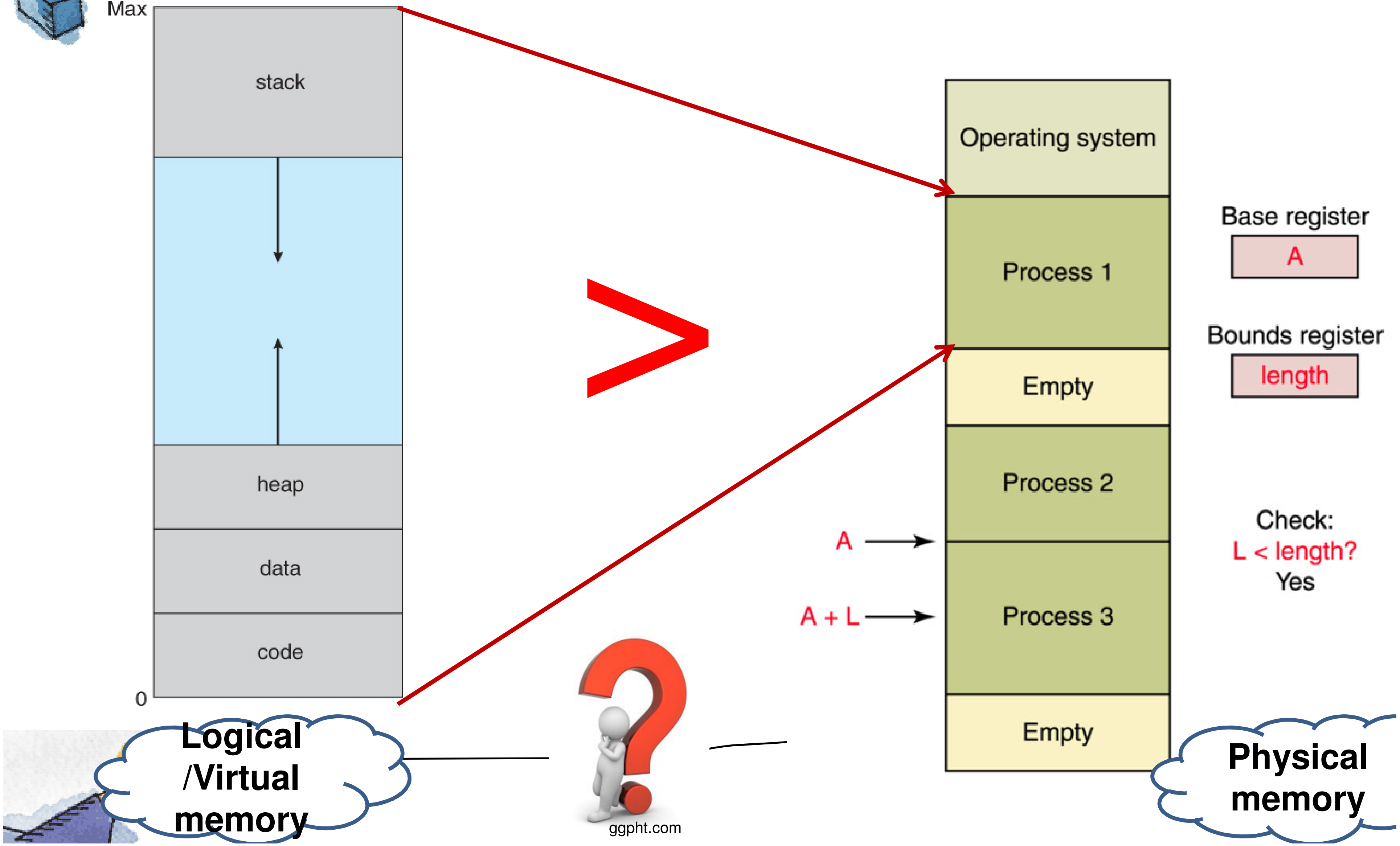
Operating Systems

Virtual Memory

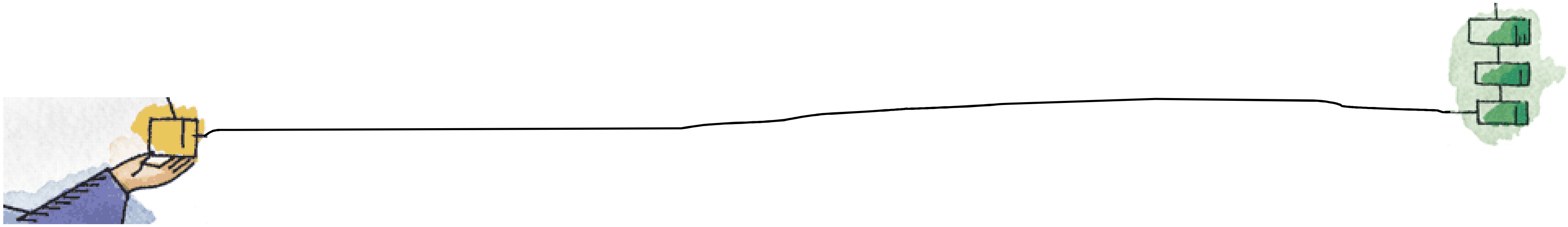
Dr. Shamim Akhter

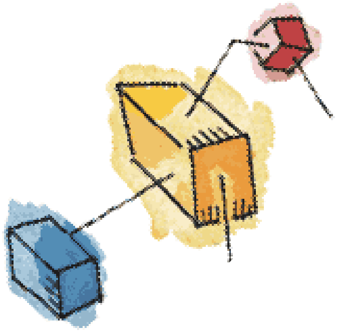
Programmers are concerned about over physical memory storage limitation.



ggpht.com

- Process Execution Phase:
  - – entire program may not be needed at the same time
  - – active parts are in memory & rest are in hard disk

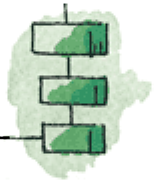




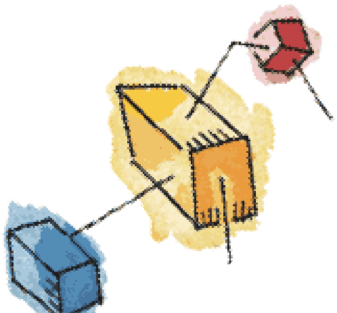
Virtual Memory

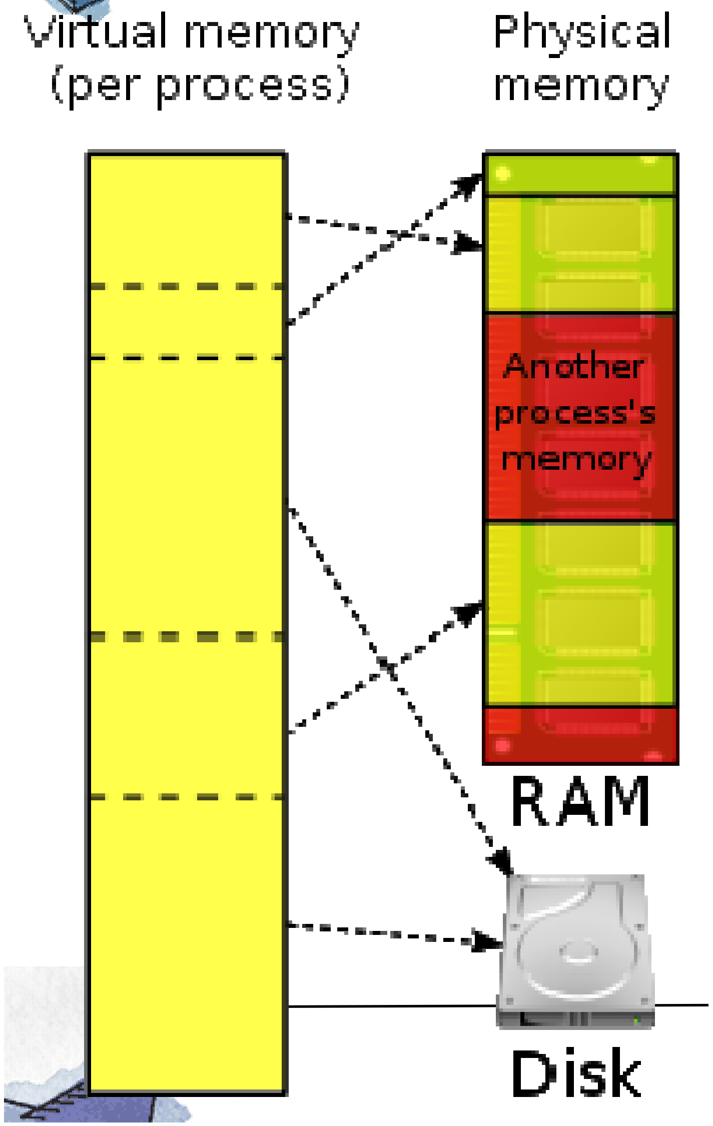
A technique that allows the execution of processes that may not completely in memory

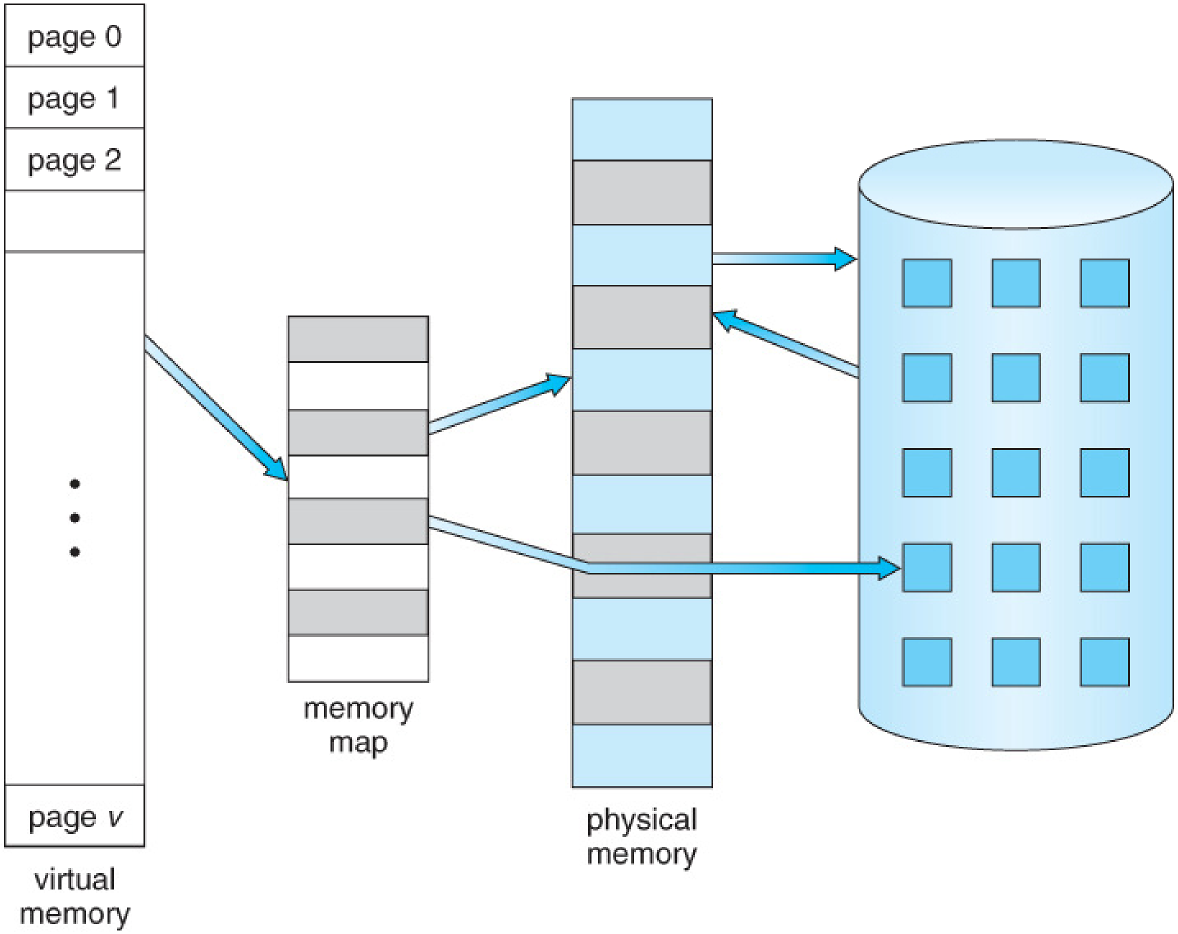
Commonly implemented by demand paging

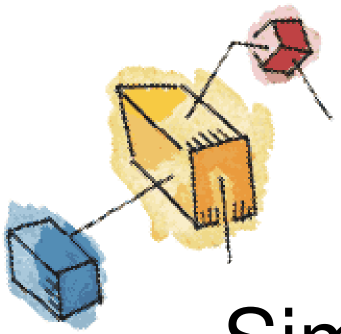


Virtual Memory









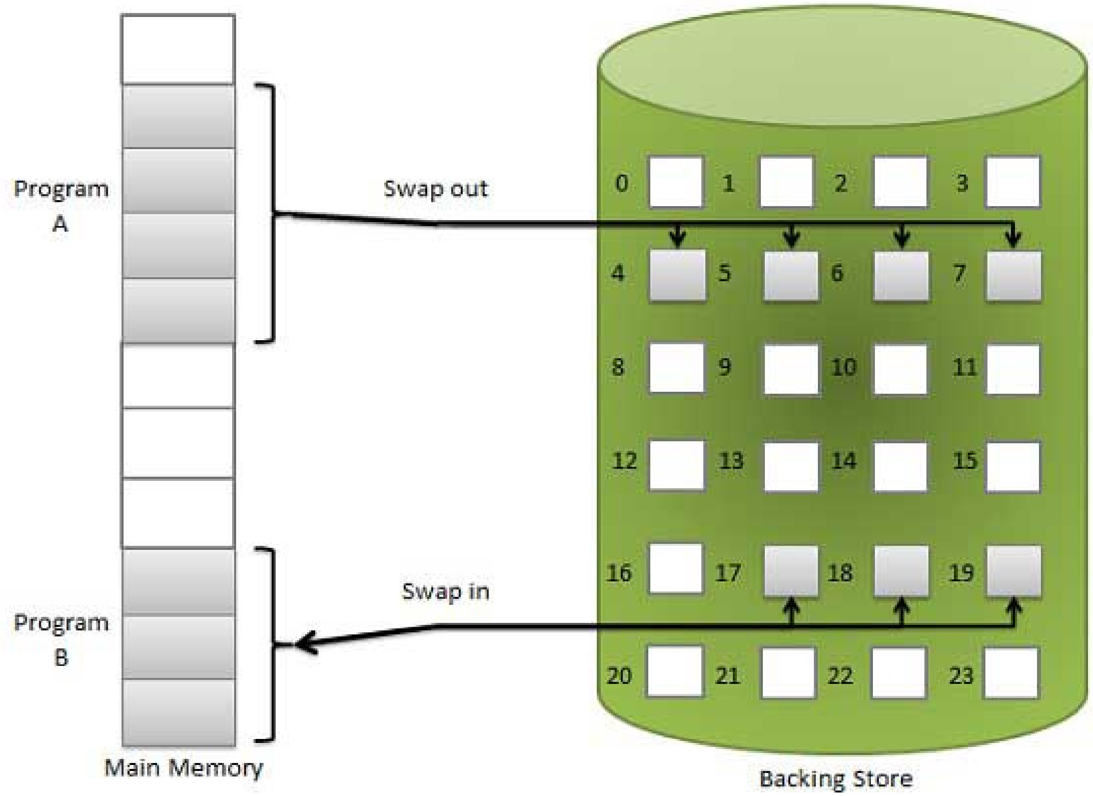
Demand Paging

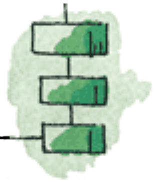
Similar to paging system with swapping

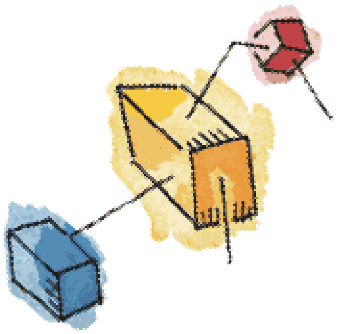
Lazy Swapper (swap pages)

Pager

Bring only necessary pages into memory

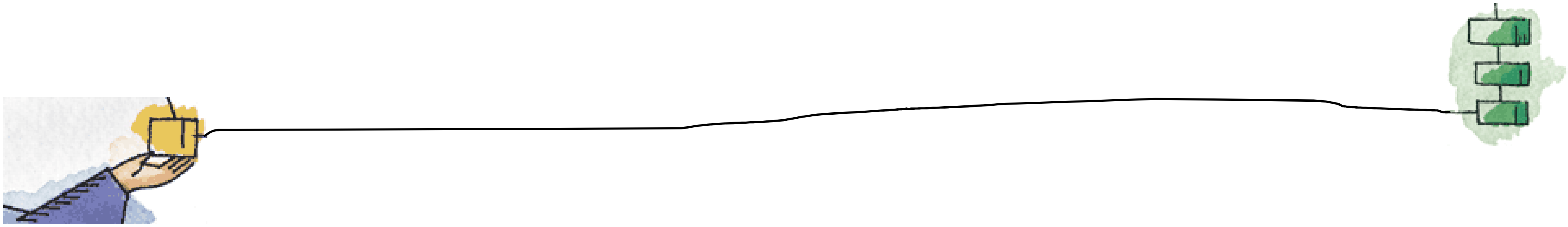


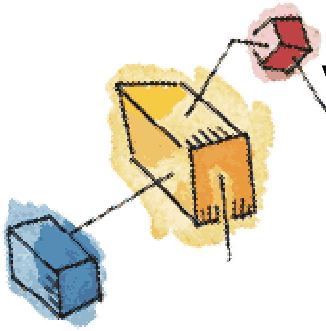




Demand Paging

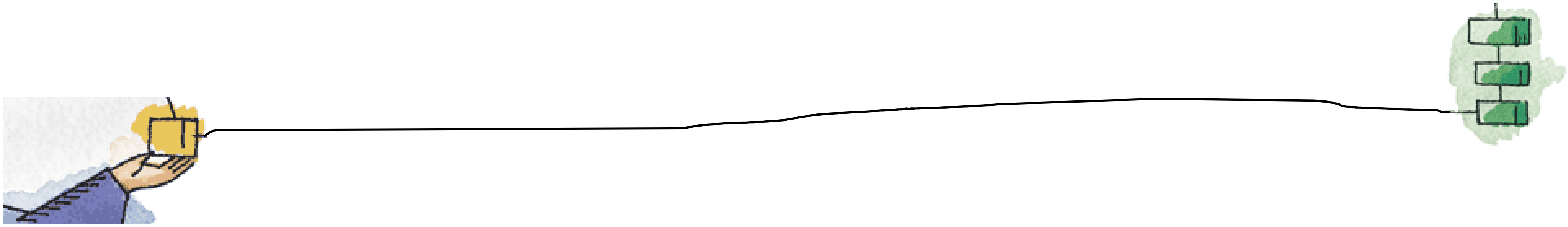
- Pager needs hardware supports to distinguish
  - pages are in memory and pages are on disk
- Sol^{n}: valid-invalid bit scheme



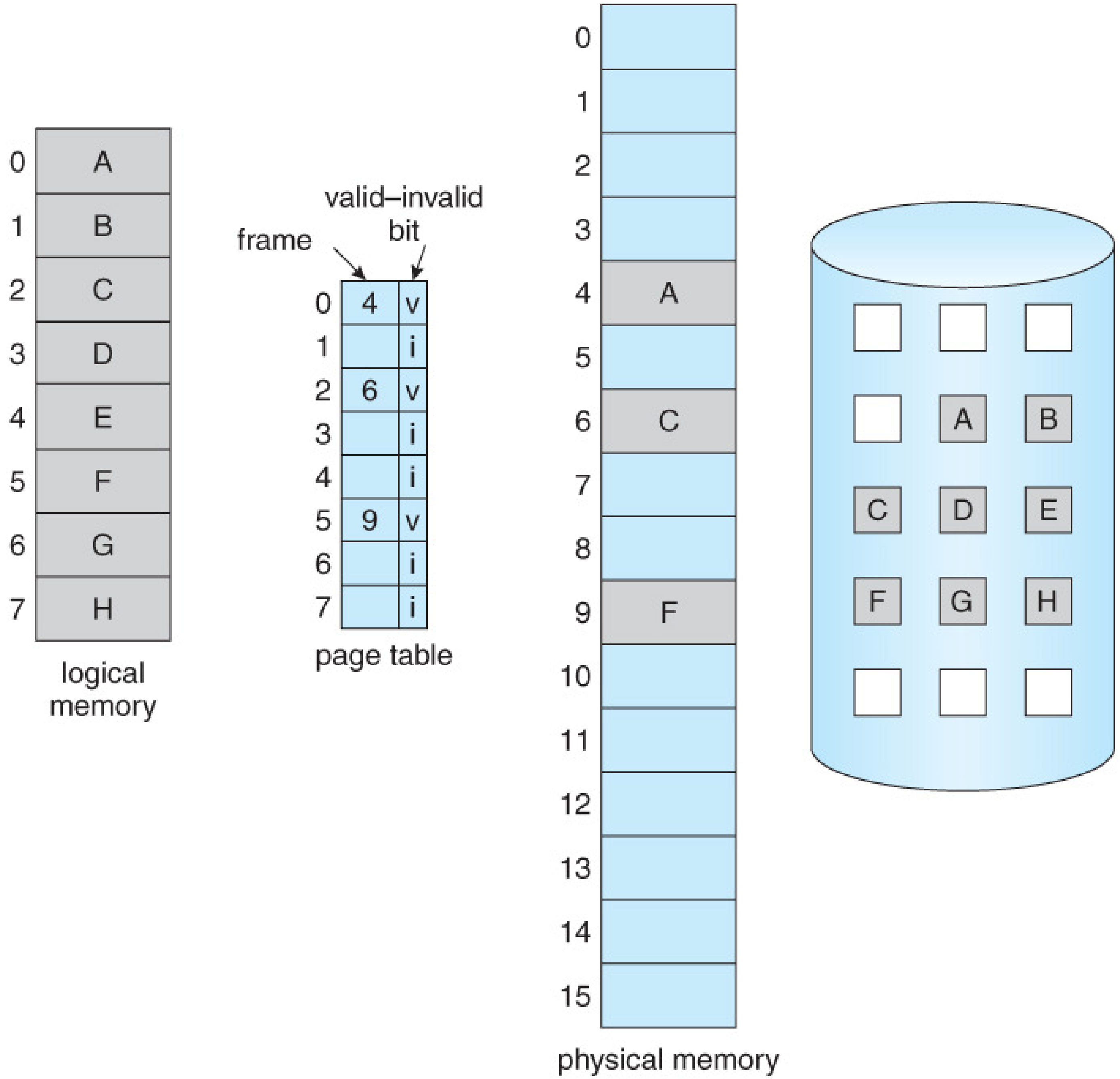


Valid-Invalid Bit Scheme

- Validity bit is used and resides with page table
- Set 1: page is valid and resides in memory
- Set 0: page is invalid-not in process LA space
  or is valid but currently on disk



Valid-Invalid Bit Scheme



	frame	valid–invalid bit
0	4	v
1		i
2	6	v
3		i
4		i
5	9	v
6		i
7		i

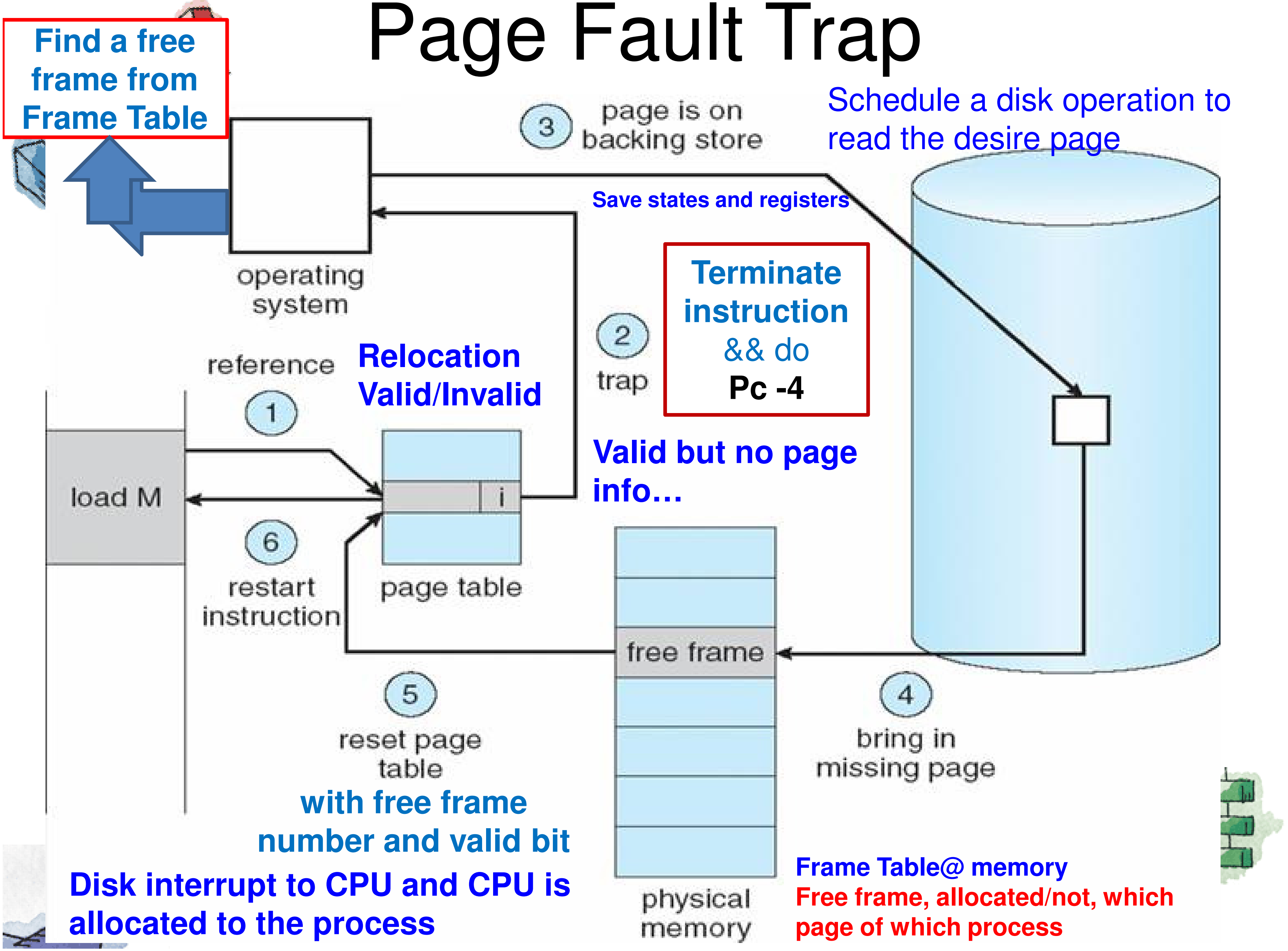
page table

- What happens if the process tried to use a page that was not in memory?

Page fault

Threads in User Space: a page fault would block the entire process (i.e., all the threads).
Threads in the Kernel: a page fault, or infinite loop in one thread does not automatically block the other threads in the process.
Hybrid Implementations: page fault blocks only one kernel thread, the multi-threaded application as a whole can still run since user-level threads in other kernel-level threads of this process are still runnable.

Page Fault Trap



Find a free frame from Frame Table

Schedule a disk operation to read the desire page

Save states and registers

Terminate instruction && do **Pc -4**

Relocation Valid/Invalid

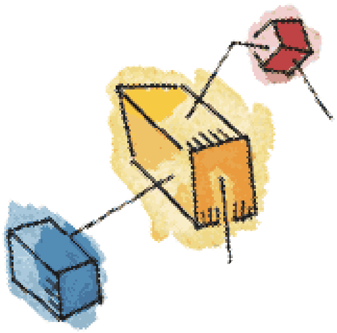
Valid but no page info…

with free frame number and valid bit

Disk interrupt to CPU and CPU is allocated to the process

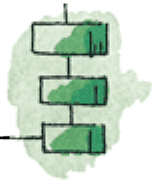
Frame Table@ memory

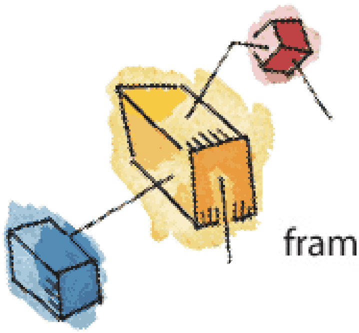
Free frame, allocated/not, which page of which process



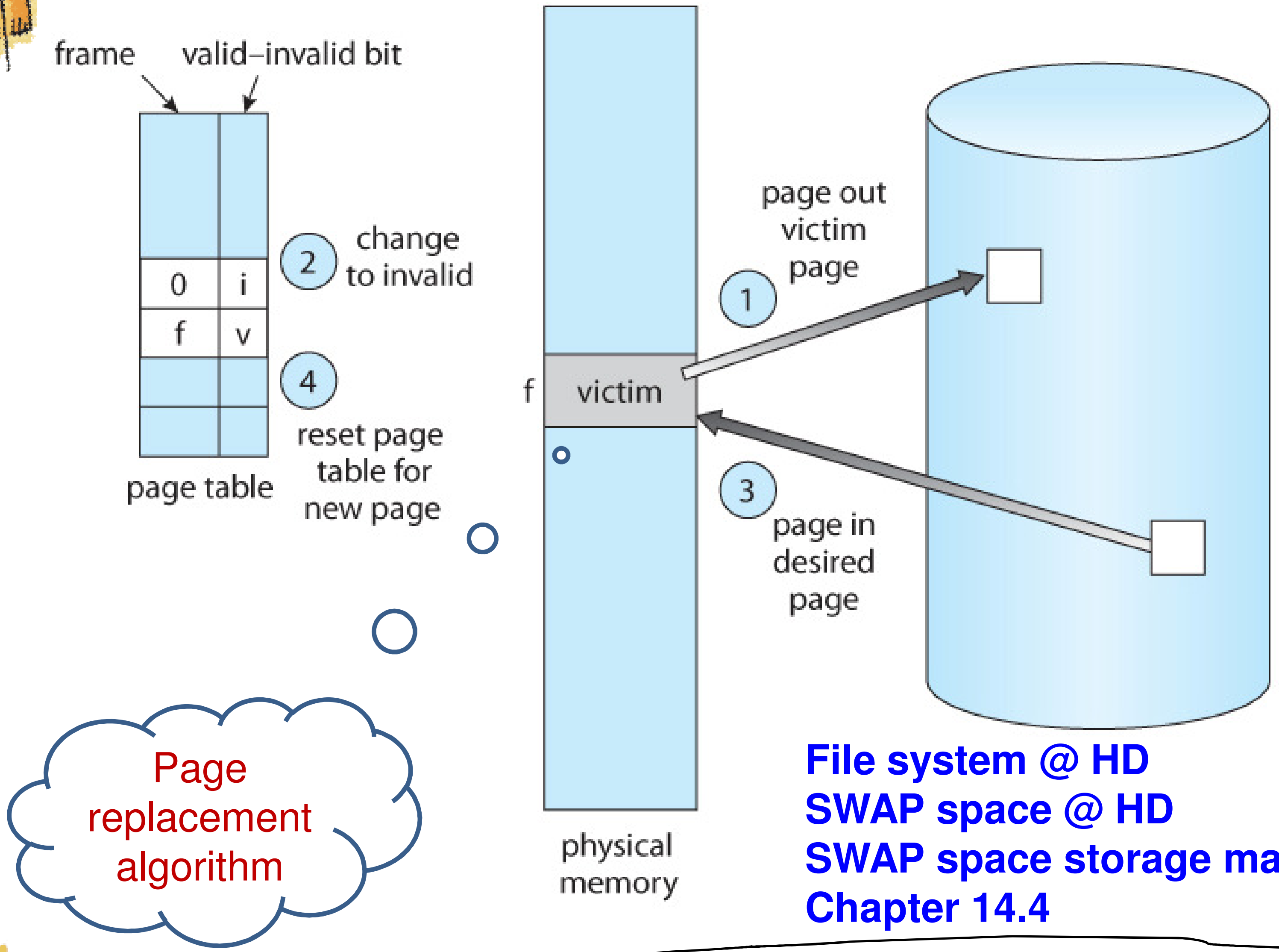
Page Replacement

- Desire page is in disk but no free frame in memory
- What will OS do?
  - Option 1: terminate the process
  - Option 2: swap out a process, freeing all its frames and reducing the level of multiprogramming.
  - Option 3: Page replacement

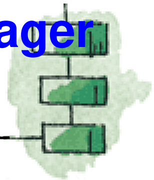




Page Replacement



File system @ HD
SWAP space @ HD
SWAP space storage manager
Chapter 14.4



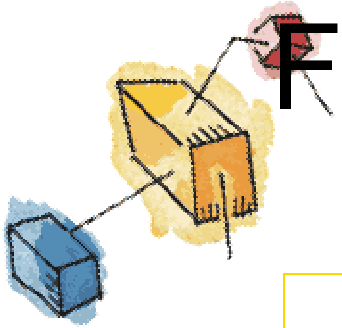
Double time than free frame method

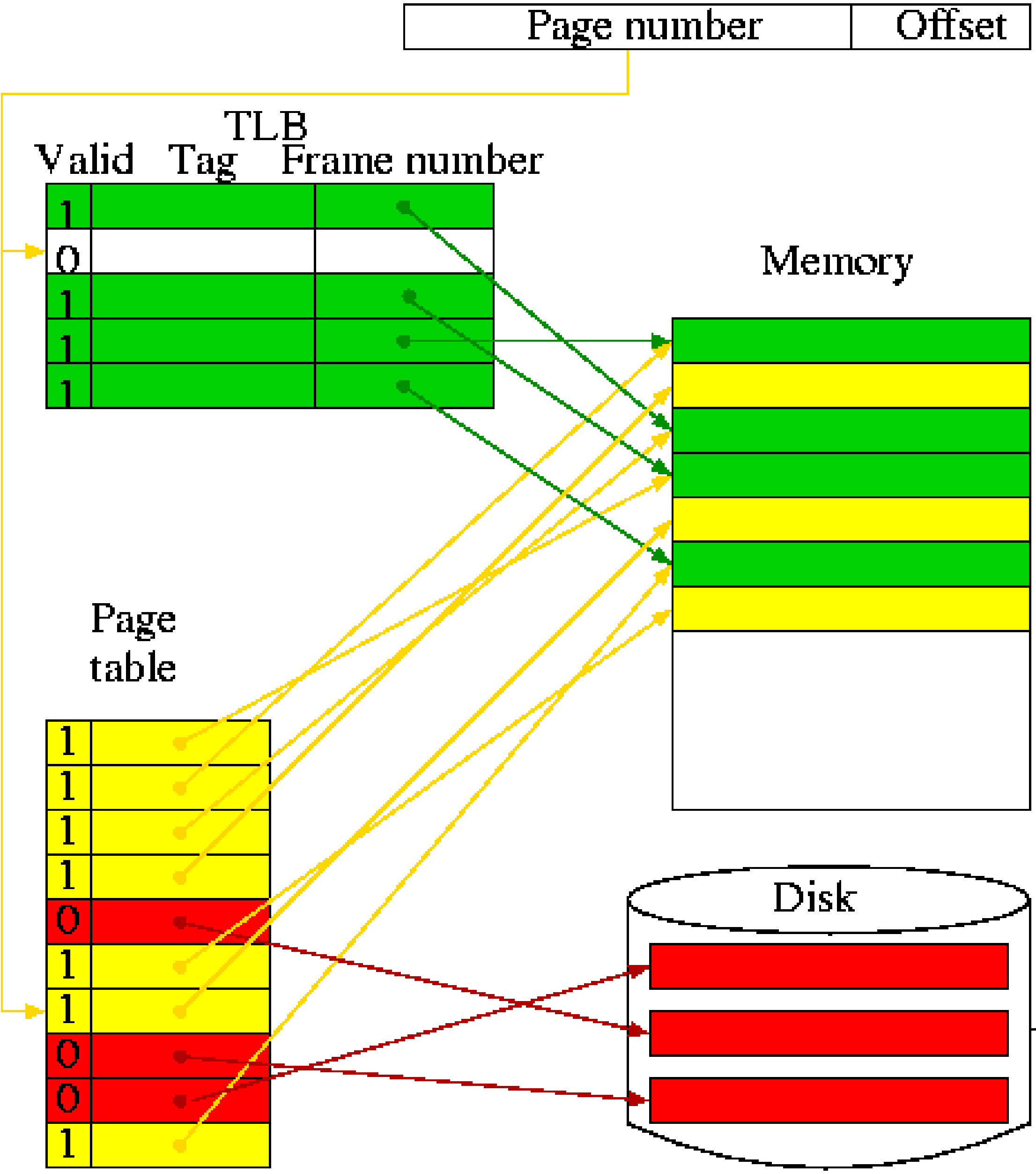
Solution: modify bit (dirty bit)

Each page has a modify bit.

```
If  modify bit==1
            Update the memory
Else
            No need to copy the victim page
```

Fast Translation using TLB





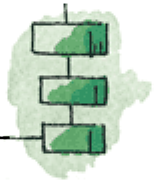
TLB

Valid	Tag	Frame number
1		
0		
1		
1		
1		

Page table

1	
1	
1	
1	
0	
1	
1	
0	
0	
1	

Translation Lookaside Buffer (TLB)

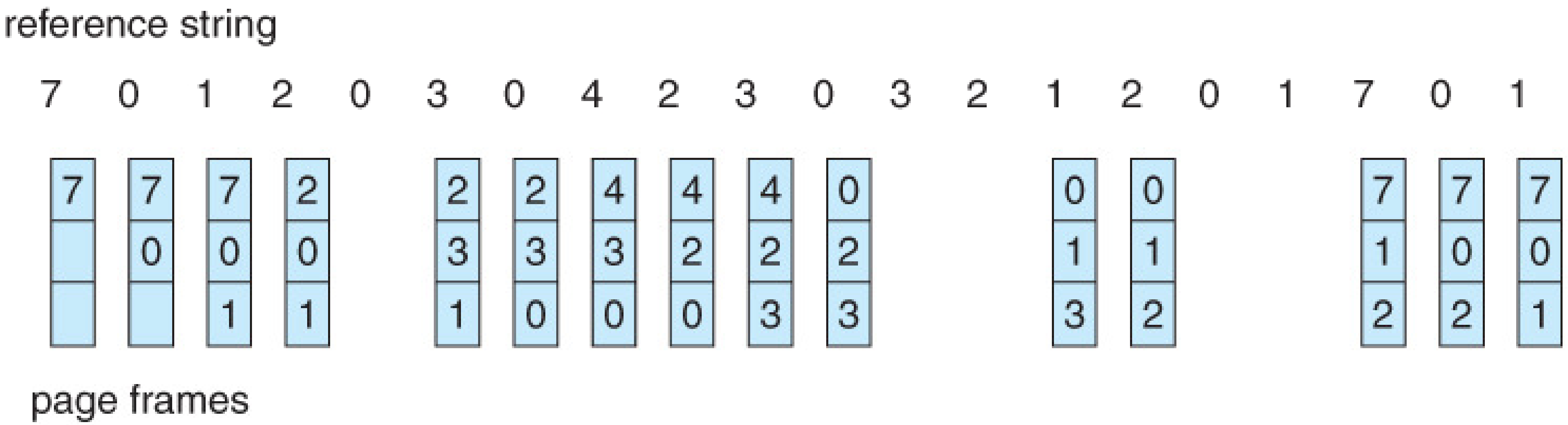


Page Replacement Algorithms

- Fixed Number of Frames
  - First In/ First Out (FIFO)
  - Optimal (OPT, MIN)
  - Least Recently Used (LRU)
  - Clock
  - Second-chance Cyclic
- Variable Number of Frames
  - Working Set (WS)
  - Page Fault Frequency (PFF)

FIFO Page Replacement

- As new pages are brought in, they are added to the tail of a queue, and the page at the head of the queue is the next victim.

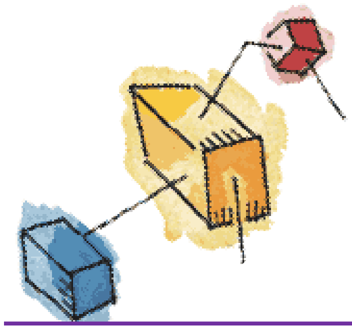


reference string

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

page frames

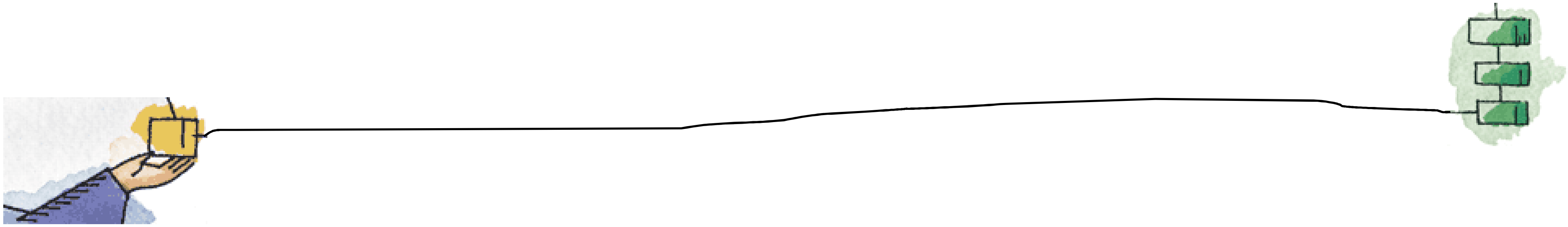
20 page requests result in 15 page faults.



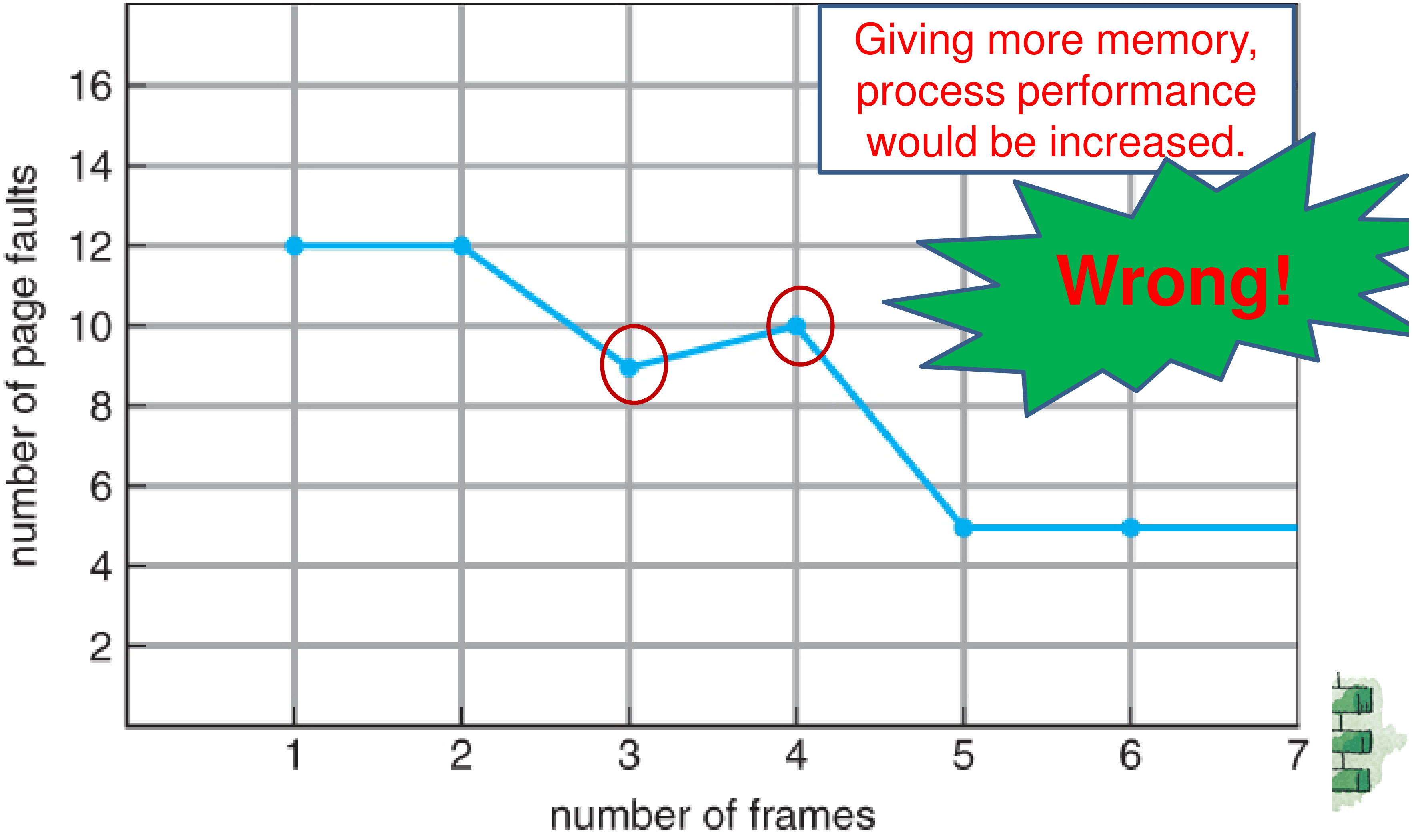
FIFO: Example 2

time	0	1	2	3	4	5	6	7	8	9	10
ω		*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
frame 0	→*a*	→*a*	→*a*	→*a*	→*a*	*e*	*e*	*e*	*e*	→*e*	*d*
frame 1	*b*	*b*	*b*	*b*	*b*	→*b*	→*b*	*a*	*a*	*a*	→*a*
frame 2	*c*	*c*	*c*	*c*	*c*	*c*	*c*	→*c*	*b*	*b*	*b*
frame 3	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	→*d*	*c*	*c*
page fault						1		2	3	4	5
page(s) loaded						*e*		*a*	*b*	*c*	*d*
page(s) removed						*a*		*b*	*c*	*d*	*e*

We shall demonstrate these algorithms by running them on the reference string ω = *cadbebabcd* and assume that, initialy, pages *a*, *b*, *c*, and *d*occupy frames 0, 1, 2, and 3 respectively. When appropriate, the little arrow → indicates the location of the "pointer" that indicates where the search for the next victim will begin.

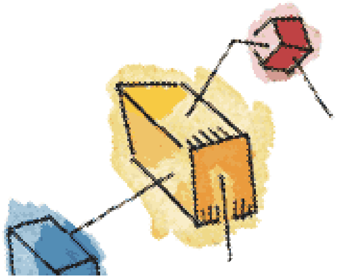


Belady’s Anomaly



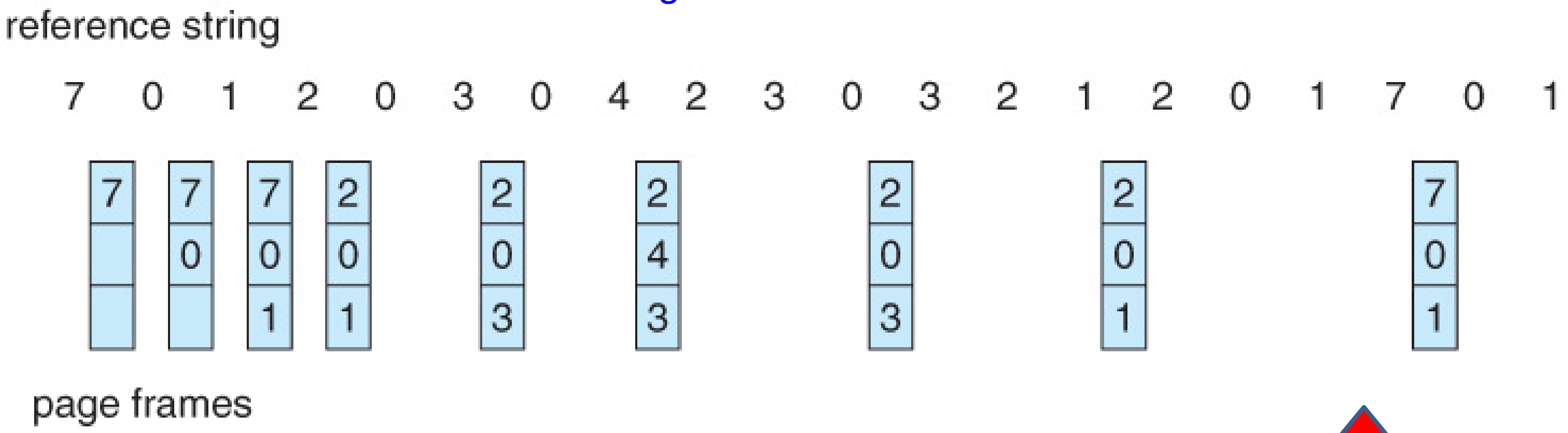
Giving more memory, process performance would be increased.

Wrong!



Optimal Page Replacement

"Replace the page that will not be used for the longest time in the future."



Yields the lowest of all possible page-faults

Does not suffer from Belady's anomaly

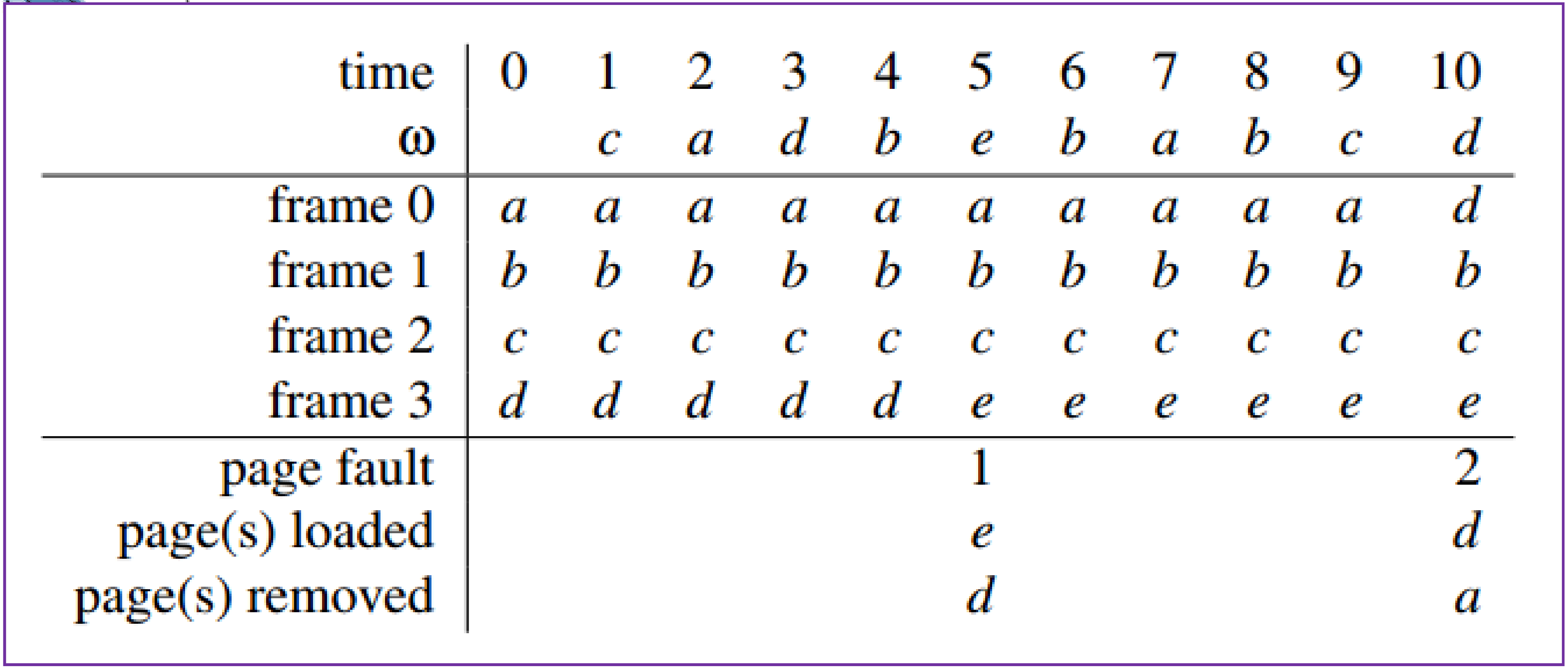
Requires future knowledge, difficult to implement

Mainly used for comparison study

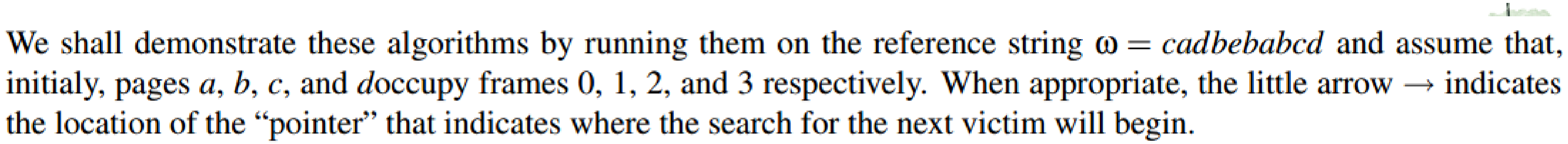
Lowest page fault

Best among all

Optimal Page Replacement



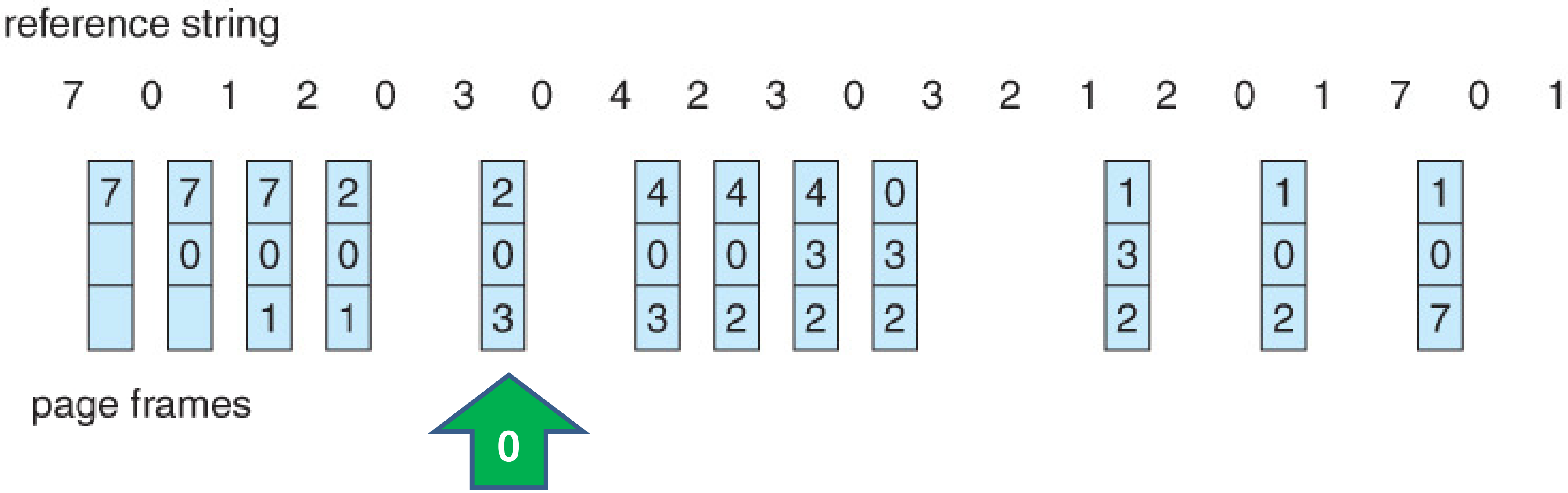
time	0	1	2	3	4	5	6	7	8	9	10
ω		*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
frame 0	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*d*
frame 1	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*
frame 2	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*
frame 3	*d*	*d*	*d*	*d*	*d*	*e*	*e*	*e*	*e*	*e*	*e*
page fault						1					2
page(s) loaded						*e*					*d*
page(s) removed						*d*					*a*



We shall demonstrate these algorithms by running them on the reference string ω = *cadbebabcd* and assume that, initialy, pages *a*, *b*, *c*, and *d*occupy frames 0, 1, 2, and 3 respectively. When appropriate, the little arrow → indicates the location of the "pointer" that indicates where the search for the next victim will begin.

LRU Page Replacement

Victim page that has not been used in the longest time

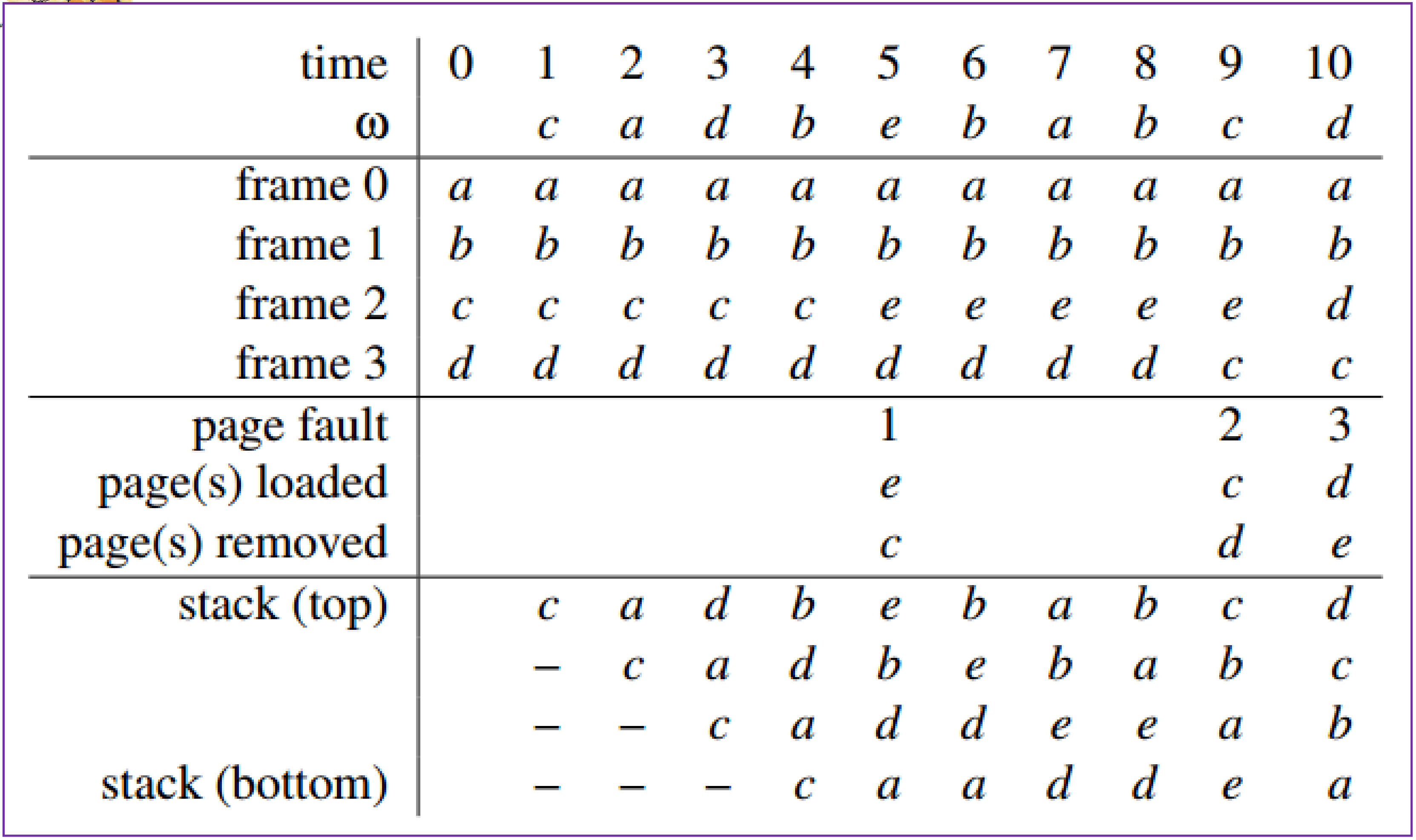


FIFO looks at the oldest load time, LRU looks at the oldest use time.

Analogous with OPT: OPT looks from future , LRU looks from past.

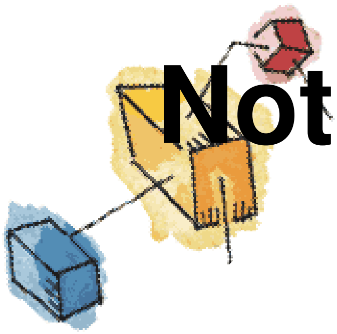
Yielding 12 page faults, (as compared to 15 for FIFO and 9 for OPT.)

LRU Page Replacement

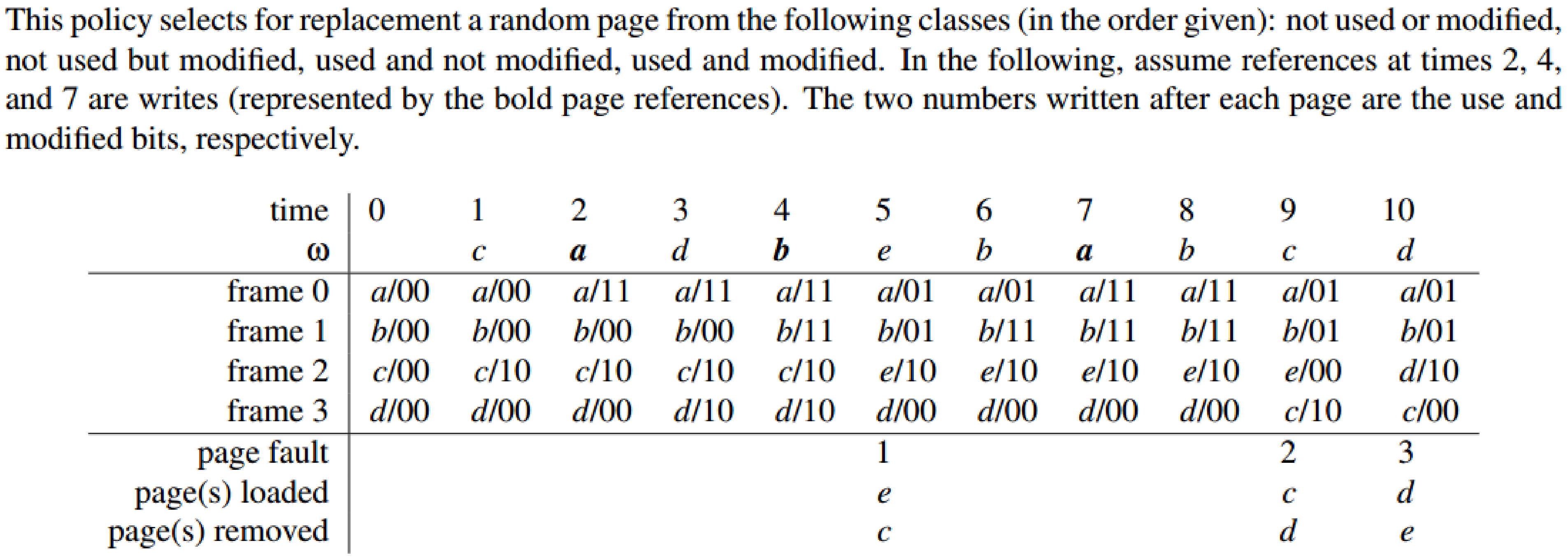


time	0	1	2	3	4	5	6	7	8	9	10
ω		*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
frame 0	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*
frame 1	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*
frame 2	*c*	*c*	*c*	*c*	*c*	*e*	*e*	*e*	*e*	*e*	*d*
frame 3	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*c*	*c*
page fault						1				2	3
page(s) loaded						*e*				*c*	*d*
page(s) removed						*c*				*d*	*e*
stack (top)		*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
		–	*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*
		–	–	*c*	*a*	*d*	*d*	*e*	*e*	*a*	*b*
stack (bottom)		–	–	–	*c*	*a*	*a*	*d*	*d*	*e*	*a*

We shall demonstrate these algorithms by running them on the reference string ω = *cadbebabcd* and assume that, initialy, pages *a*, *b*, *c*, and *d*occupy frames 0, 1, 2, and 3 respectively. When appropriate, the little arrow → indicates the location of the "pointer" that indicates where the search for the next victim will begin.

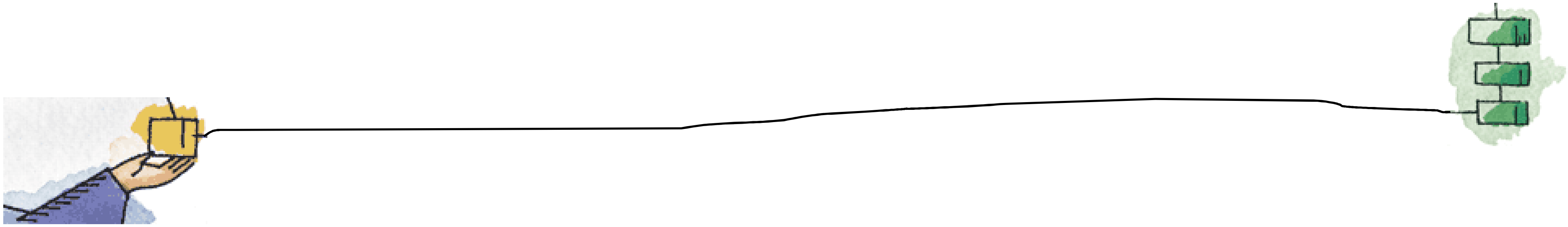


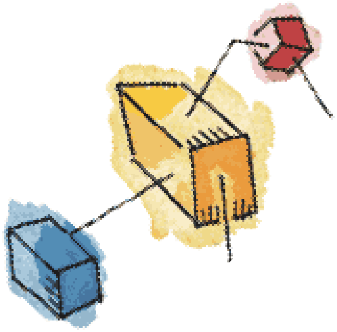
Not-Recently-Used or Not Used Recently (NRU, NUR)



This policy selects for replacement a random page from the following classes (in the order given): not used or modified, not used but modified, used and not modified, used and modified. In the following, assume references at times 2, 4, and 7 are writes (represented by the bold page references). The two numbers written after each page are the use and modified bits, respectively.

time	0	1	2	3	4	5	6	7	8	9	10
ω		*c*	***a***	*d*	***b***	*e*	*b*	***a***	*b*	*c*	*d*
frame 0	*a*/00	*a*/00	*a*/11	*a*/11	*a*/11	*a*/01	*a*/01	*a*/11	*a*/11	*a*/01	*a*/01
frame 1	*b*/00	*b*/00	*b*/00	*b*/00	*b*/11	*b*/01	*b*/11	*b*/11	*b*/11	*b*/01	*b*/01
frame 2	*c*/00	*c*/10	*c*/10	*c*/10	*c*/10	*e*/10	*e*/10	*e*/10	*e*/10	*e*/00	*d*/10
frame 3	*d*/00	*d*/00	*d*/00	*d*/10	*d*/10	*d*/00	*d*/00	*d*/00	*d*/00	*c*/10	*c*/00
page fault						1				2	3
page(s) loaded						*e*				*c*	*d*
page(s) removed						*c*				*d*	*e*

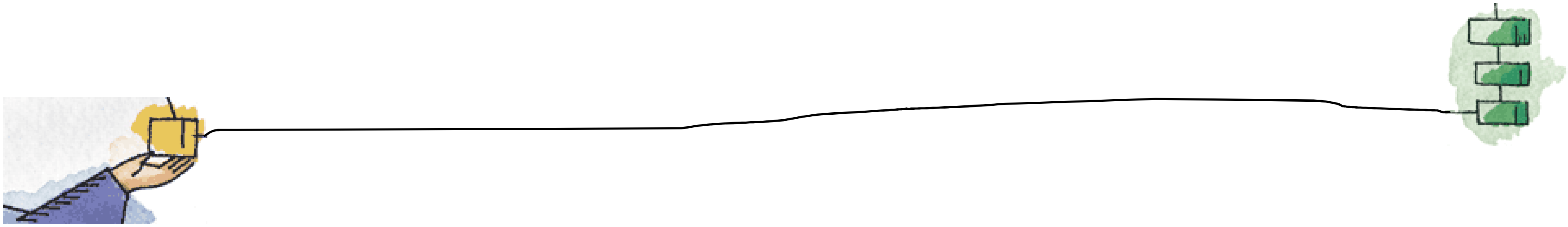


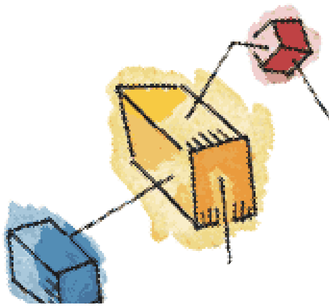


Clock

This policy is similar to LRU and FIFO. Whenever a page is referenced, the use bit is set. When a page must be replaced, the algorithm begins with the page frame pointed to. If the frame's use bit is set, it is cleared and the pointer advanced. If not, the page in that frame is replaced. Here the number after the page is the use bit; we'll assume all pages have been referenced initially.

time	0	1	2	3	4	5	6	7	8	9	10
ω		*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
frame 0	*a*/0	→*a*/0	→*a*/1	→*a*/1	→*a*/1	*e*/1	*e*/1	*e*/1	*e*/1	→*e*/1	*d*/1
frame 1	*b*/0	*b*/0	*b*/0	*b*/0	*b*/1	→*b*/0	→*b*/1	*b*/0	*b*/1	*b*/1	*b*/0
frame 2	*c*/0	*c*/1	*c*/1	*c*/1	*c*/1	*c*/0	*c*/0	*a*/1	*a*/1	*a*/1	*a*/0
frame 3	*d*/0	*d*/0	*d*/0	*d*/1	*d*/1	*d*/0	*d*/0	→*d*/0	→*d*/0	*c*/1	*c*/0
page fault						1		2		3	4
page(s) loaded						*e*		*a*		*c*	*d*
page(s) removed						*a*		*c*		*d*	*e*

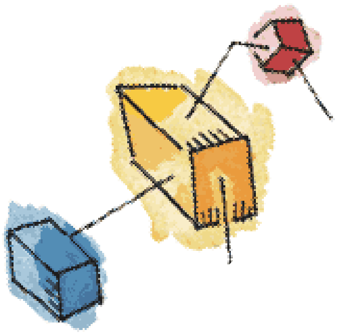




Second-chance Cyclic

This policy merges the clock algorithm and the NRU algorithm. Each page frame has a use and a modified bit. Whenever a page is referenced, the use bit is set; whenever modified, the modify bit is set. When a page must be replaced, the algorithm begins with the page frame pointed to. If the frame's use bit and modify bit are set, the use bit is cleared and the pointer advanced; if the use bit is set but the modify bit is not, the use bit is cleared and the pointer advanced; if the use bit is clear but the modify bit is set, the modify bit is cleared (and the algorithm notes that the page must be copied out before being replaced; here, the page is emboldened) and the pointer is advanced; if both the use and modify bits are clear, the page in that frame is replaced. In the following, assume references at times 2, 4, and 7 are writes (represented by the bold page references). The two numbers written after each page are the use and modified bits, respectively. Initially, all pages have been used but none are modified.

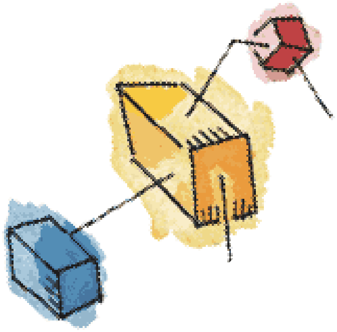
time	0	1	2	3	4	5	6	7	8	9	10
ω		*c*	***a***	*d*	***b***	*e*	*b*	***a***	*b*	*c*	*d*
frame 0	*a*/00	→*a*/00	→*a*/11	→*a*/11	→*a*/11	***a***/00	***a***/00	*a*/11	*a*/11	→*a*/11	***a***/00
frame 1	*b*/00	*b*/00	*b*/00	*b*/00	*b*/11	***b***/00	***b***/10	***b***/10	***b***/10	***b***/10	*d*/10
frame 2	*c*/00	*c*/10	*c*/10	*c*/10	*c*/10	*e*/10	*e*/10	*e*/10	*e*/10	*e*/10	→*e*/00
frame 3	*d*/00	*d*/00	*d*/00	*d*/10	*d*/10	→*d*/00	→*d*/00	→*d*/00	→*d*/00	*c*/10	*c*/00
fault						1				2	3
loaded						*e*				*c*	*d*
removed						*c*				*d*	*e*



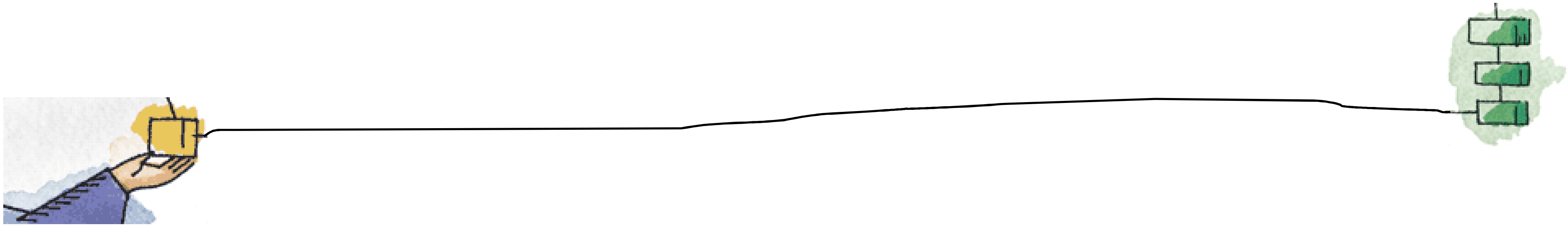
- Process spends more time on paging than executing- > Thrashing occurs
  - Sol: process must have as much frame as they need.

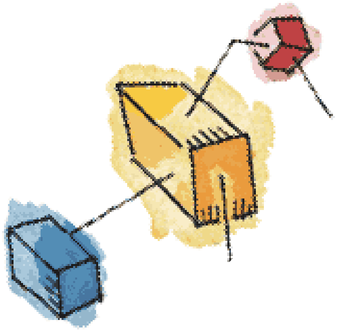
- But how to determine process needs??

- Working set strategy



Variable Number of Frames

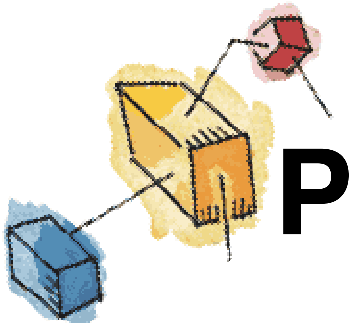




Working Set (WS)

This policy tries to keep all pages in a process' working set in memory. This table shows the pages consitiuting the working set at each reference. Here, we take the working set to be that set of pages which has been referenced during the last $\tau = 4$ units. We also assume that *a* was referenced at time 0, *d* at time −1, and *e* at time −2. The window begins with the current reference.

time	−2	−1	0	1	2	3	4	5	6	7	8	9	10
ω	*e*	*d*	*a*	*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
page *a*	–	–	*a*	*a*	*a*	*a*	*a*	*a*	–	*a*	*a*	*a*	*a*
page *b*	–	–	–	–	–	–	*b*	*b*	*b*	*b*	*b*	*b*	*b*
page *c*	–	–	–	*c*	*c*	*c*	*c*	–	–	–	–	*c*	*c*
page *d*	–	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	–	–	–	*d*
page *e*	*e*	*e*	*e*	*e*	–	–	–	*e*	*e*	*e*	*e*	–	–
page fault				1			2	3		4		5	6
page(s) loaded				*c*			*b*	*e*		*a*		*c*	*d*
page(s) removed					*e*			*c*	*a*	*d*		*e*	

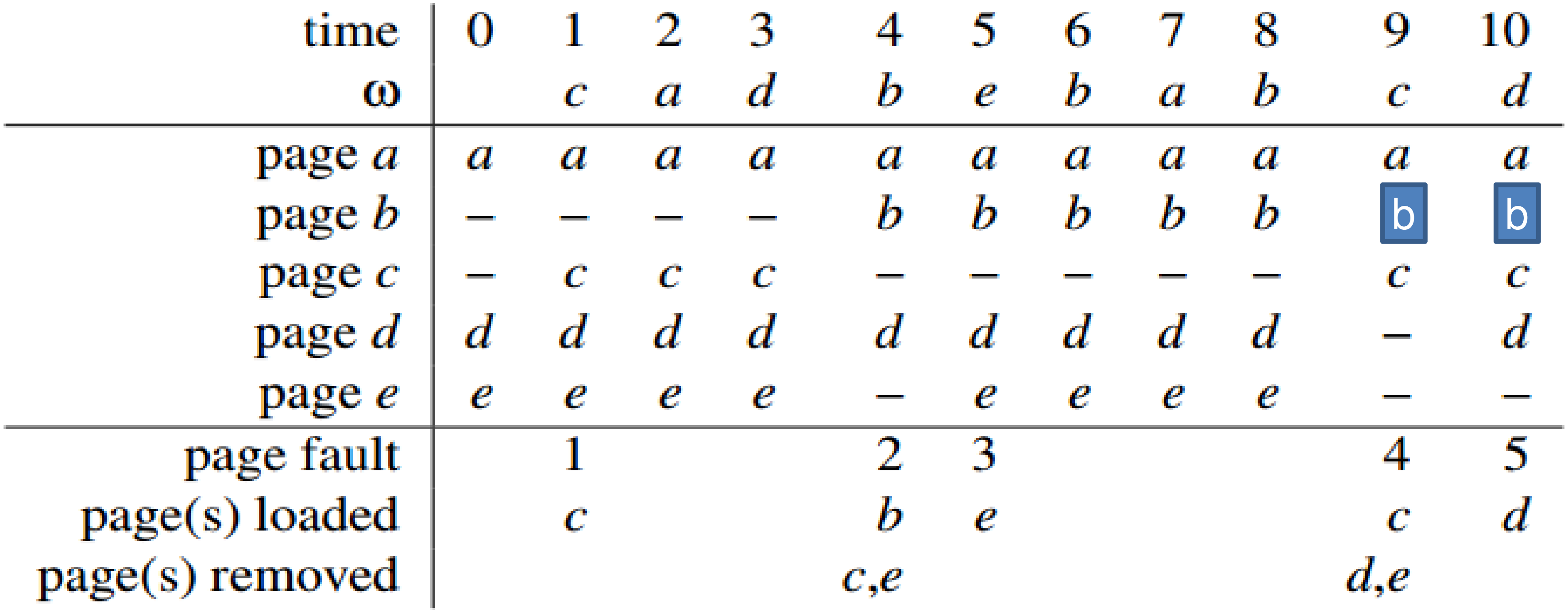


Page Fault Frequency (PFF)

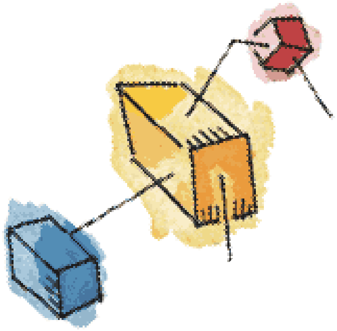
This approximation to the working set policy tries to keep page faulting to some prespecified range. If the time between the current and the previous page fault exceeds some critical value *p*, then all pages not referenced between those page faults are removed. This table shows the pages resident at each reference. Here, we take *p* = 2 units and assume that initially, *a*, *d*, and *e* are resident. This example assumes the interval between page faults does *not* include the reference that caused the previous page fault.

tcurr-tlast>p remove other than [tcurr, tlast]

tcurr-tlast<=p add faulting page only



time	0	1	2	3	4	5	6	7	8	9	10
ω		*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
page *a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*
page *b*	–	–	–	–	*b*	*b*	*b*	*b*	*b*	b	b
page *c*	–	*c*	*c*	*c*	–	–	–	–	–	*c*	*c*
page *d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	–	*d*
page *e*	*e*	*e*	*e*	*e*	–	*e*	*e*	*e*	*e*	–	–
page fault		1			2	3				4	5
page(s) loaded		*c*			*b*	*e*				*c*	*d*
page(s) removed					*c,e*					*d,e*	



Reading Materials

- Chapter 10:
  - – 10.1,10.2,10.4,10.4.1-10.4.4