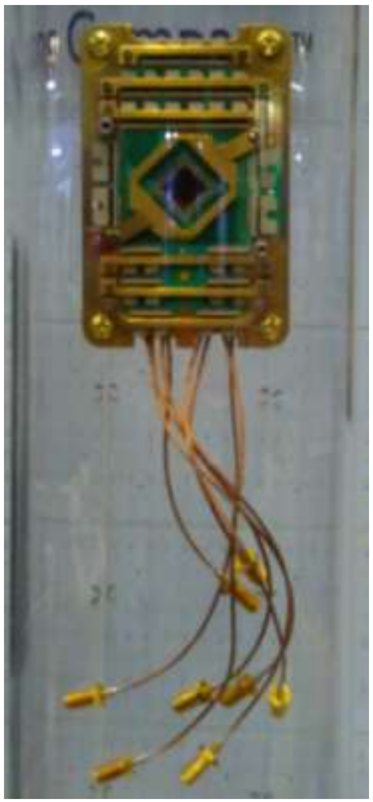
Virtual Memory



Yeah, unrelated to
VM but pretty cool
(Quantum processor)

These slides were compiled from the OSC textbook slides (Silberschatz, Galvin, and Gagne) and the instructor’s class materials.

Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Other considerations

Background

- Virtual memory – abstract main memory into a **large**, uniform array of storage.
  - Only part of the program needs to be in memory for execution.
  - Logical space can be much larger than physical address space
  - Allows addresses to be shared by several processes
  - Allows for more efficient process creation
  - Can be implemented via: **Demand paging or Demand Segmentations**

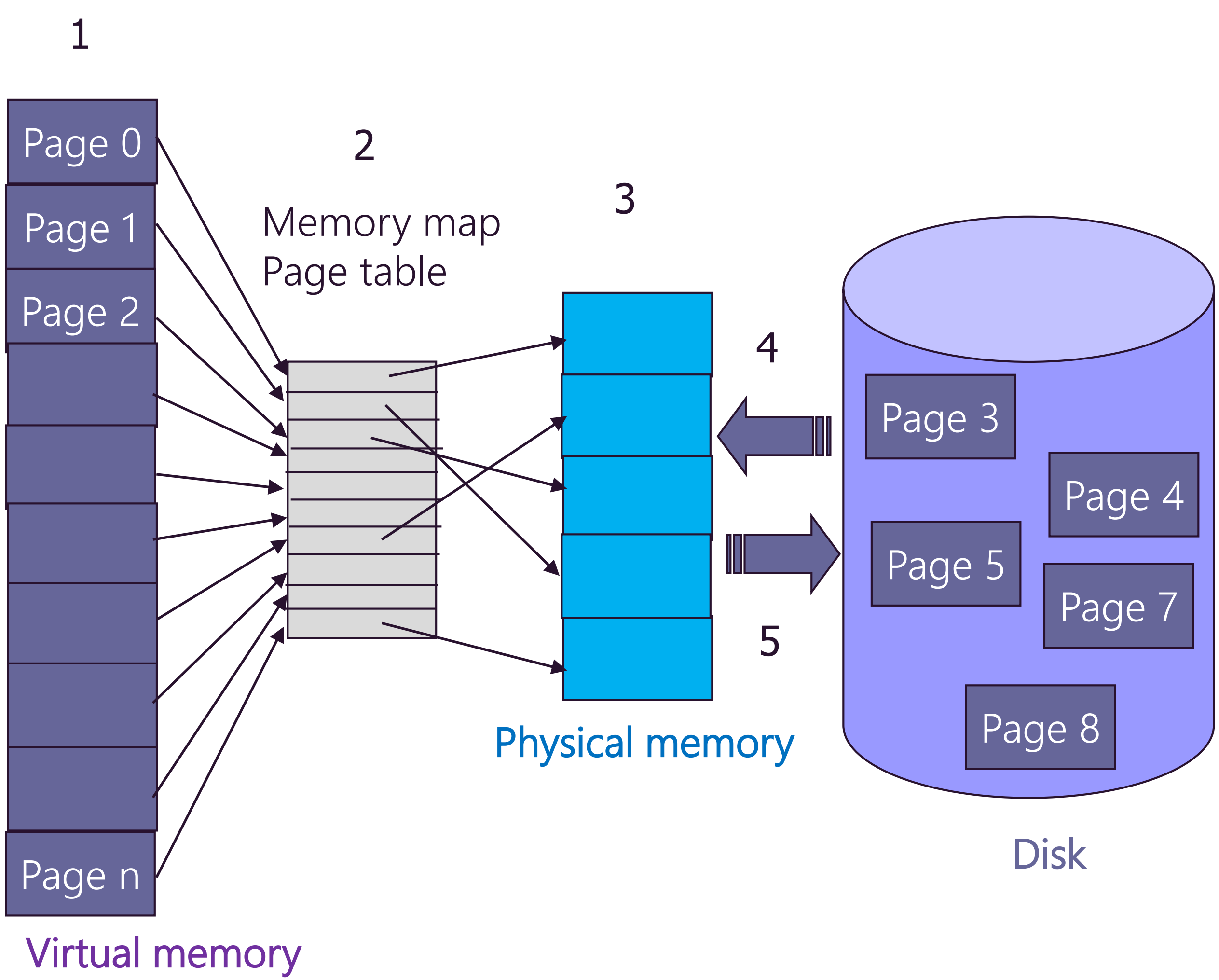
Benefits

- **No more constraints** by physical memory
- Less physical memory needed, thus **increasing multiprogramming** degree
- Less I/O needed, thus **improving performance**

Downside

- Not easy to implement
- Could incur in **decreased performance**

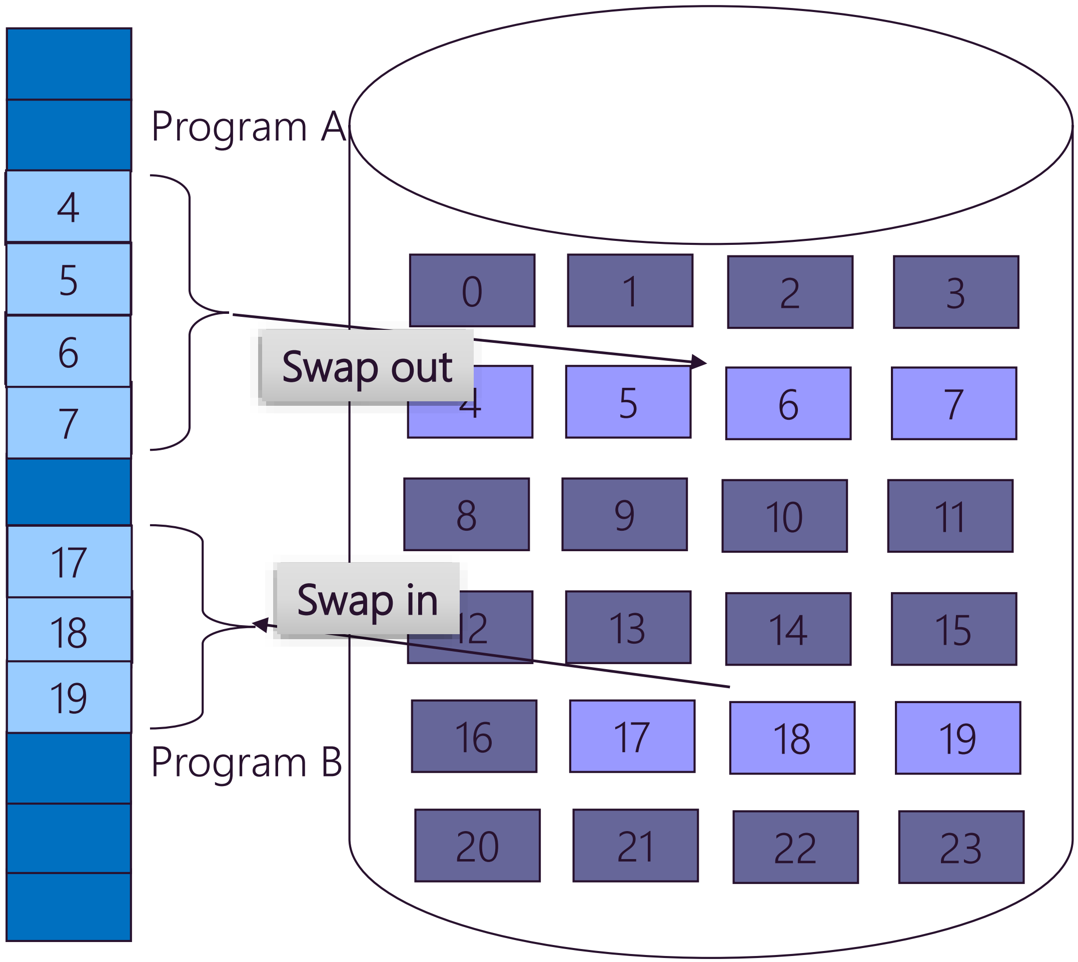
Virtual Memory



General approach

1. Page is referenced
2. Check memory map
3. If the corresponding entry points a physical memory frame: *Reference it*
4. Otherwise, the page does not exit in physical memory: *Bring it from to disk to memory*
5. If physical memory is full,
   - Find a victim page
   - Swap it out to disk

Swapping vs. Demand Paging



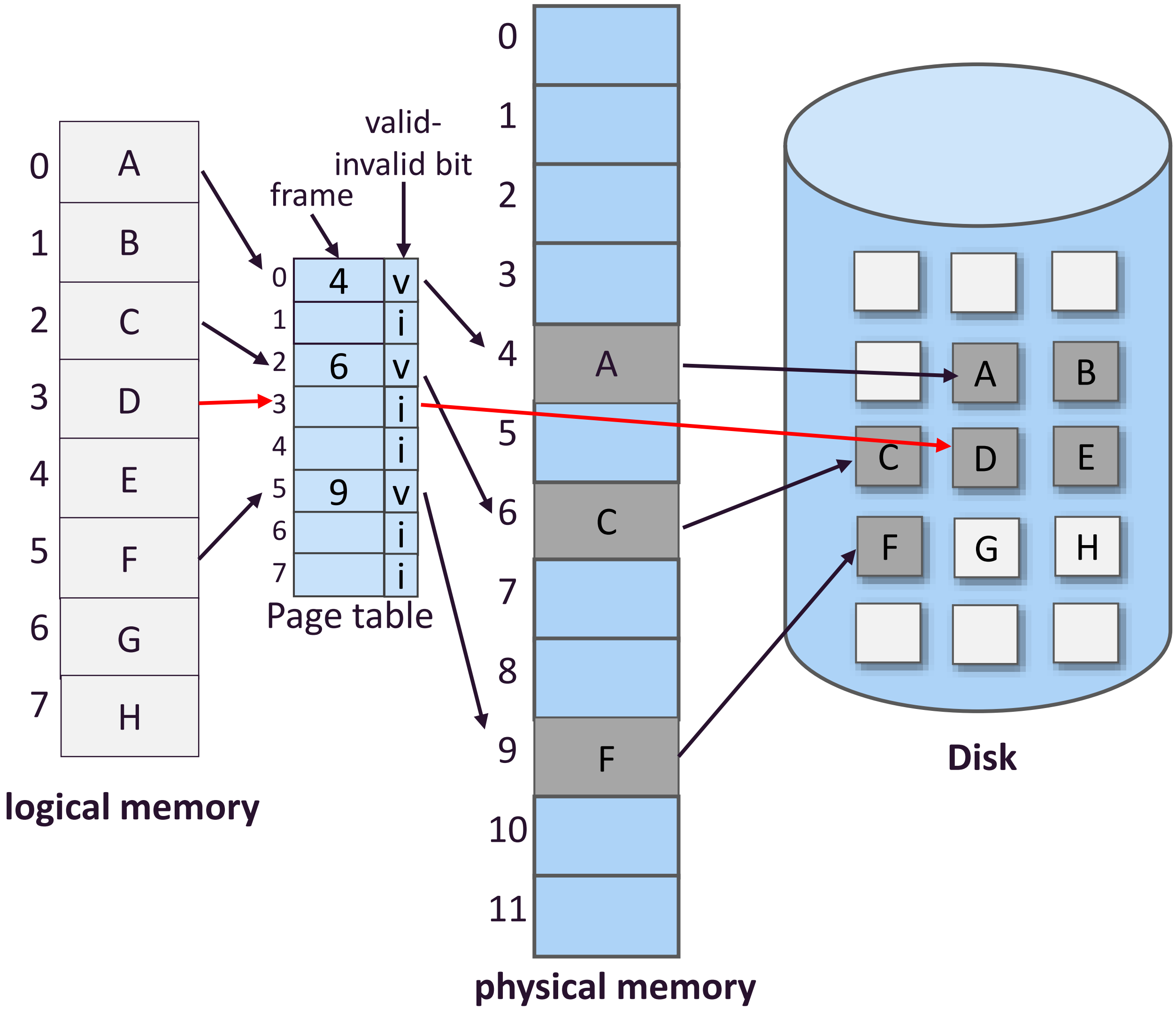
Physical memory

- **Swapping**
  - All pages composing a program are brought into memory and taken out to disk.
- **Demand Paging**
  - Only pages currently referenced by a program are **brought** into memory and taken out to disk.
  - A *lazy swapper* never swaps a page into memory unless that page will be needed

Demand Paging

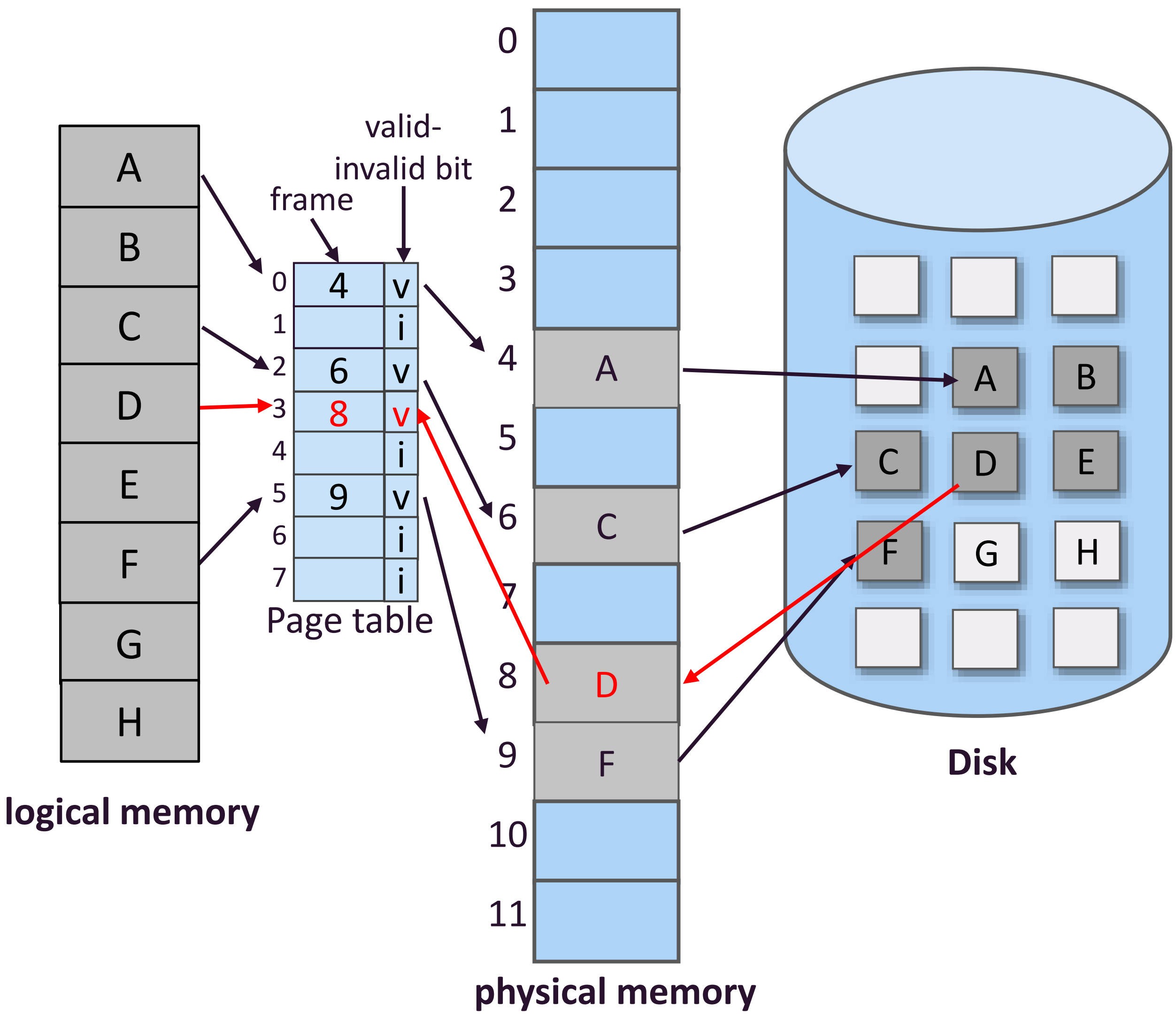
Load only needed pages

Page Table for Managing VM



- Bring-in only needed pages
- Valid/invalid bits indicate if page in memory or on disk
- If frame bit is valid then frame is in physical memory
- Otherwise – page fault occurs

Demand Paging– w/Some Pages not in Main Memory



- If frame bit is valid then frame is in physical memory
- Otherwise – page fault occurs
  - Corresponding page is loaded from disk
  - A new memory space is allocated.

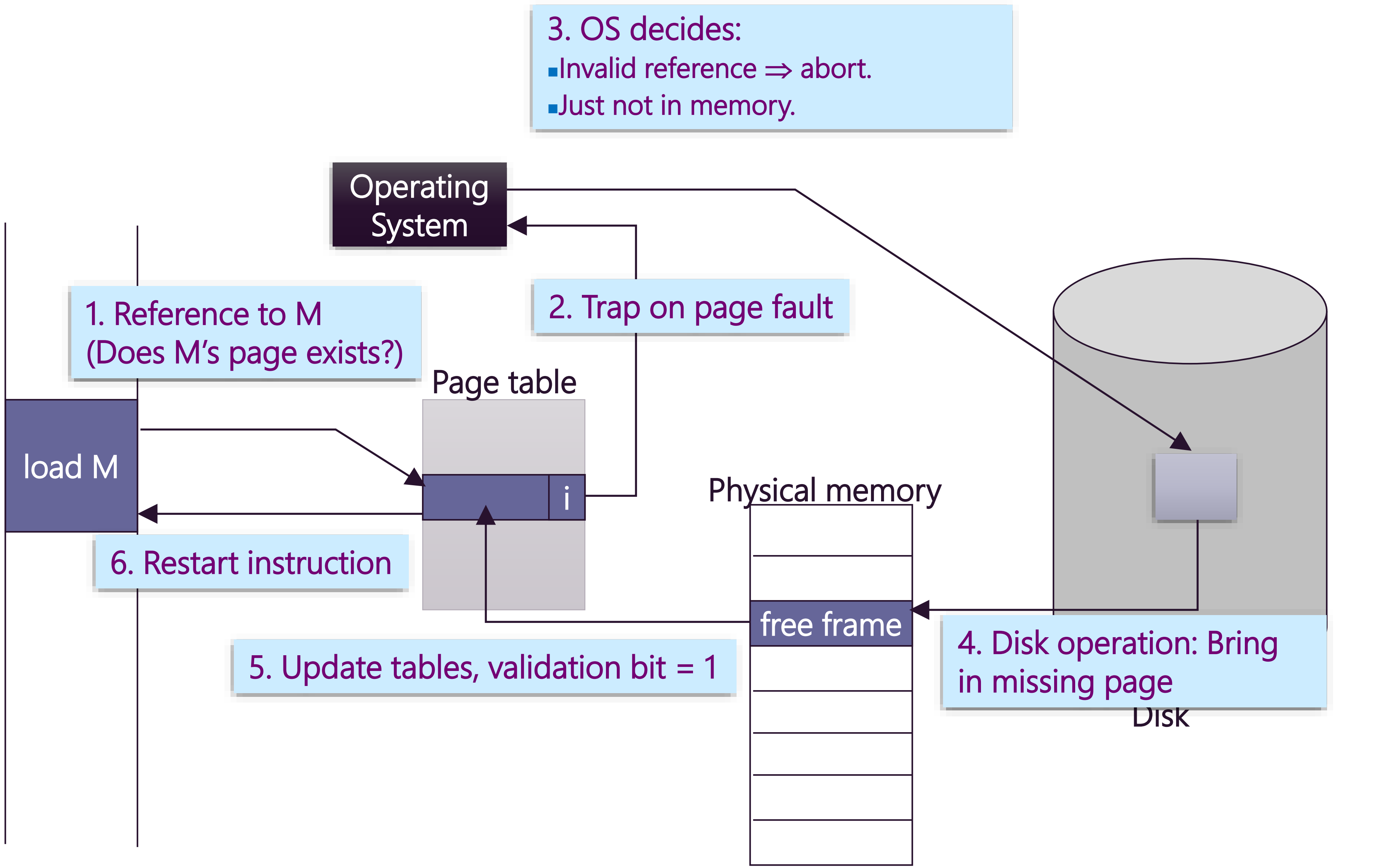
Pure Demand Paging

- **Pure Demand Paging** - never bring a page into memory until it is required.

- Programs tend to have ***locality of reference*** which results in reasonable performance from demand paging.

- A **crucial requirement** for demand paging is the ability to restart any instruction after a page fault

Page Fault



3. OS decides:
▪Invalid reference ⇒ abort.
▪Just not in memory.
Operating System
1. Reference to M
(Does M's page exists?)
2. Trap on page fault
Page table
load M
i
Physical memory
6. Restart instruction
free frame
5. Update tables, validation bit = 1
4. Disk operation: Bring in missing page
Disk

Performance Impact of Demand Paging

- Page Fault Rate $0 \le p \le 1.0$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault

$$\text{Effective Access Time (EAT)} = (1 - p) \times ma + p \times \textit{page-fault time}$$

- *ma* – Memory Access time (typically 10-200 nanoseconds)
- *page-fault time* – page fault overhead

  + swap page out

  + swap page in

  + restart overhead

Demand Paging Example

- Let *ma* = 200 nanoseconds

- Average page-fault time = 8 milliseconds

EAT = $(1 – p) \times 200 + p$ (8 milliseconds)

$= (1 – p \times 200 + p \times 8{,}000{,}000$

$= 200 + p \times 7{,}999{,}800$ (nanoseconds)

- If one access out of 1,000 causes a page fault, then

  EAT = 8.2 microseconds.

  This is a slowdown by a factor of 40!!

Page Replacement

- What happens if there are no more free frames available?
- **Page replacement**: if no free frames, select a **victim**

- Use a **modify/dirty bit** to tag pages: was the page modified since it was read?
  - Yes => need to write it
  - No => no need

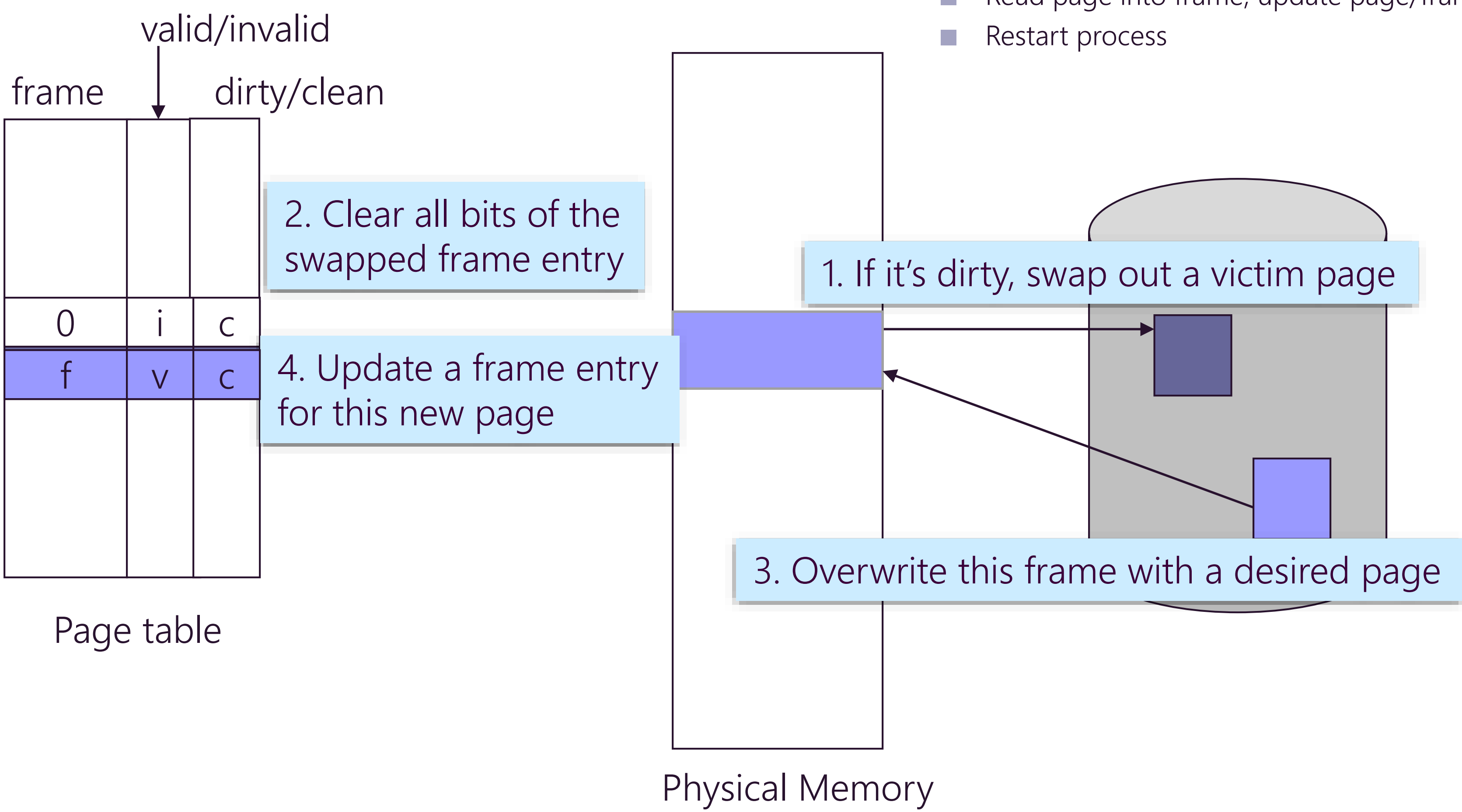
- Main problems to address:
  - **Frame allocation** algorithm
  - **Page replacement** algorithm

- Has a lot of overhead
- How about degree of multiprogramming?

Page Replacement Mechanism

Page fault happens!

- Find desired page on disk
- Find a free frame
  - Free frame→ use it, otherwise find victim frame
  - Write victim to disk, update page/frame tables
- Read page into frame, update page/frame tables
- Restart process



Page Replacement Algorithms

- Want lowest page-fault rate
- Evaluate algorithm by running it on a particular string of memory references (**reference string**) and computing the number of page faults on that string
  - Example of a reference string:

    `7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7`
- **Algorithms**
  - FIFO (The oldest page may be no longer used.)
  - OPT (All page references are known a priori)
  - LRU (The least recently used page may be no longer used.)
  - Second Chance (A simpler form of LRU)

First-In-First-Out (FIFO) Algorithm

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

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
FIFO METHOD																					Page Faults
STRING	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1	
FIFO DEPTH	7	7	1	1	0	0	0	4	4	3	3	3	2	2	2	0	0	0	0	1	
2		0	0	2	2	3	3	3	2	2	0	0	0	1	1	1	1	7	7	7	
	1	1	1	1	1	1		1	1	1	1		1	1		1		1		1	**15**
STRING	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1	
FIFO DEPTH	7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	0	7	7	7	
3		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	
	1	1	1	1		1	1	1	1	1	1			1	1			1	1	1	**15**
STRING	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1	
FIFO DEPTH	7	7	7	7	7	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	
4		0	0	0	0	0	0	4	4	4	4	4	4	4	4	0	0	0	0	0	
			1	1	1	1	1	1	1	1	0	0	0	0	1	1	1	1	1	1	
				2	2	2	2	2	2	2	2	2	2	1	2	2	2	7	2	2	
	1	1	1	1		1		1			1			1	1			1			**10**
STRING	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1	
FIFO DEPTH	7	7	7	7	7	7	7	4	4	4	4	4	4	4	4	4	4	4	4	4	
5		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	7	7	7	
			1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	
				2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	1	
						3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	
	1	1	1	1		1		1										1	1	1	**9**

➢ **more frames ⇒ less page faults** **(is this always true??????)**

Another Example

- Reference string: **`1,2,3,4,1,2,5,1,2,3,4,5`**

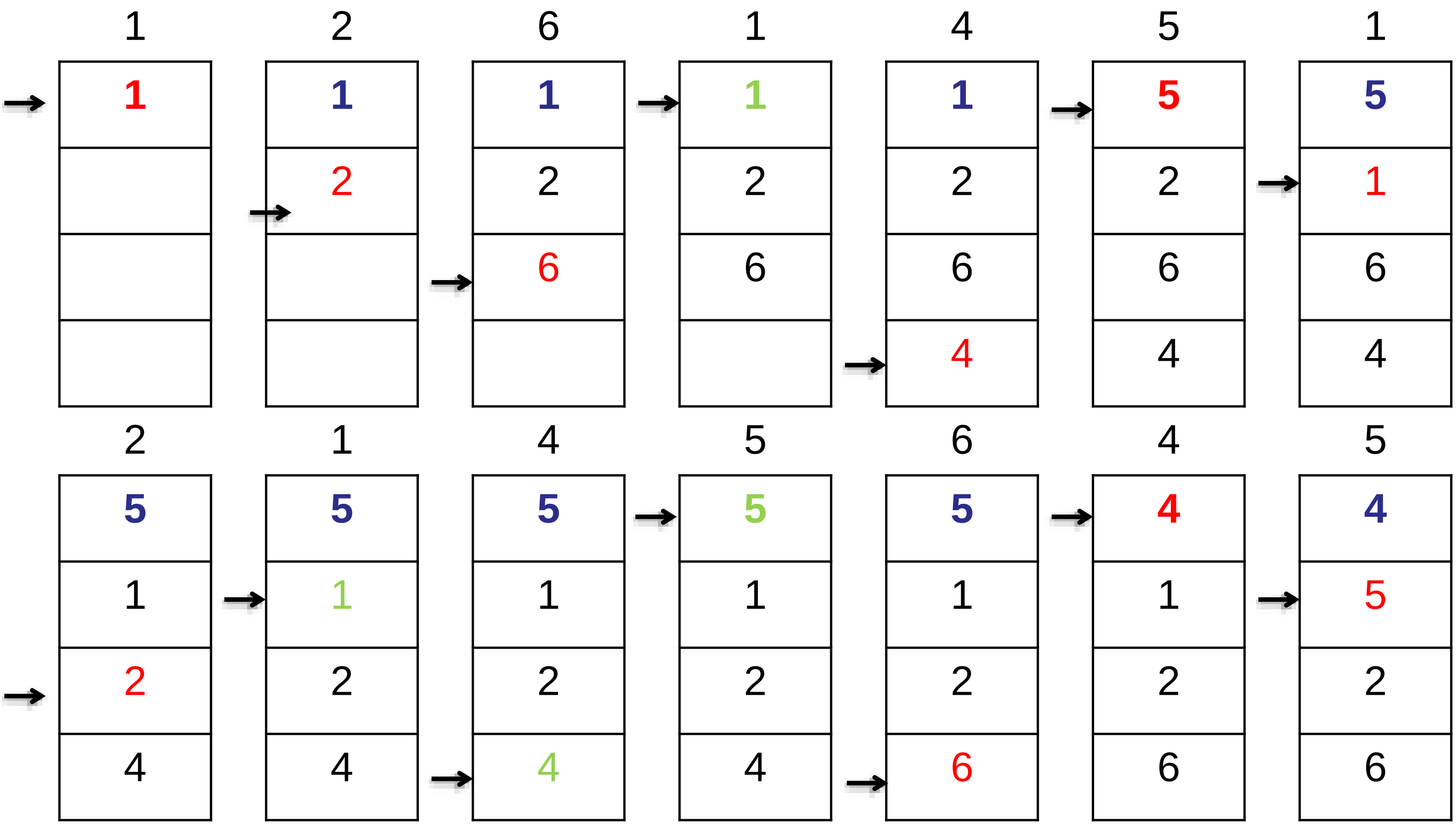
In this example numbers are placed using push

First-In-First-Out Algorithm

1 2 6 1 4 5 1 2 1 4 5 6 4 5 (___ faults)

First-In-First-Out Algorithm

1 2 6 1 4 5 1 2 1 4 5 6 4 5 (___ faults)



Page Fault / Page Hit

Optimal (OPT) Algorithm

- Replace page that will not be used for longest period of time.
- 3 frames example

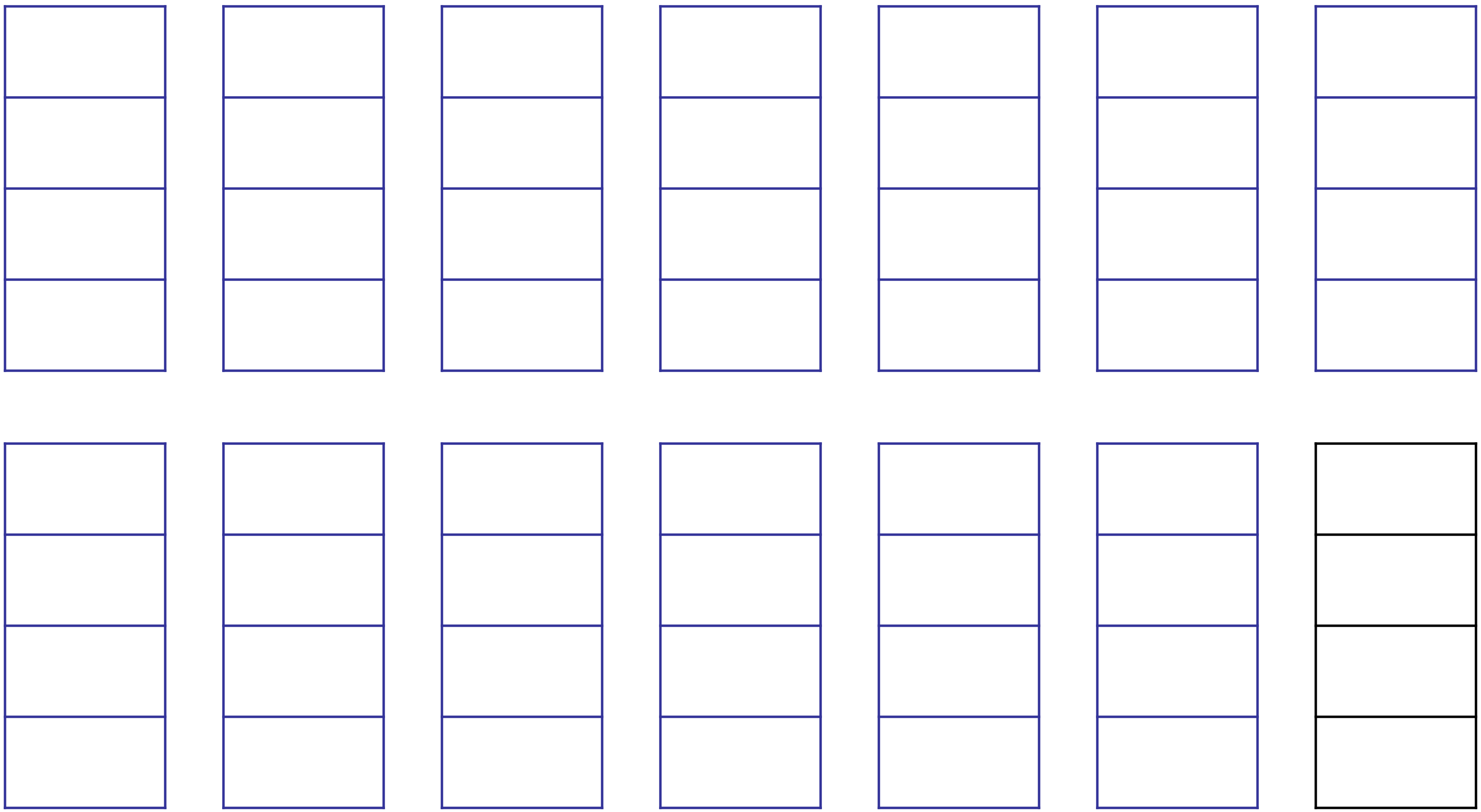
`7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7`

7	7	7	2	2	2	2	2	2	2	2	2	2	2	2	2	2	7
	0	0	0	0	0	0	4	4	4	0	0	0	0	0	0	0	0
		1	1	1	3	3	3	3	3	3	3	3	1	1	1	1	1
Page fault				Page hit		Page hit		Page hit	Page hit		Page hit	Page hit		Page hit	Page hit	Page hit	

- Must be able to "see the future"
- Is the optimal (minimum number of page faults)
- May not be practical (like SJF algorithm)
- Used for measuring algorithm performance

OPT Algorithm (w=2 using stack)

1 2 6 1 4 5 1 2 1 4 5 6 4 5 (_ faults)



OPT (Optimal) Algorithm (w=2, stack)

1 2 6 1 4 5 1 2 1 4 5 6 4 5 – (6 faults)

Note that in this example when the memory is not full the referenced number is moved to the top of the stack, when the memory is full, it doesn't move. You don't need to do it like this (you can keep same behaviour across all cases)

1	2	6	-	4	5	-
				4	4	4
		6	1	1	1	1
	2	2	6	6	**5**	5
1	1	1	2	2	2	2

-	-	-	-	6	-	-
4	4	4	4	4	4	4
1	1	1	1	**6**	6	6
5	5	5	5	5	5	5
2	2	2	2	2	2	2

Least Recently Used (LRU) Algorithm

- Replace the page that has not been used for the longest period of time
- Use the recent past as an approximation of the near future
- Reference string: **`7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7`**

7	7	7	2	2	2	2	4	4	4	0	0	0	1	1	1	1	1
	0	0	0	0	0	0	0	0	3	3	3	3	3	3	0	0	0
		1	1	1	3	3	3	2	2	2	2	2	2	2	2	2	7
Page fault				Page hit		Page hit					Page hit	Page hit		Page hit		Page hit	

- **Counter implementation**
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter.
  - When a page needs to be changed, look at the counters to determine which are to change.
- **Stack implementation** – keep a stack of page numbers in a double link form:
  - *Page referenced, move it to the top*
  - No search for replacement

LRU (Least Recently Used) Algorithm

1 2 6 1 4 5 1 2 1 4 5 6 4 5 (_ faults)

LRU Algorithm - Stack

1 2 6 1 4 5 1 2 1 4 5 6 4 5 – (7 faults)

1	2	6	-	4	5	-
				4	**5**	**1**
		6	1	1	4	5
	2	2	6	6	1	4
1	1	1	2	2	6	6

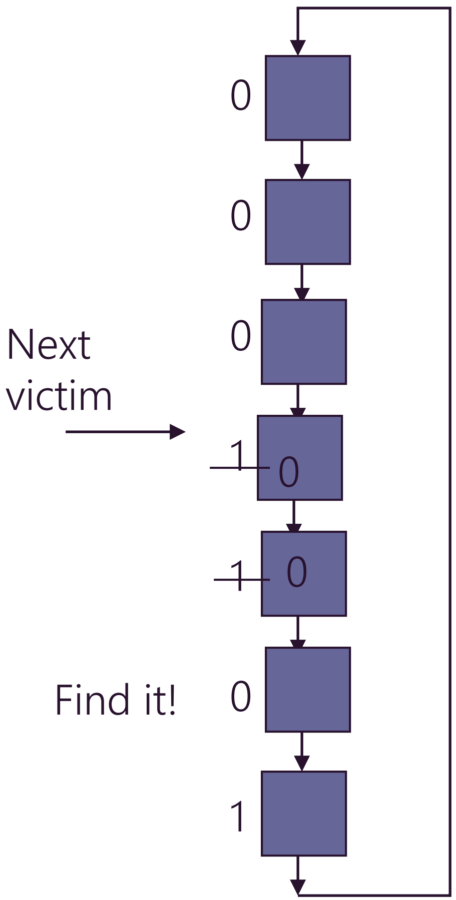
2	-	-	-	6	-	-
2	**1**	**4**	**5**	**6**	**4**	**5**
1	2	1	4	5	6	4
5	5	2	1	4	5	6
4	4	5	2	1	1	1

Clock (second-chance) algorithm

- Goal: remove a page that has not been referenced recently
  - good LRU-approximate algorithm

- Concept of Operation
  - A reference bit per page
  - Memory reference: hardware sets bit to 1
  - Page replacement: OS finds a page with reference bit cleared
  - OS traverses all pages, clearing bits over time
  - Combining FIFO with LRU: give the page FIFO selects to replace a second chance
  - Operates on an independent sampling clock than the page replacement frequency

Second-Chance (Clock) Algorithm

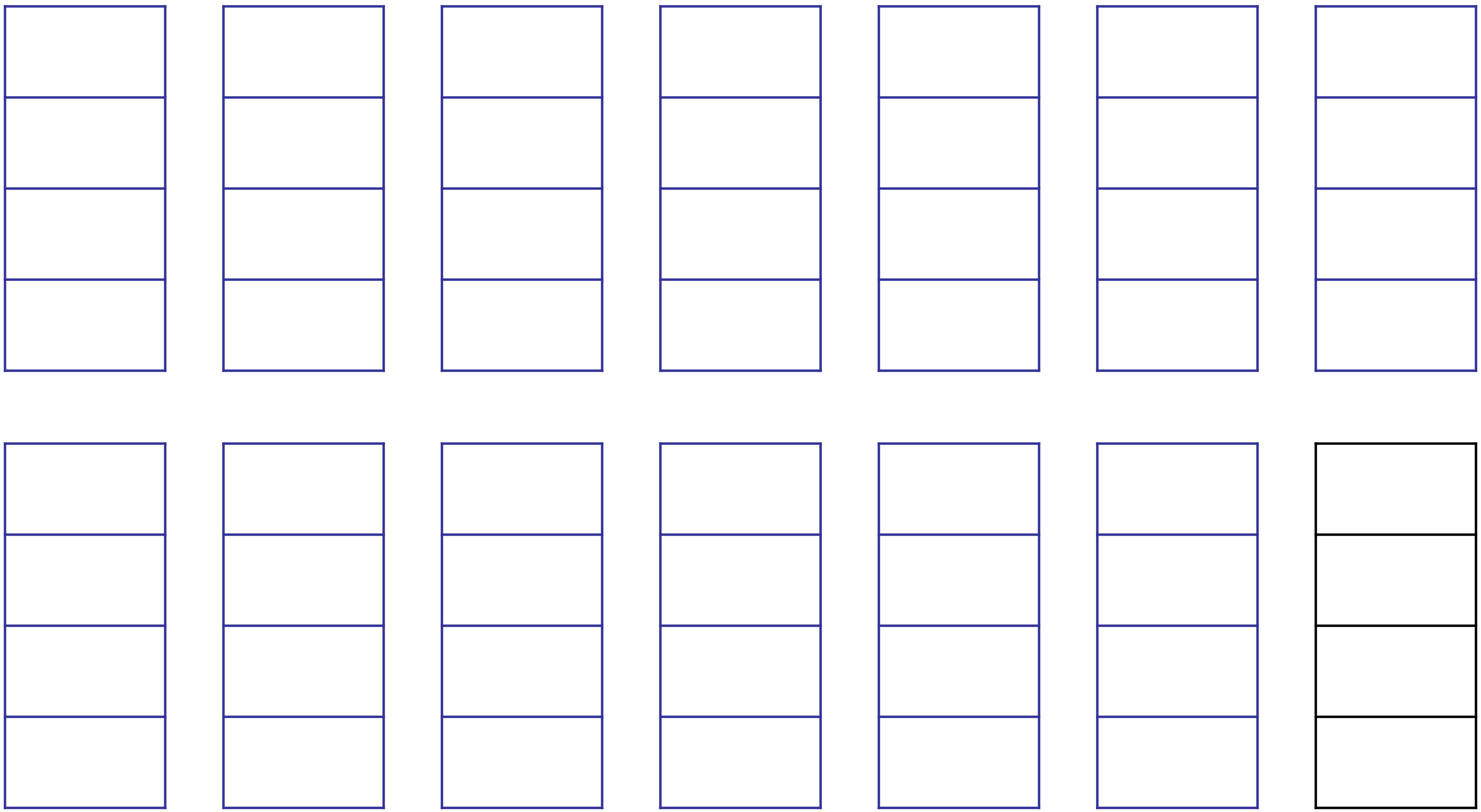


- **Reference bit**
  - Each page has a reference bit, initially = 0
  - When page is referenced, bit set to 1.
- **Algorithm**
  - All pages are maintained in a circular list.
  - If page to be replaced,
    1. If the next victim pointer points to a page whose reference bit = 0

       Replace it with a new brought-in page
    2. Otherwise, (i.e.,if the reference bit = 1), reset the bit to 0.
    3. Advance the next victim pointer.
    4. Go to 1.

Second Chance Algorithm

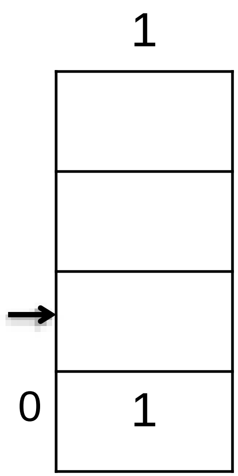
1 2 6 1 4 5 1 2 1 4 5 6 4 5 (_ faults)



Second Chance Algorithm

1 2 6 1 4 5 1 2 1 4 5 6 4 5 (_ faults)

Pay close attention to the pointer →

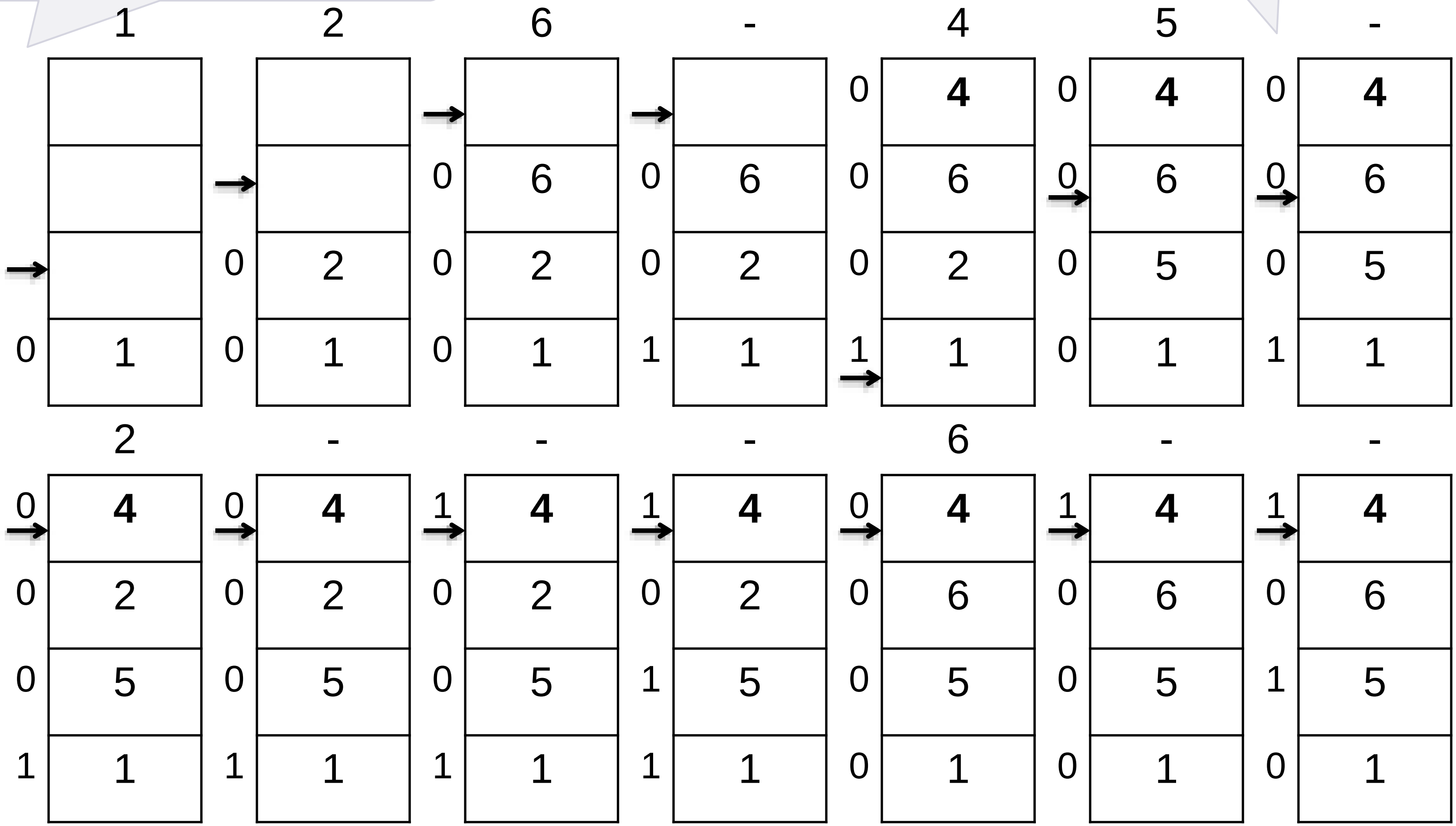


Second Chance Algorithm

1 2 6 1 4 5 1 2 1 4 5 6 4 5 (_ faults)

Pay close attention to the pointer →

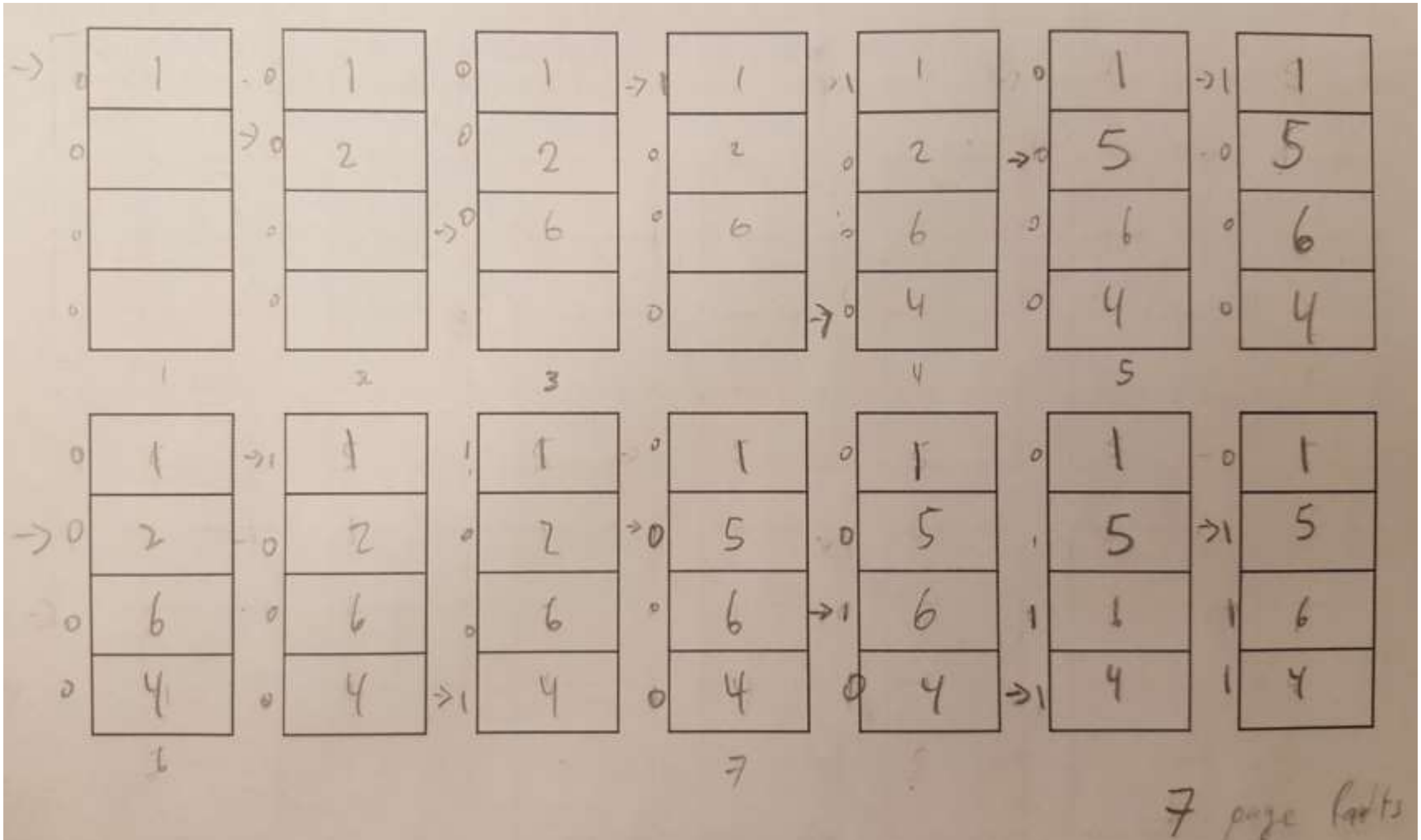
Ref bit is 0 when page is loaded, switch to 1 when referenced



Second Chance Algorithm

An alternative way of solving

1 2 6 1 4 5 1 2 1 4 5 6 4 5 (_ faults)



Frame Allocation

Allocation of Frames

- How many frames should each process get?
- Must allocate **enough** frames to hold all the different pages that any single instruction can reference
- Minimum number defined by the computer architecture
- Two major allocation schemes
  - Fixed (or equal) allocation
  - Priority allocation

Page Allocation Examples

#frames = m #processes = n

- Minimum number of frames
- **Equal allocation:** m/n frames are allocated to each process.
- **Proportional allocation**: allocate according to process size

$$s_i = \text{size of process } p_i$$
$$S = \sum s_i$$
$$m = \text{total number of frames}$$
$$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$$

$$m = 64$$
$$s_1 = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \text{ ´ } 64 \text{ » } 5$$
$$a_2 = \frac{127}{137} \text{ ´ } 64 \text{ » } 59$$

- This allocation is achieved regardless of process priority

Priority Allocation

- Priority allocation
  - A process with a **higher priority** receives more frames.
  - Use a *proportional* allocation scheme using *priorities* rather than size
- If process P_i generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with *lower priority number*

Global vs. Local Allocation

- Global replacement – process selects a replacement frame from the set of all frames; one process can take a frame from another

- Local replacement – each process selects from only its own set of allocated frames

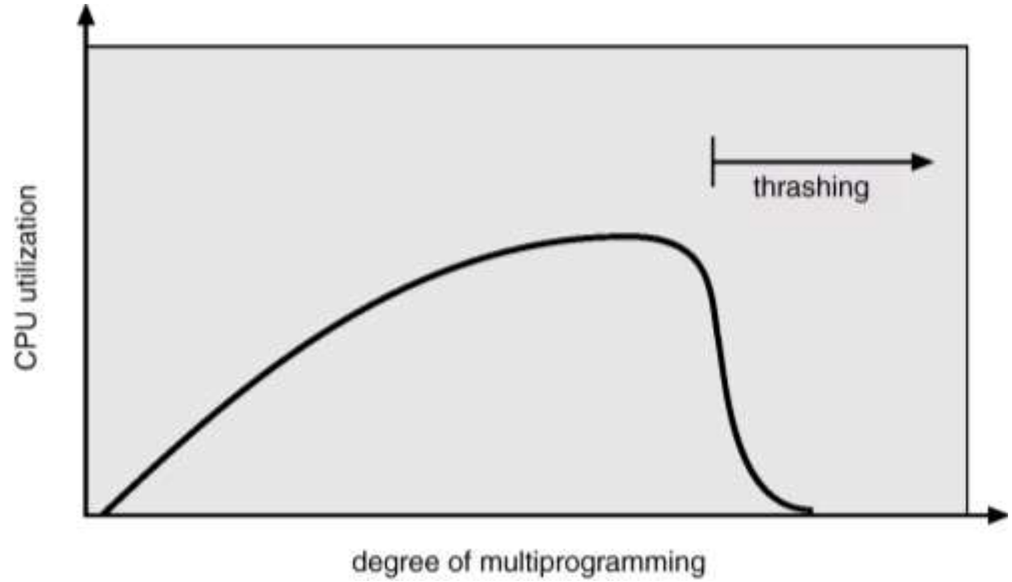
Working Set and Thrashing

Thrashing

- If there are not "enough" pages, the page-fault rate goes very high. This leads to:
  - low CPU utilization.
  - operating system thinks that it needs to increase the degree of multiprogramming.
  - another process added to the system.
  - The new process requesting pages

This is called Thrashing

- Limit effects of thrashing by using a local replacement algorithm (or priority replacement algorithm)



Busy swapping pages in and out

Page Replacement and Locality

- **Local** (or **priority**) **replacement** algorithm
  - **Effect**: A thrashing process cannot steal frames from another process and thus trashing will not be disseminated.
  - **Problem**: Thrashing processes causes frequent page faults and thus degrade the average service time.

- **Locality**
  - A memory location or related locations are frequently accessed
  - Three types of locality

1. **Temporal**: a resourced referenced at one point in time is referenced again soon
2. **Spatial**: there is a greater probability of referencing a location nearby
3. **Sequential**: storage is accessed sequentially

Locality and Thrashing

- **Locality model**
  - **Locality:** Set of pages actively used together, (e.g., a function)
  - Principle behind caching
  - Process migrates from one locality to another
  - Localities may overlap

- **Thrashing occurs**
  - When Σ size of locality > total memory size
  - What to do? → suspend some processes

Discussion: Program Structure Considerations

- Program structure
  - **`Int[1024,1024] data;`**

  Each row is stored in one page (4K pages).

  Which program performs better (less page faults)?

  - Program 1

```
for (j = 0; j <1024; j++)
     for (i = 0; i < 1024; i++)
          data[i,j] = 0;
```

1024 x 1024 = 1M page faults!

  - Program 2

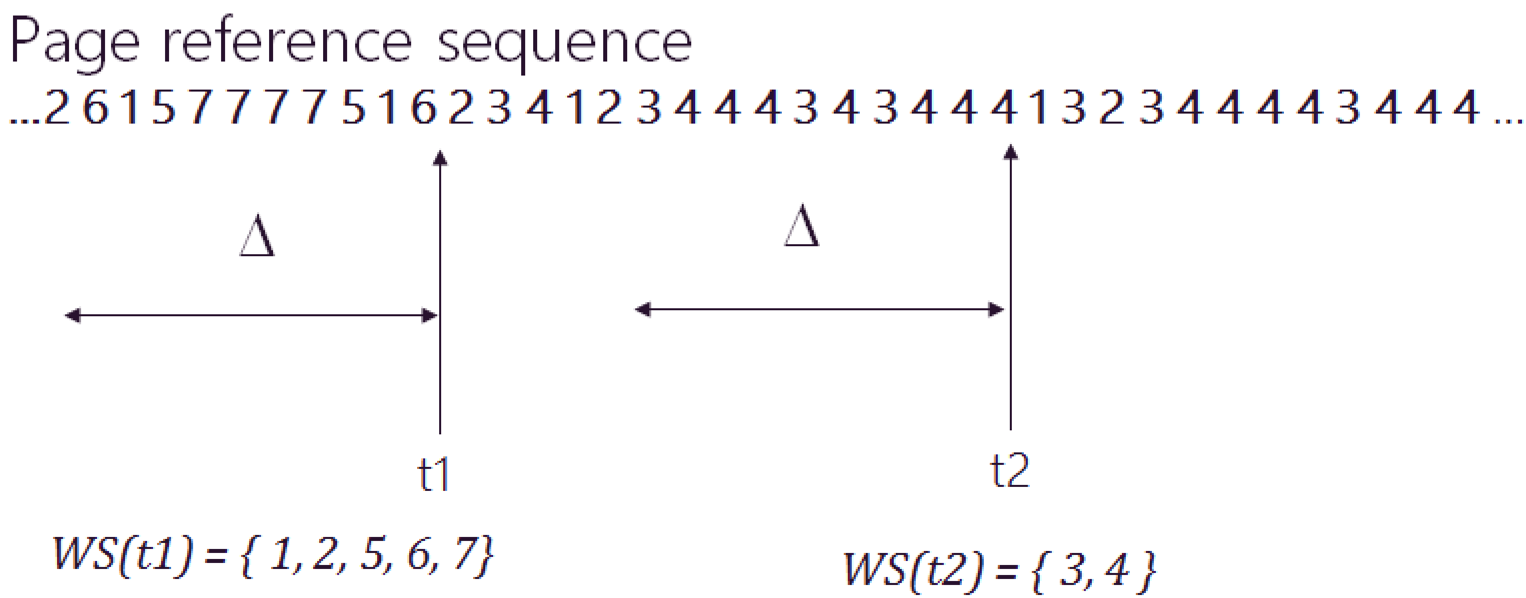
```
for (i = 0; i < 1024; i++)
     for (j = 0; j < 1024; j++)
          data[i,j] = 0;
```

1024 page faults

Working-Set Model

Working-set is an approximation of the program's locality

- Based on the assumption of locality
- **Working-set** (*WS*) window: $\Delta \equiv$ a fixed number of page references
- Set of *active* pages
- WS ***size*** (WSS_i) is the most relevant quality



Controlling Multiprogramming Level by Working-Set Model

- WSS_i (WS size of Process P_i) = total # of pages referenced in the most recent Δ
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities.
  - if $\Delta = \infty \Rightarrow$ will encompass entire program.

- $D = \Sigma\, WSS_i \equiv$ total frames demanded by all processes
- if $D > m \Rightarrow$ *Thrashing* (m is # available frames)
  - Policy if $D > m$, then suspend one of the processes.

When $\Delta = 3$, $m = 8$

P0:		1,	2,	3,	2,	3,	3,	2,	2,	1,	2,	3,	2,	1,	2,	3,	1
WSS0		1	2	3	2	2	2	2	2	2	2	3	2	3	2	3	3
P1:		5,	5,	5,	6,	7,	5,	7,	7,	7,	6,	6,	7,	5,	7,	6,	7
WSS1		1	1	1	2	3	3	2	2	1	2	2	2	3	2	3	2
P2:		8,	9,	9,	8,	4,	8,	8,	8,	4,	9,	8,	4,	9,	8,	4,	9
WSS2		1	2	2	2	3	2	2	1	2	3	3	3	3	3	3	3
ΣWSS_i		3	5	6	6	8	7	6	5	4	7	8	7	9	7	9	8

Thrashing