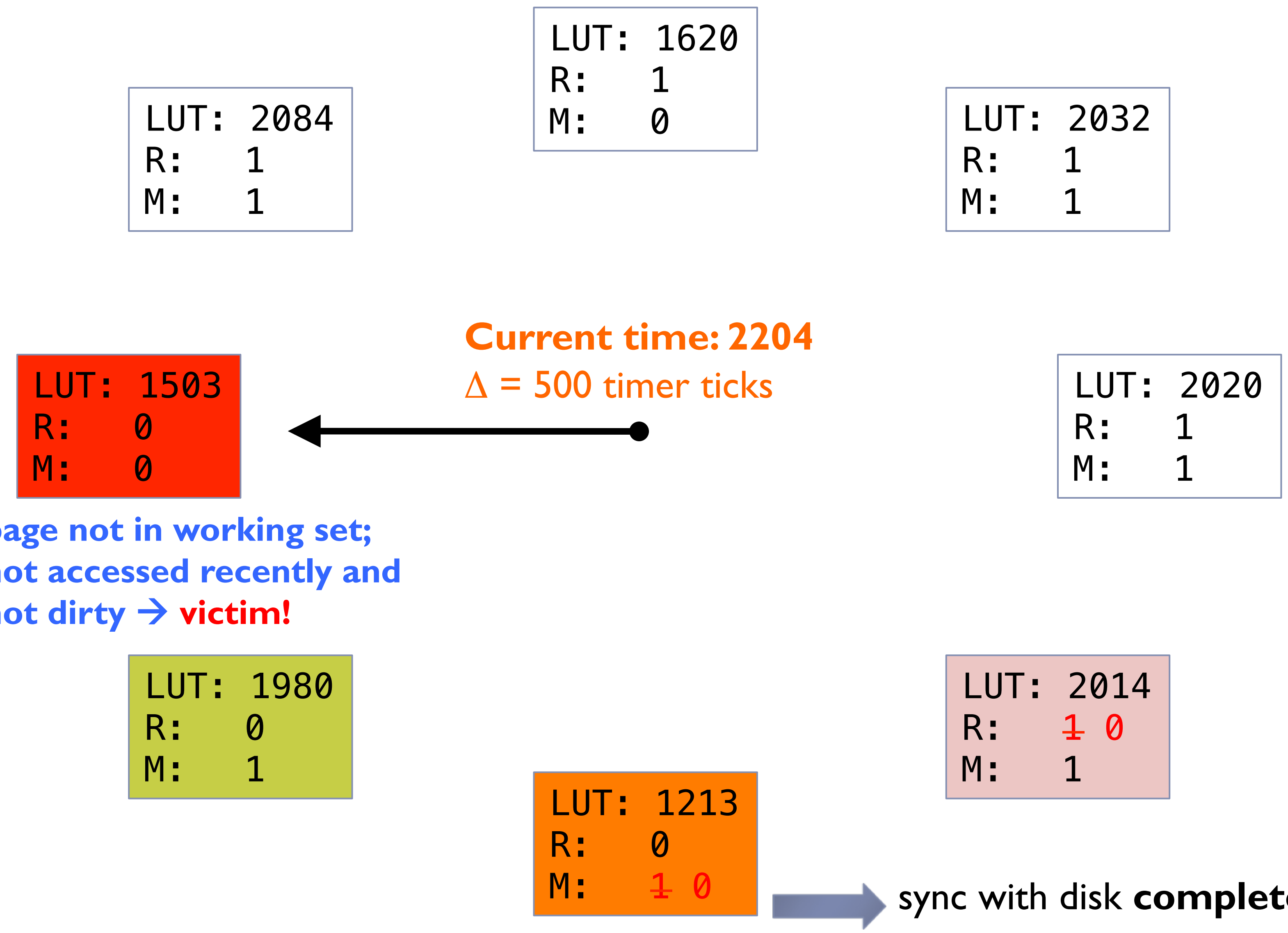
Δ = 500 timer ticks

page not in working set; not accessed recently, but dirty → schedule write to disk

WSClock Algorithm



WSClock Algorithm



Current time: 2204

Δ = 500 timer ticks

page not in working set; not accessed recently and not dirty → victim!

sync with disk **complete!**

Global Vs. Local Replacement

- **Global replacement**: select a replacement frame from the set of all frames
  - one process can take a frame from another
  - process cannot control its page-fault rate
- **Local replacement**: each process selects from only its own set of allocated frames
  - does not make less used frames available to other processes

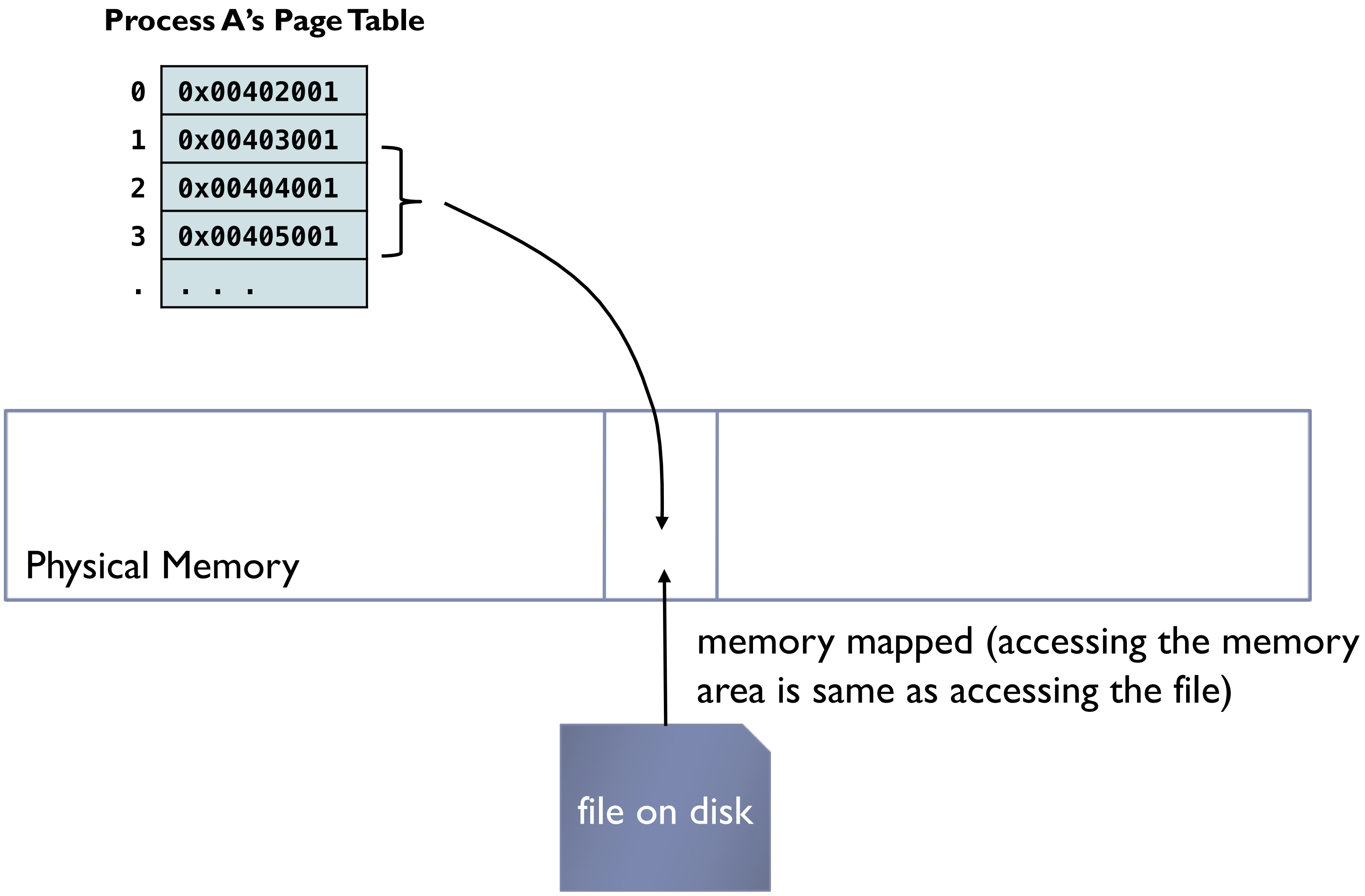
Page Size

- Page size selection must take into consideration
  - fragmentation
    - smaller pages ⇒ less internal fragmentation
  - locality
    - smaller pages ⇒ better *resolution* in locality
  - table size
    - larger pages ⇒ smaller page table
  - I/O overhead
    - larger pages ⇒ less I/O time
  - TLB
    - larger pages ⇒ faster translation of more of the address space

Shared Pages and Shared Libraries

- **Shared pages**: pages in different processes mapped to same frames (see slide L9-2)
  - must update page tables of all processes when swapping out one process
    - too many page faults when other processes try to access shared memory
  - must be able to discover that frames are shared when one process finishes
- **Shared library**: commonly used code shared across processes (e.g. libc)
  - implemented using shared pages
  - call to a shared library function is replaced with small stub that calls function implemented in the shared pages

Memory Mapped Files



Page Pinning

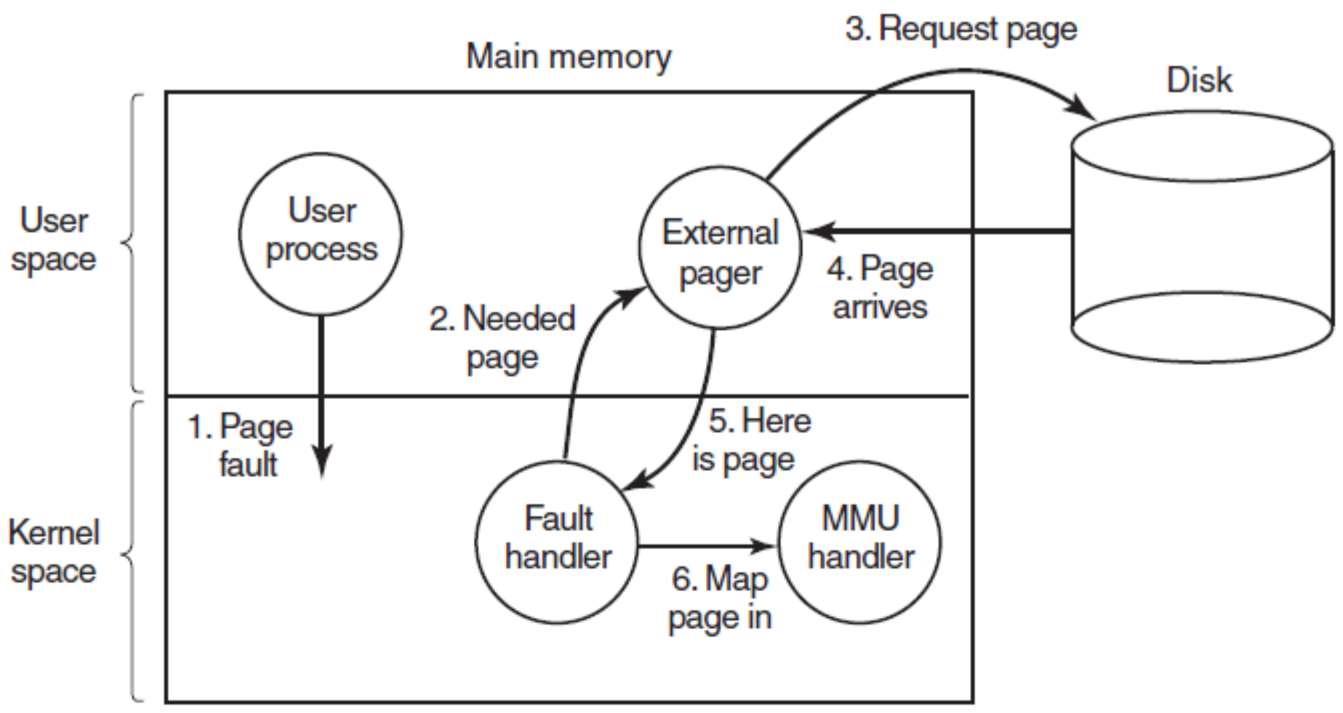
- Pages must sometimes be locked into memory
  - locked means it cannot be chosen as a victim for page replacement

- E.g. pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm

Implementing Swap Space

- Keep aside an area of disk for swap space
  - should be large enough to hold most (better if all) processes
- Divide swap space into "swap regions" of size same as a page
  - one sector = 512 bytes; 8 sectors = 4 KB (page size)
- Maintain an array in kernel to store sector address of beginning of swap regions, and which ones are not in use
- When a page is swapped out (written to some swap region), store array index in page table entry

Overall Page Fault Handling



References

- Chapter 3.4-3.6, Modern Operating Systems, A. Tanenbaum and H. Bos, 4th Edition.