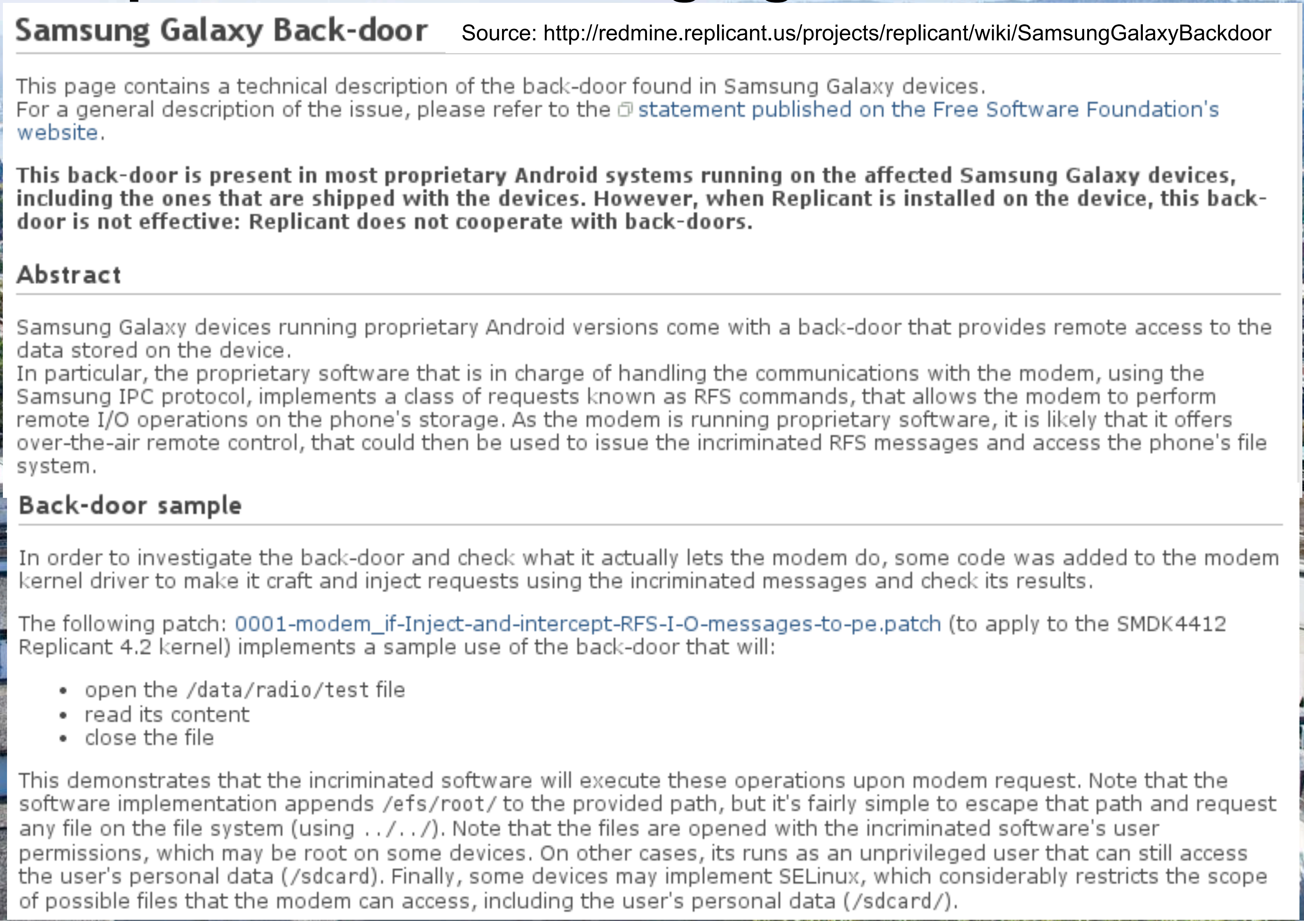
ADRIAN PERRIG & TORSTEN HOEFLER

Networks and Operating Systems (252-0062-00)
Chapter 6: Demand Paging



Source: http://redmine.replicant.us/projects/replicant/wiki/SamsungGalaxyBackdoor

Samsung Galaxy Back-door

This page contains a technical description of the back-door found in Samsung Galaxy devices.
For a general description of the issue, please refer to the statement published on the Free Software Foundation's website.

This back-door is present in most proprietary Android systems running on the affected Samsung Galaxy devices, including the ones that are shipped with the devices. However, when Replicant is installed on the device, this back-door is not effective: Replicant does not cooperate with back-doors.

Abstract

Samsung Galaxy devices running proprietary Android versions come with a back-door that provides remote access to the data stored on the device.
In particular, the proprietary software that is in charge of handling the communications with the modem, using the Samsung IPC protocol, implements a class of requests known as RFS commands, that allows the modem to perform remote I/O operations on the phone's storage. As the modem is running proprietary software, it is likely that it offers over-the-air remote control, that could then be used to issue the incriminated RFS messages and access the phone's file system.

Back-door sample

In order to investigate the back-door and check what it actually lets the modem do, some code was added to the modem kernel driver to make it craft and inject requests using the incriminated messages and check its results.

The following patch: 0001-modem_if-Inject-and-intercept-RFS-I-O-messages-to-pe.patch (to apply to the SMDK4412 Replicant 4.2 kernel) implements a sample use of the back-door that will:

- open the `/data/radio/test` file
- read its content
- close the file

This demonstrates that the incriminated software will execute these operations upon modem request. Note that the software implementation appends `/efs/root/` to the provided path, but it's fairly simple to escape that path and request any file on the file system (using `../../`). Note that the files are opened with the incriminated software's user permissions, which may be root on some devices. On other cases, its runs as an unprivileged user that can still access the user's personal data (/sdcard). Finally, some devices may implement SELinux, which considerably restricts the scope of possible files that the modem can access, including the user's personal data (/sdcard/).

If you miss a key …

- … after yesterday's exercise session …

- … pick it up here!

Next Thursday (March 12th)

- **No lecture!**
- **Friday will be a lecture ☺!**

TLB shootdown

TLB management

- **Recall: the TLB is a *cache*.**
- **Machines have many MMUs on many cores**
  ⇒ many TLBs
- **Problem: TLBs should be coherent. Why?**
  - Security problem if mappings change
  - E.g., when memory is reused

TLB management

	Process ID	VPN	PPN	access
Core 1 TLB:	0	0x0053	0x03	r/w
	1	0x20f8	0x12	r/w
Core 2 TLB:	0	0x0053	0x03	r/w
	1	0x0001	0x05	read
Core 3 TLB:	0	0x20f8	0x12	r/w
	1	0x0001	0x05	read

TLB management

	Process ID	VPN	PPN	access
Core 1 TLB:	0	0x0053	0x03	r/w
	1	0x20f8	0x12	r/w
Core 2 TLB:	0	0x0053	0x03	r/w
	1	0x0001	0x05	read
Core 3 TLB:	0	0x20f8	0x12	r/w
	1	0x0001	0x05	read

Change to read only

TLB management

	Process ID	VPN	PPN	access
Core 1 TLB:	0	0x0053	0x03	r/w
	1	0x20f8	0x12	r/w
Core 2 TLB:	0	~~0x0053~~	0x03	r/w
	1	0x0001	0x05	read
Core 3 TLB:	0	0x20f8	0x12	r/w
	1	0x0001	0x05	read

Change to read only

TLB management

	Process ID	VPN	PPN	access
Core 1 TLB:	0	0x0053	0x03	r/w
	1	0x20f8	0x12	r/w
Core 2 TLB:	0	~~0x0053~~	0x03	r/w
	1	0x0001	0x05	read
Core 3 TLB:	0	0x20f8	0x12	r/w
	1	0x0001	0x05	read

Change to read only

Process 0 on core 1 can only continue once shootdown is complete!

Keeping TLBs consistent

1. **Hardware TLB coherence**
   - Integrate TLB mgmt with cache coherence
   - Invalidate TLB entry when PTE memory changes
   - Rarely implemented
2. **Virtual caches**
   - Required cache flush / invalidate will take care of the TLB
   - High context switch cost!
     $\Rightarrow$ Most processors use physical caches
5. **Software TLB shootdown**
   - Most common
   - OS on one core notifies all other cores - Typically an IPI
   - Each core provides local invalidation
6. **Hardware shootdown instructions**
   - Broadcast special address access on the bus
   - Interpreted as TLB shootdown rather than cache coherence message
   - E.g., PowerPC architecture

Our Small Quiz

- **True or false (raise hand)**
  1. Base (relocation) and limit registers provide a full virtual address space
  2. Base and limit registers provide protection
  3. Segmentation provides a base and limit for each segment
  4. Segmentation provides a full virtual address space
  5. Segmentation allows libraries to share their code
  6. Segmentation provides linear addressing
  7. Segment tables are set up for each process in the CPU
  8. Segmenting prevents internal fragmentation
  9. Paging prevents internal fragmentation
  10. Protection information is stored at the physical frame
  11. Pages can be shared between processes
  12. The same page may be writeable in proc. A and write protected in proc. B
  13. The same physical address can be references through different addresses from (a) two different processes – (b) the same process?
  14. Inverted page tables are faster to search than hierarchical (asymptotically)

Today

- **Uses for virtual memory**
- **Copy-on-write**
- **Demand paging**
  - Page fault handling
  - Page replacement algorithms
  - Frame allocation policies
  - Thrashing and working set

- **Book: OSPP Sections 9.5, 9.7 (all of 9 as refresh)**

Recap: Virtual Memory

- **User logical memory ≠ physical memory.**
  - Only part of the program must be in RAM for execution
    ⇒ Logical address space can be larger than physical address space
  - Address spaces can be shared by several processes
  - More efficient process creation
- ***Virtualize* memory using software+hardware**

The many uses of address translation

- Process isolation
- IPC
- Shared code segments
- Program initialization
- Efficient dynamic memory allocation
- Cache management
- Program debugging
- Efficient I/O
- Memory mapped files
- Virtual memory
- Checkpoint and restart
- Persistent data structures
- Process migration
- Information flow control
- Distributed shared memory

and many more …

Copy-on-write (COW)

Recall `fork()`

- **Can be expensive to create a complete copy of the process' address space**
  - Especially just to do `exec()`!
- **`vfork()`: shares address space, doesn't copy**
  - Fast
  - Dangerous – two writers to same heap
- **Better: only copy when you know something is going to get written**

Copy-on-Write

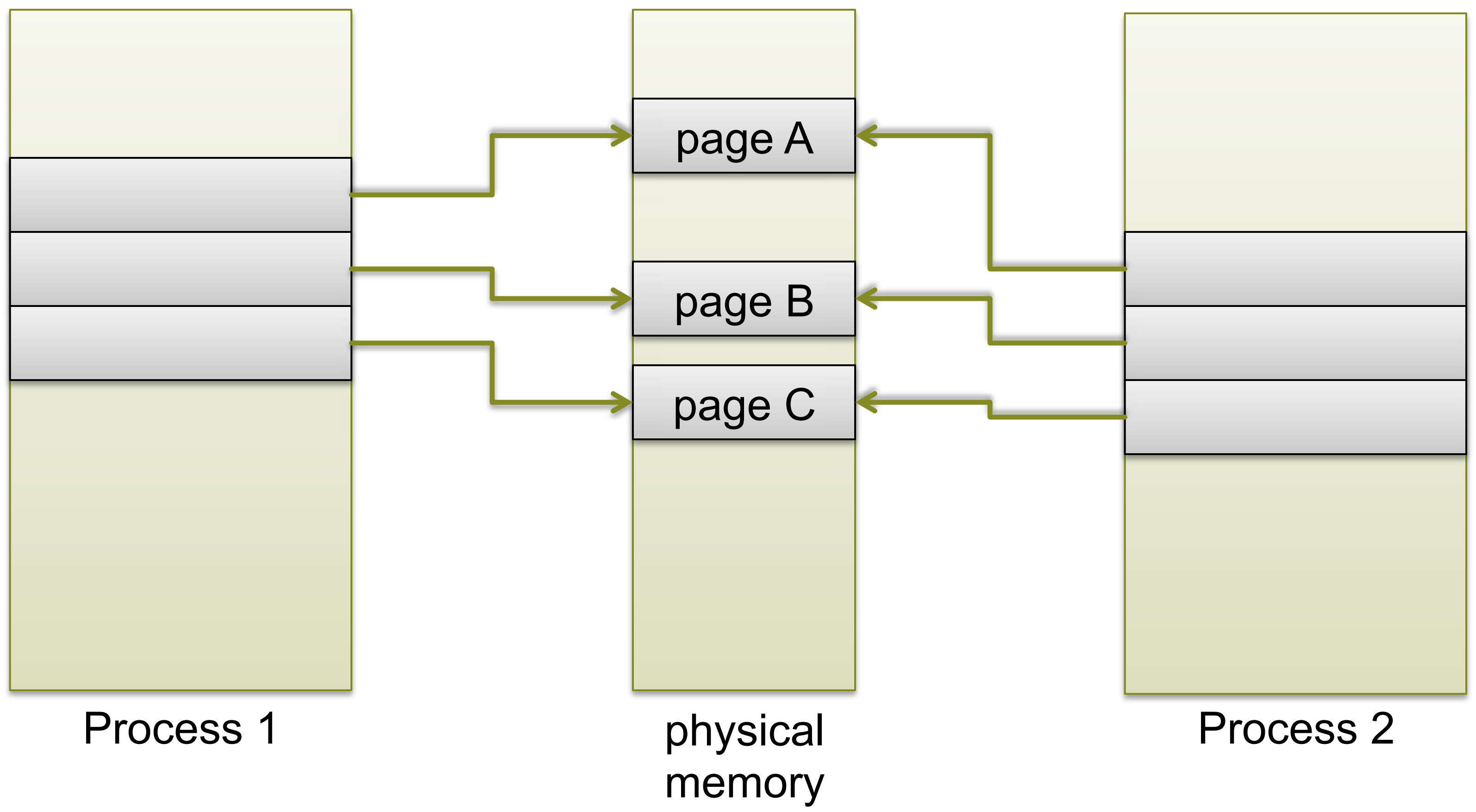
- COW allows both parent and child processes to initially *share* the same pages in memory

  If either process modifies a shared page, only then is the page copied

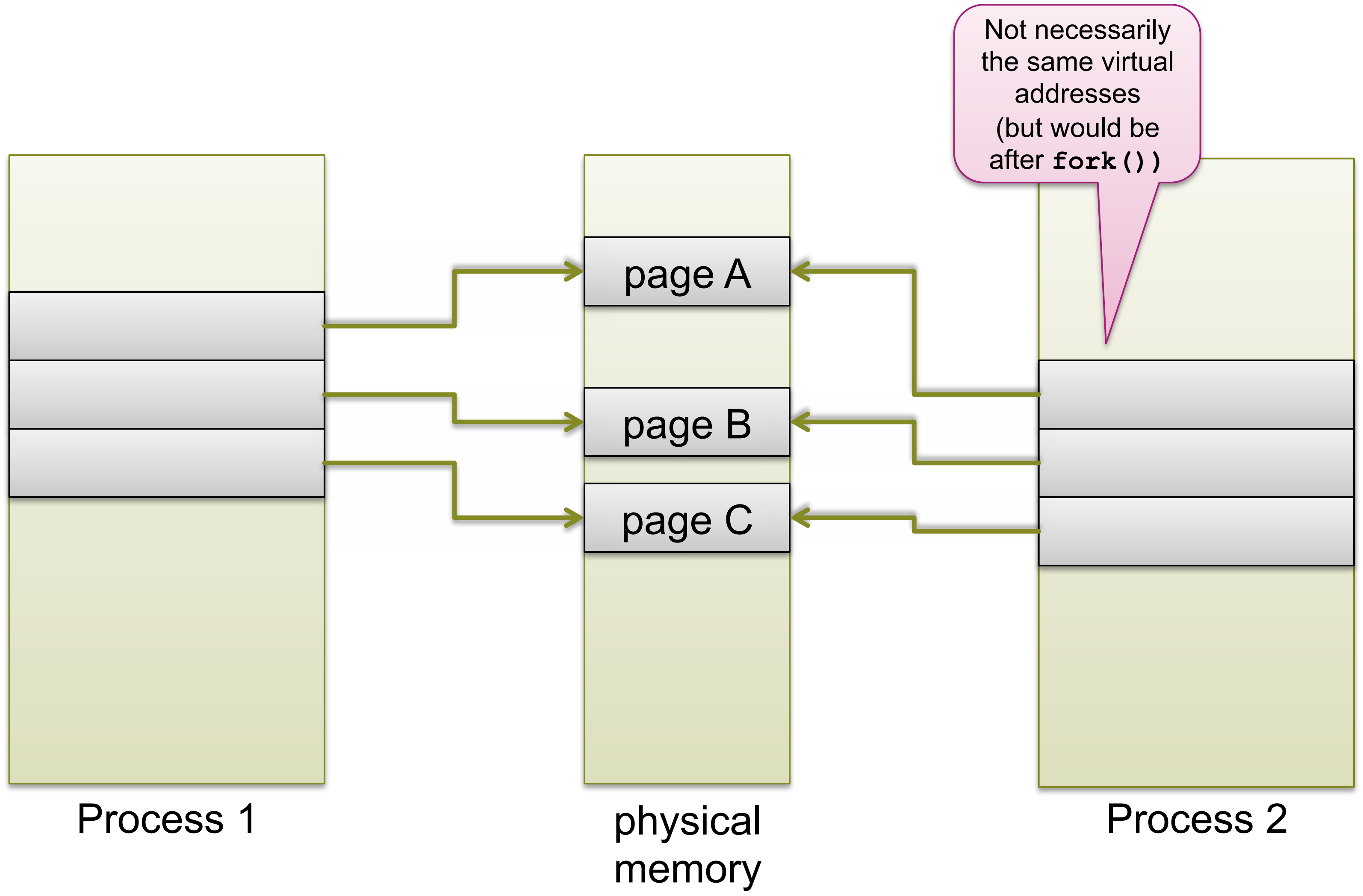
- COW allows more efficient process creation as only modified pages are copied

- Free pages are allocated from a pool of zeroed-out pages

Example: processes sharing an area of memory



Example: processes sharing an area of memory

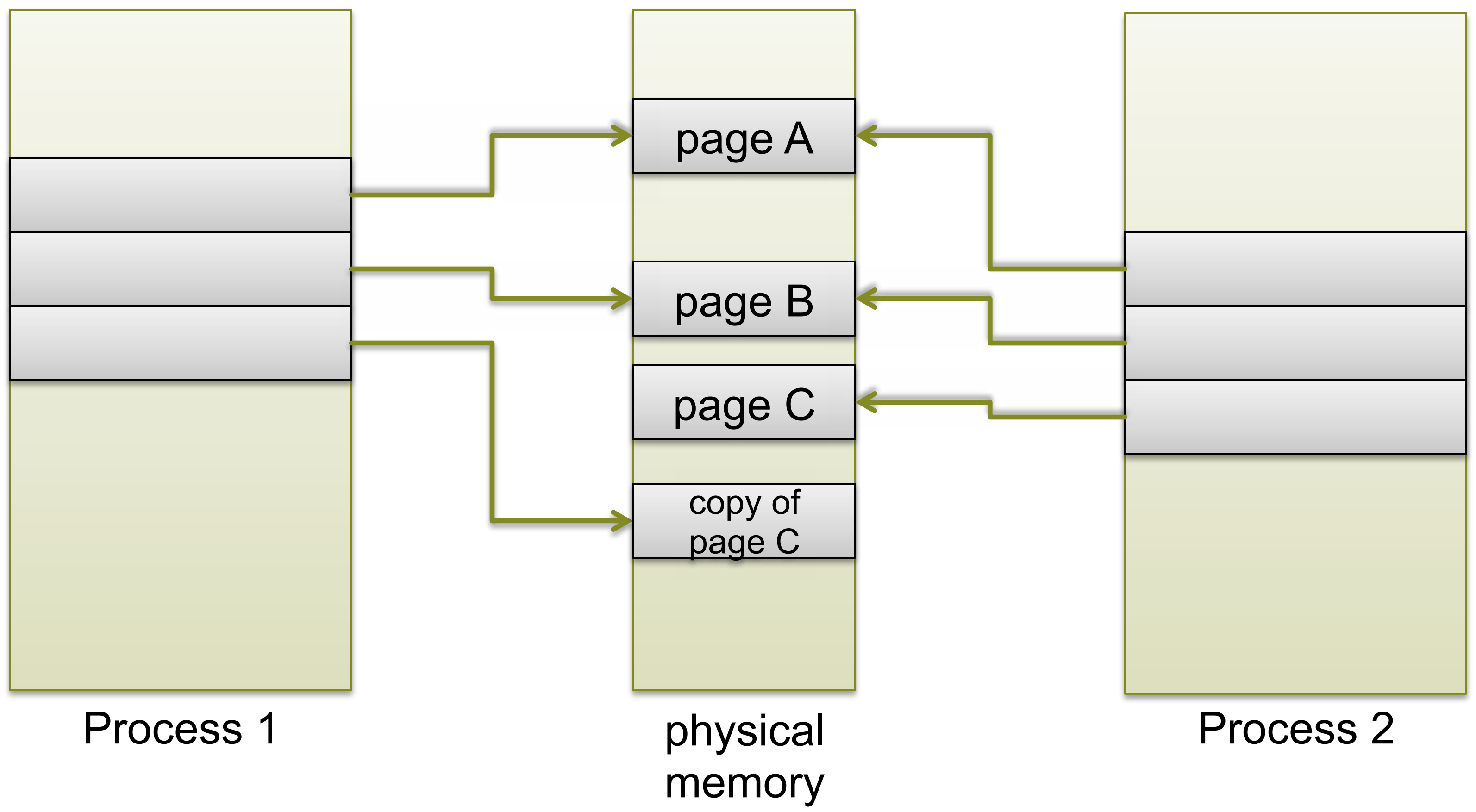

Not necessarily the same virtual addresses (but would be after `fork()`)
page A
page B
page C
Process 1
physical memory
Process 2

How does it work?

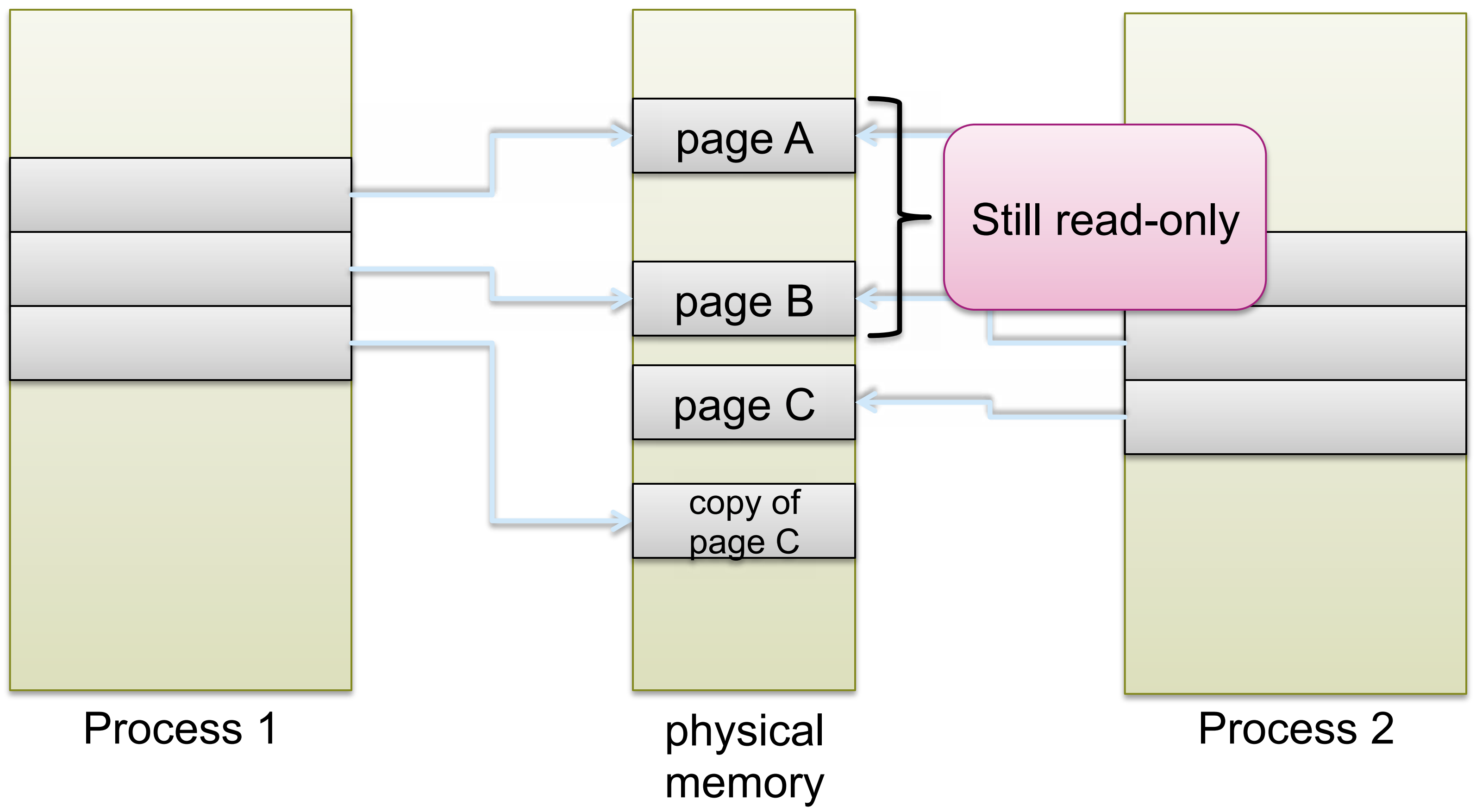
- **Initially mark all pages as read-only**
- **Either process writes ⇒ page fault**
  - Fault handler allocates new frame
  - Makes copy of page in new frame
  - Maps each copy into resp. processes writeable

- **Only modified pages are copied**
  - Less memory usage, more sharing
  - Cost is page fault for each mutated page

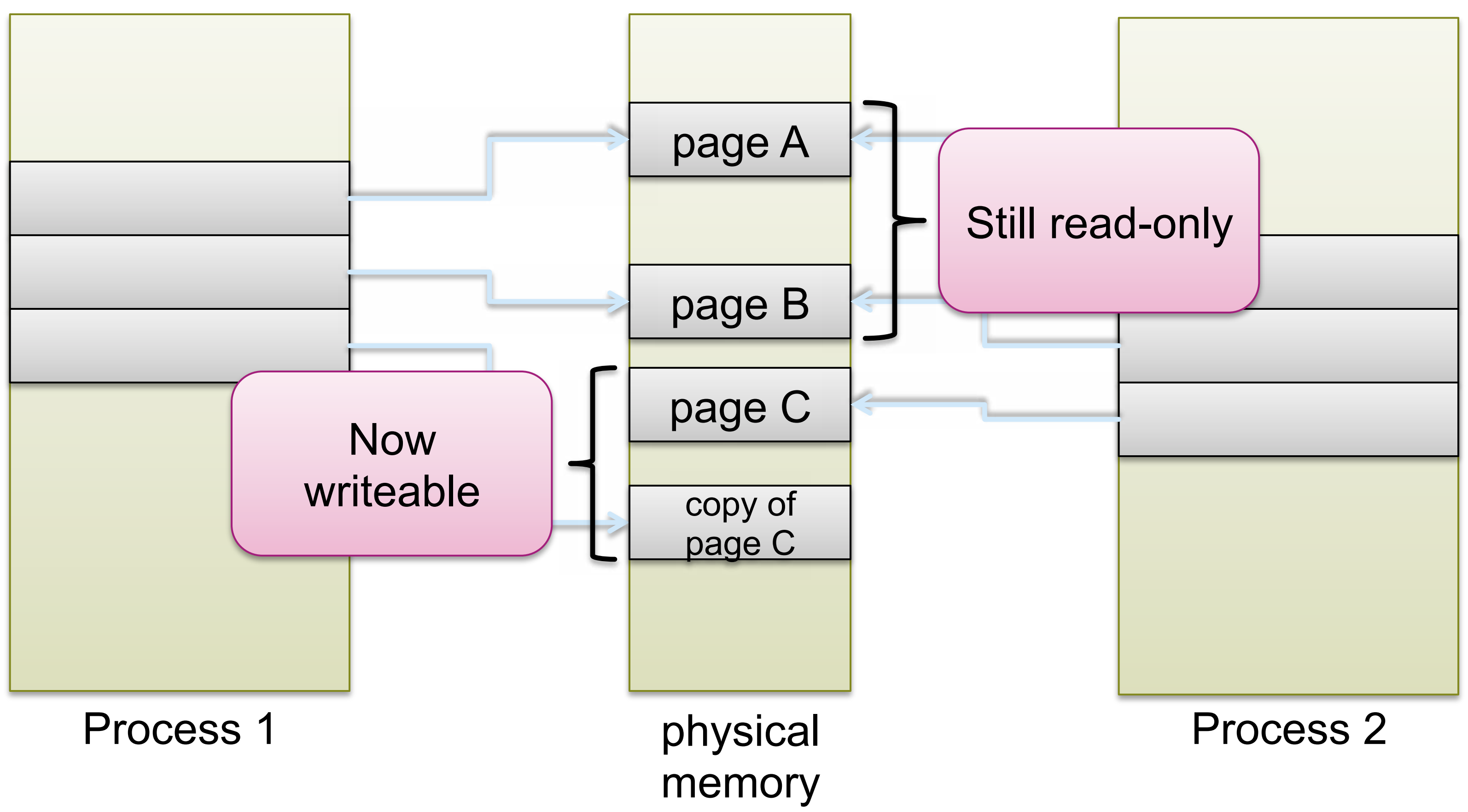
After process 1 writes to page C



After process 1 writes to page C



After process 1 writes to page C

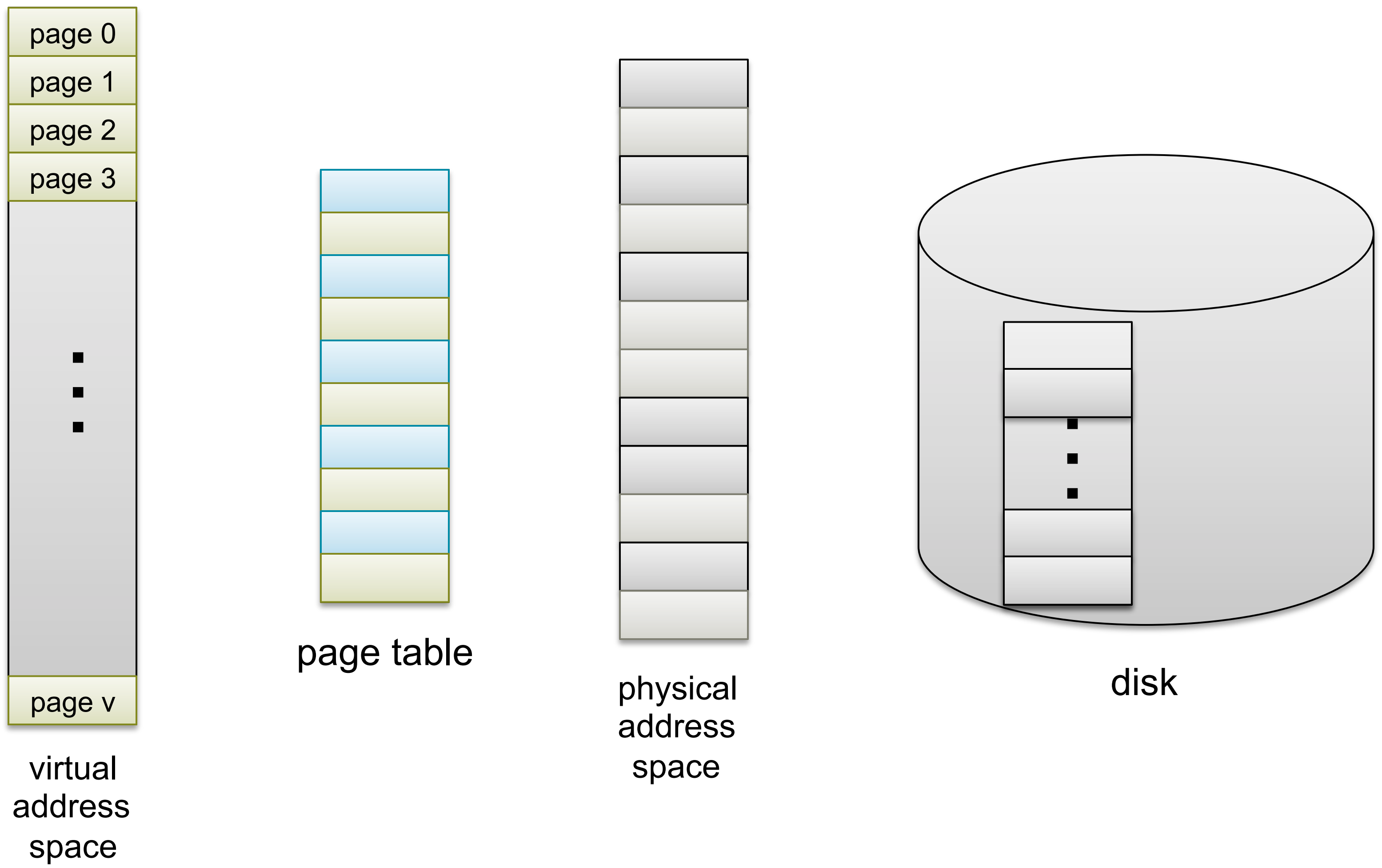


General principle

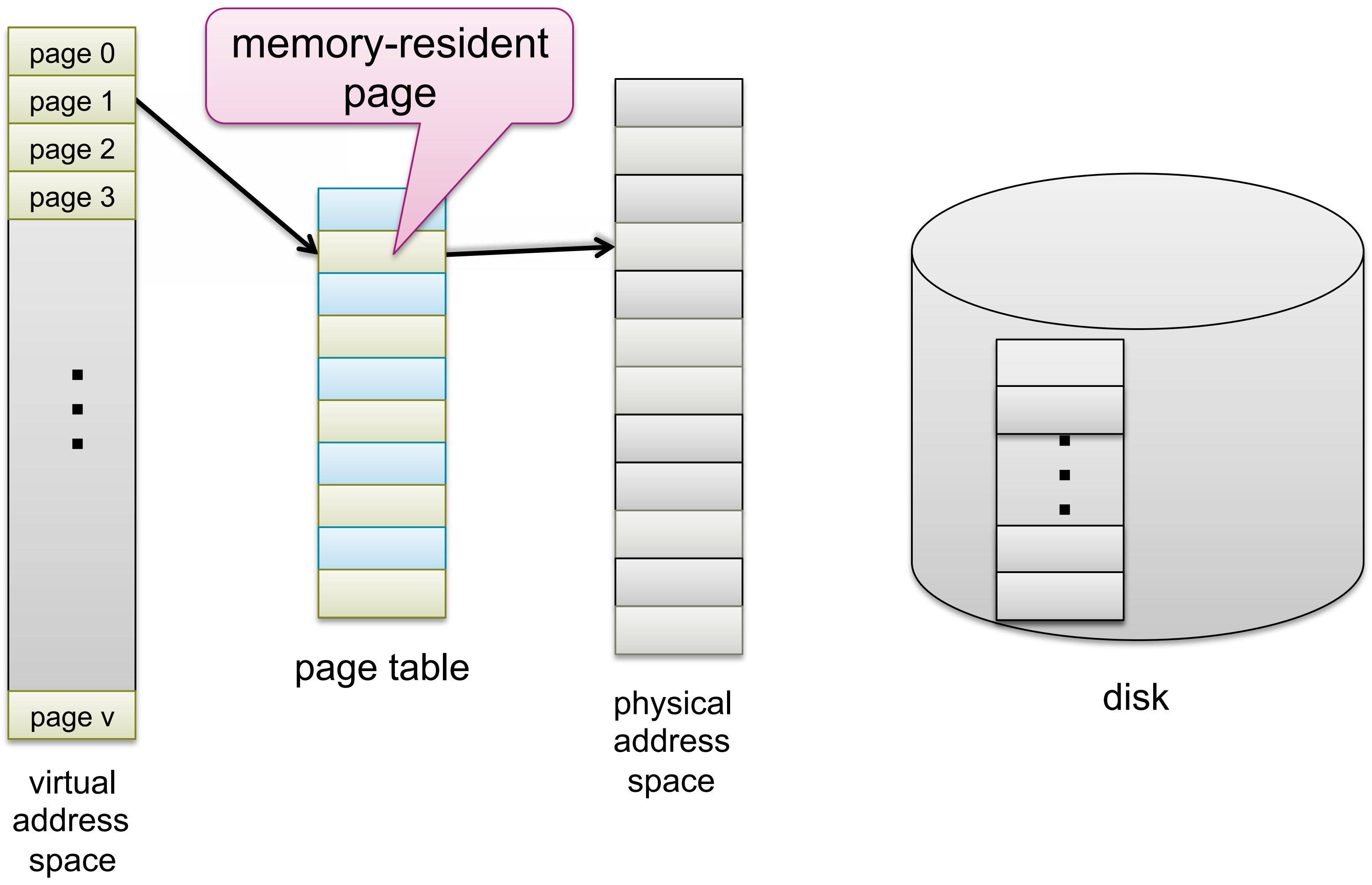
- **Mark a VPN as invalid or readonly**
  ⇒ trap indicates attempt to read or write
- **On a page fault, change mappings somehow**
- **Restart instruction, as if nothing had happened**

- **General: allows *emulation* of memory as well as *multiplexing.***
  - E.g. on-demand zero-filling of pages
  - And…

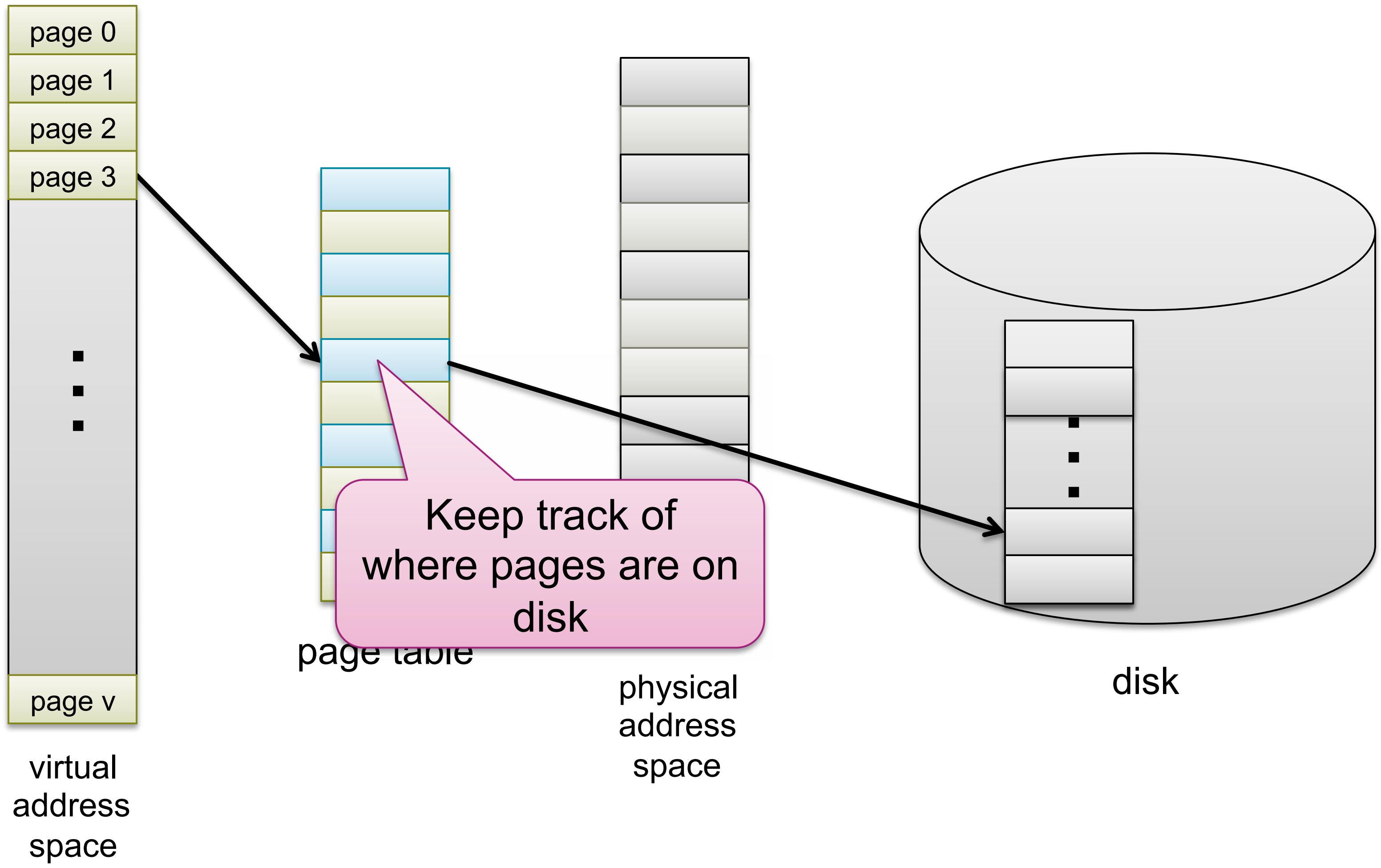
Paging concepts



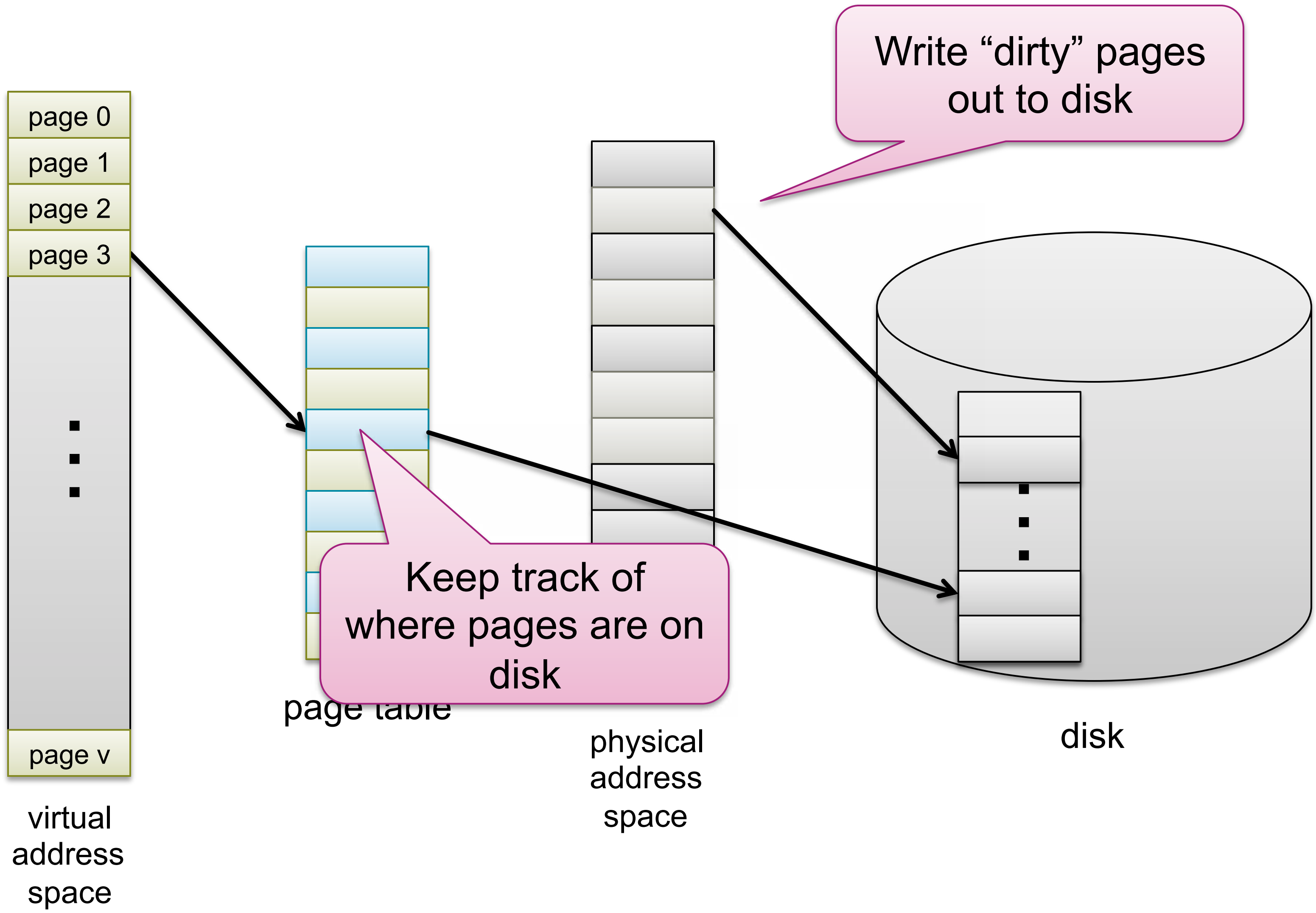
Paging concepts



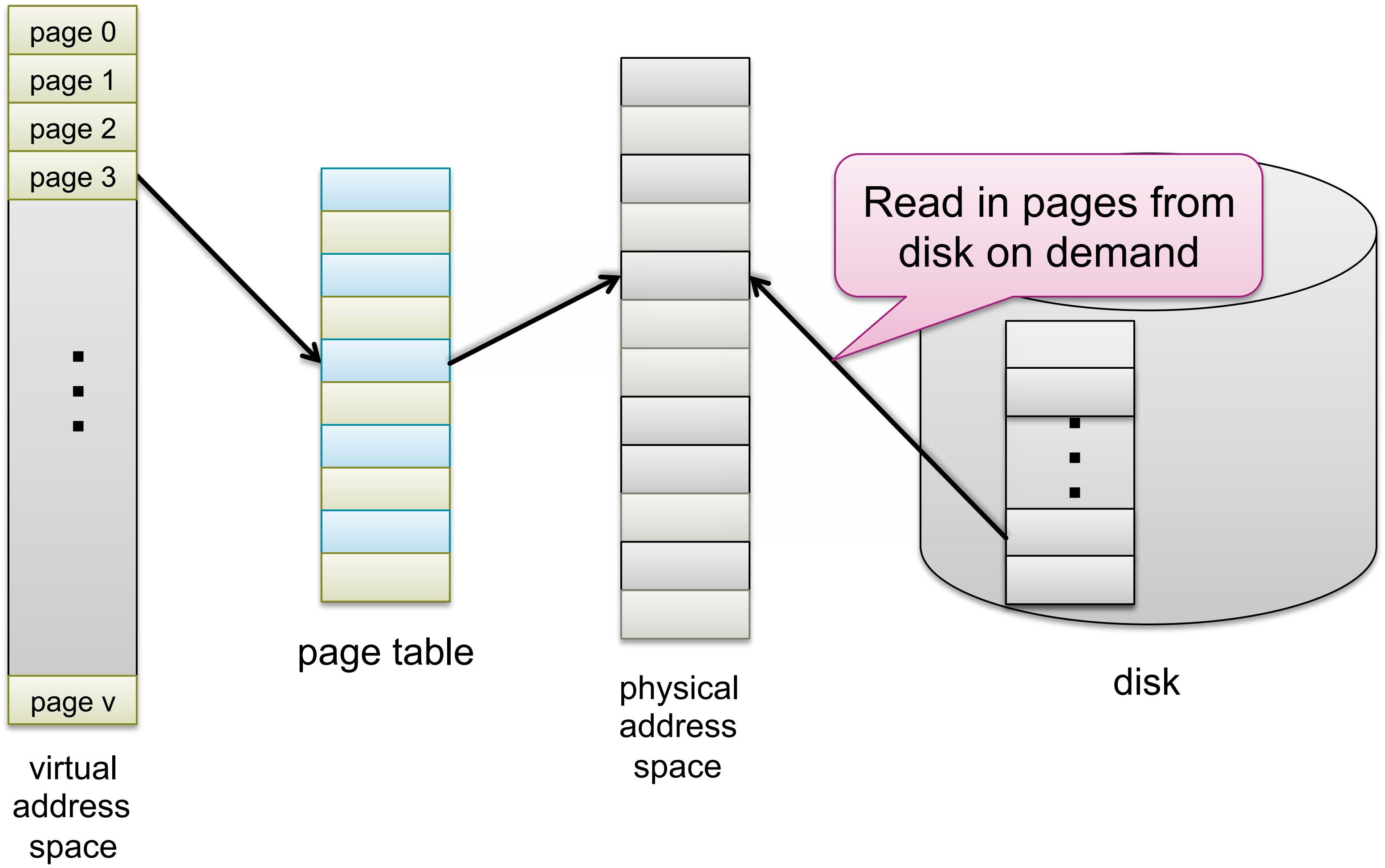
Paging concepts



Paging concepts



Paging concepts



Demand Paging

- **Bring a page into memory only when it is needed**
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users

- **Turns RAM into a *cache* for processes on *disk*!**

Demand Paging

- **Page needed ⇒ reference (load or store) to it**
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory

- ***Lazy swapper* – never swaps a page into memory unless page will be needed**
  - Swapper that deals with pages is a pager
  - Can do this with segments, but more complex

- ***Strict demand paging*: only page in when referenced**

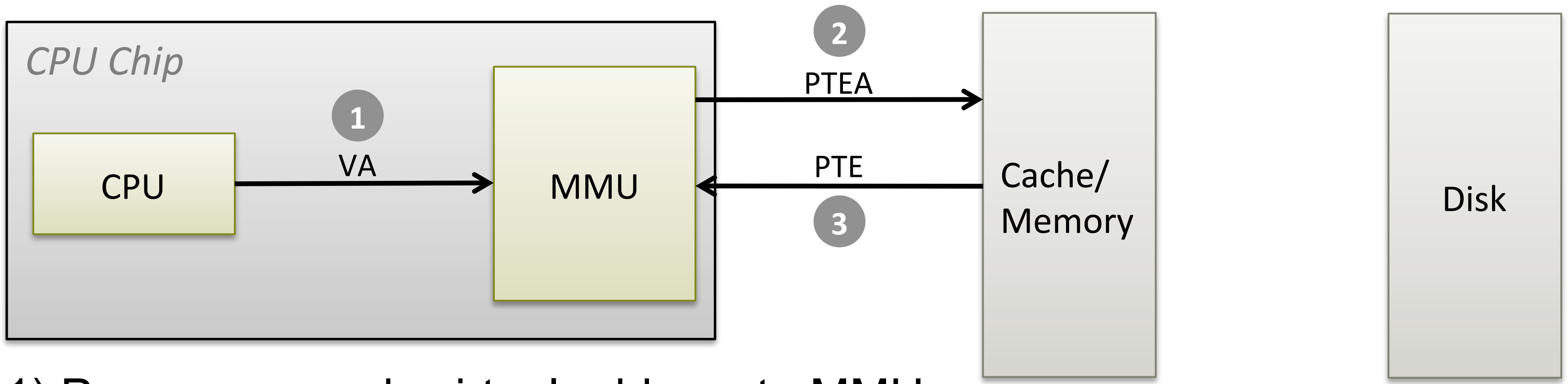
Page Fault

- **If there is a reference to a page, first reference to that page will trap to operating system:**

  page fault

1. **Operating system looks at another table to decide:**
   - Invalid reference ⇒ abort
   - Just not in memory
2. **Get empty frame**
3. **Swap page into frame**
4. **Reset tables**
5. **Set valid bit v**
6. **Restart the instruction that caused the page fault**

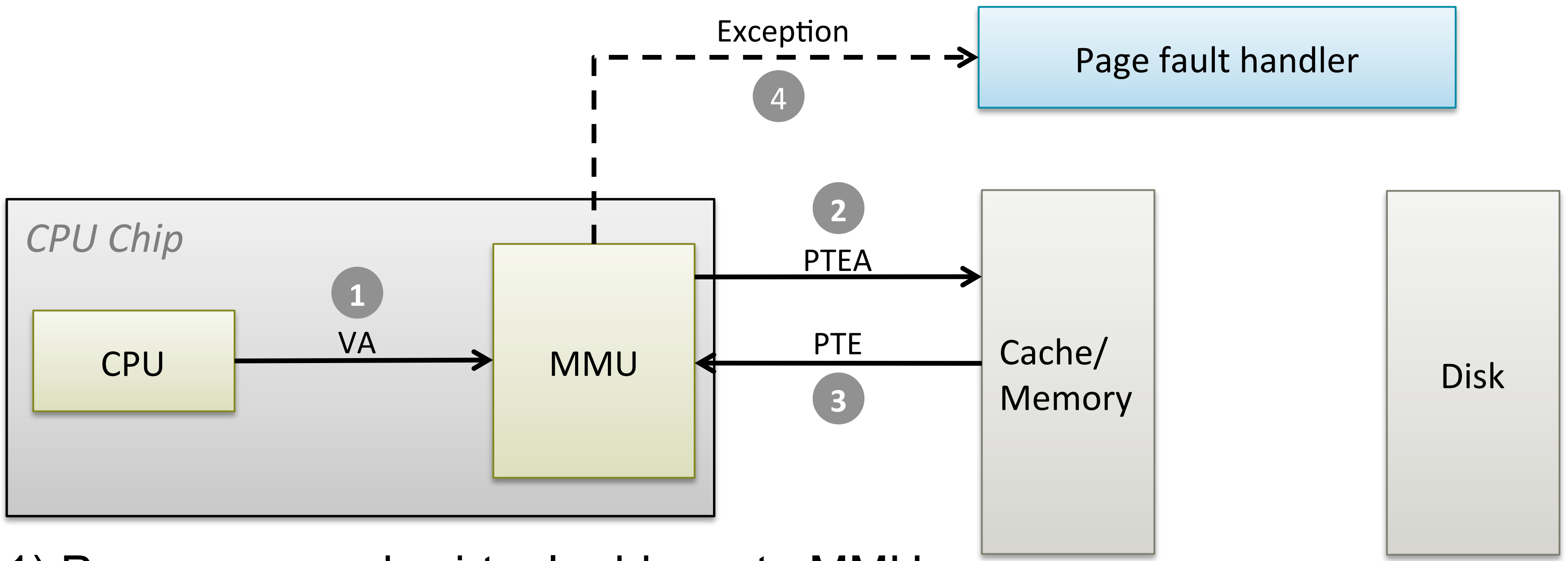
Recall: handling a page fault



1) Processor sends virtual address to MMU

2-3) MMU fetches PTE from page table in memory

Recall: handling a page fault

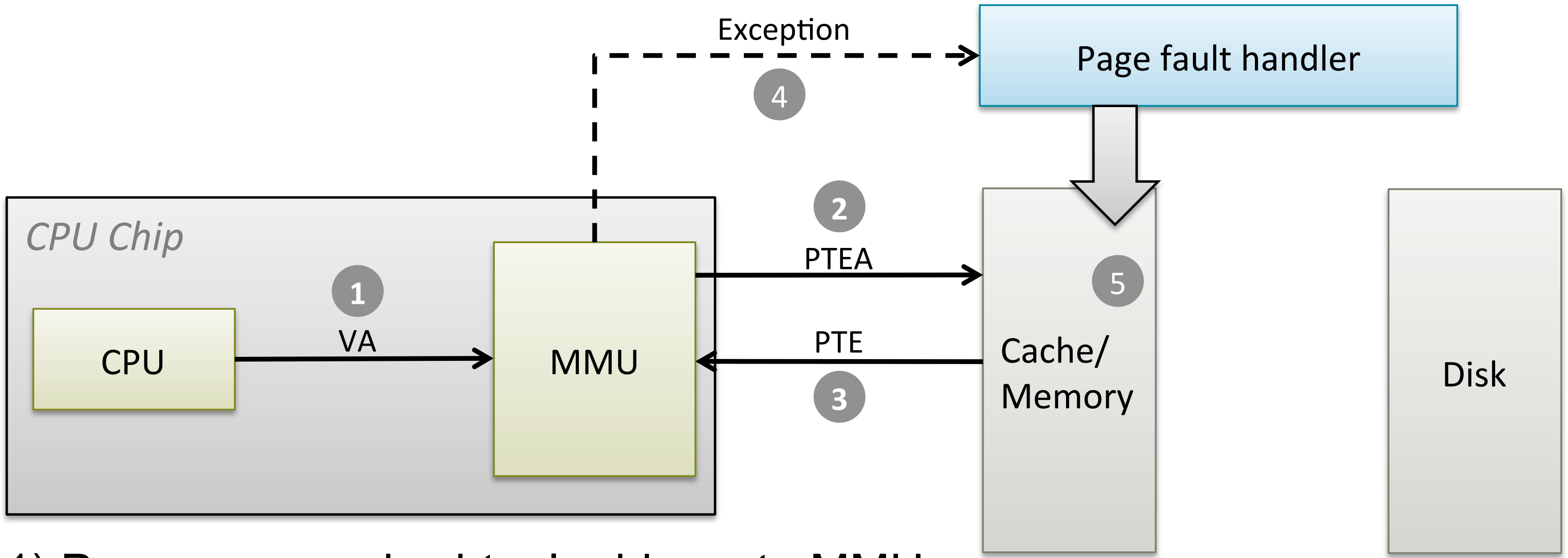


1) Processor sends virtual address to MMU

2-3) MMU fetches PTE from page table in memory

4) Valid bit is zero, so MMU triggers page fault exception

Recall: handling a page fault



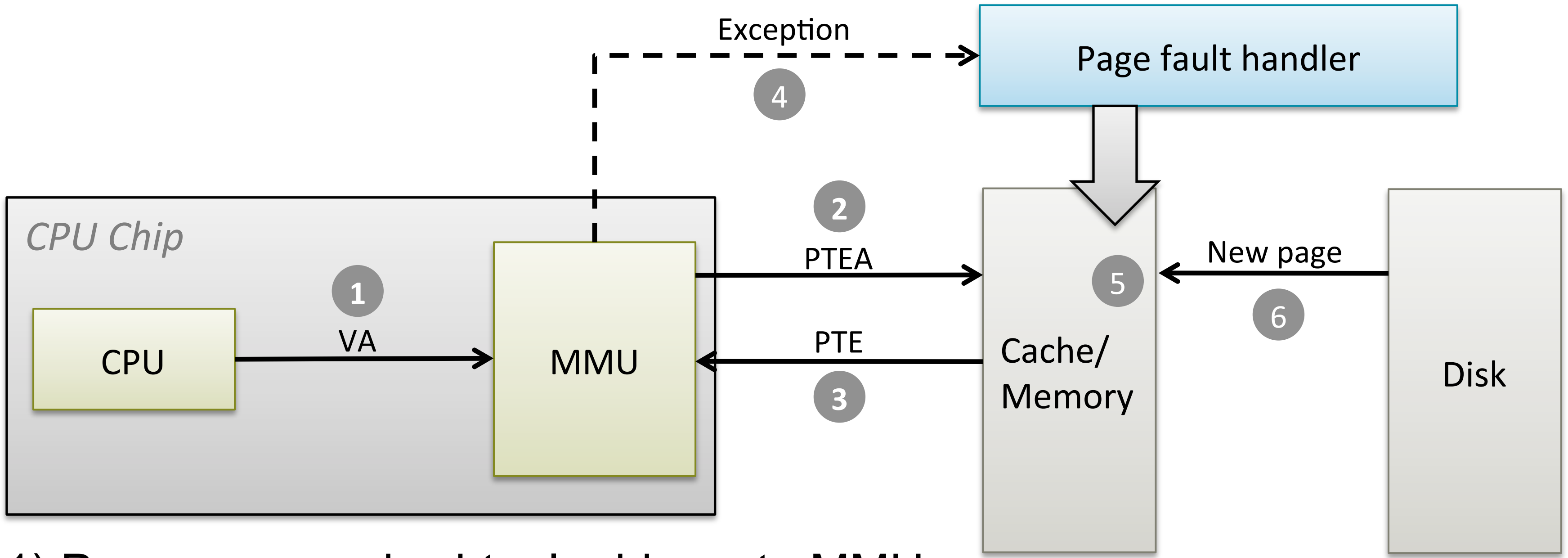
1) Processor sends virtual address to MMU

2-3) MMU fetches PTE from page table in memory

4) Valid bit is zero, so MMU triggers page fault exception

5) Handler finds a frame to use for missing page

Recall: handling a page fault



1) Processor sends virtual address to MMU

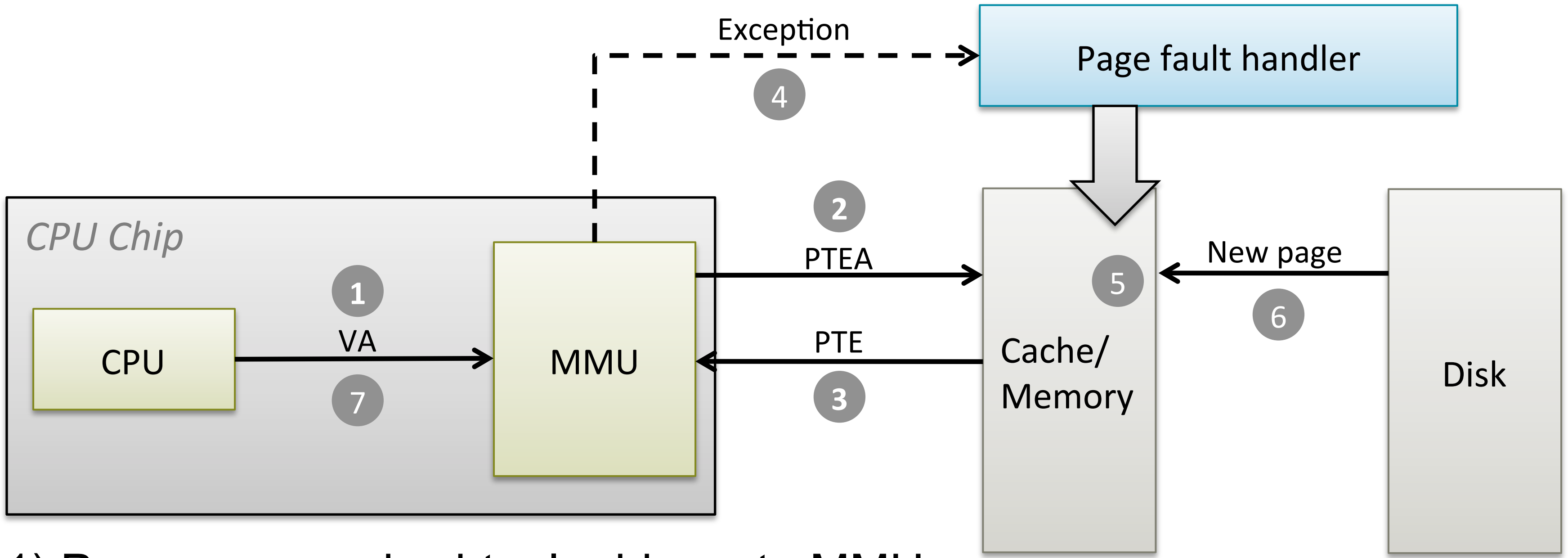
2-3) MMU fetches PTE from page table in memory

4) Valid bit is zero, so MMU triggers page fault exception

5) Handler finds a frame to use for missing page

6) Handler pages in new page and updates PTE in memory

Recall: handling a page fault



1) Processor sends virtual address to MMU

2-3) MMU fetches PTE from page table in memory

4) Valid bit is zero, so MMU triggers page fault exception

5) Handler finds a frame to use for missing page

6) Handler pages in new page and updates PTE in memory

7) Handler returns to original process, restarting faulting instruction

Performance of demand paging

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

- **Effective Access Time (EAT)**

$$\begin{aligned}\text{EAT} = (1 - p) \times \text{memory access} \\ + p\,(\text{page fault overhead} \\ + \text{swap page out} \\ + \text{swap page in} \\ + \text{restart overhead} \\ )\end{aligned}$$

Demand paging example

- Memory access time = 200 nanoseconds

- Average page-fault service time = 8 milliseconds

- EAT = (1 – p) x 200 + p (8 milliseconds)
  = (1 – p) x 200 + p x 8,000,000
  = 200 + p x 7,999,800

- 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 is no free frame?

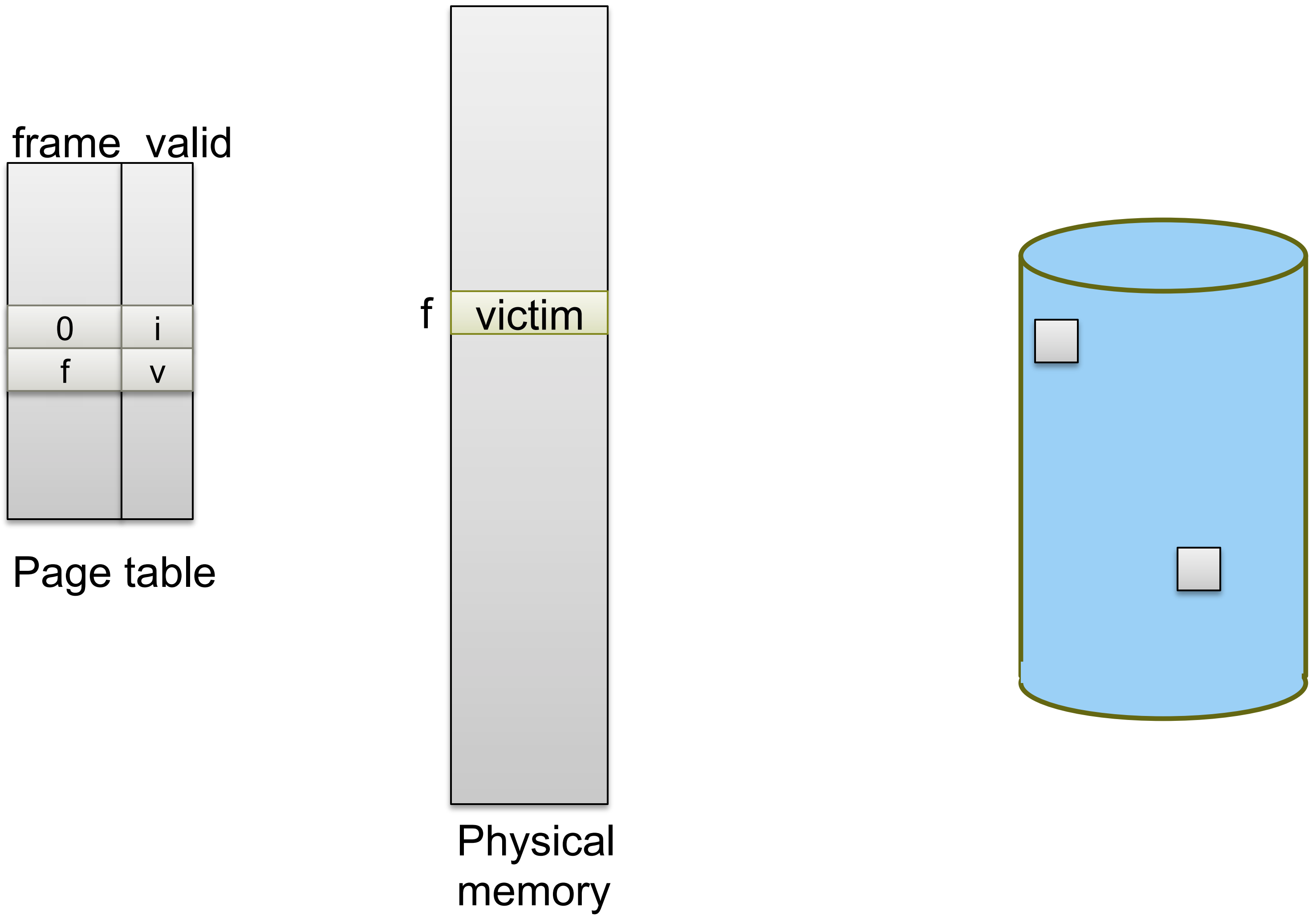
- ***Page replacement*** **– find "little used" resident page to discard or write to disk**
  - "victim page"
  - needs selection algorithm
  - performance – want an algorithm which will result in minimum number of page faults

- **Same page may be brought into memory several times**

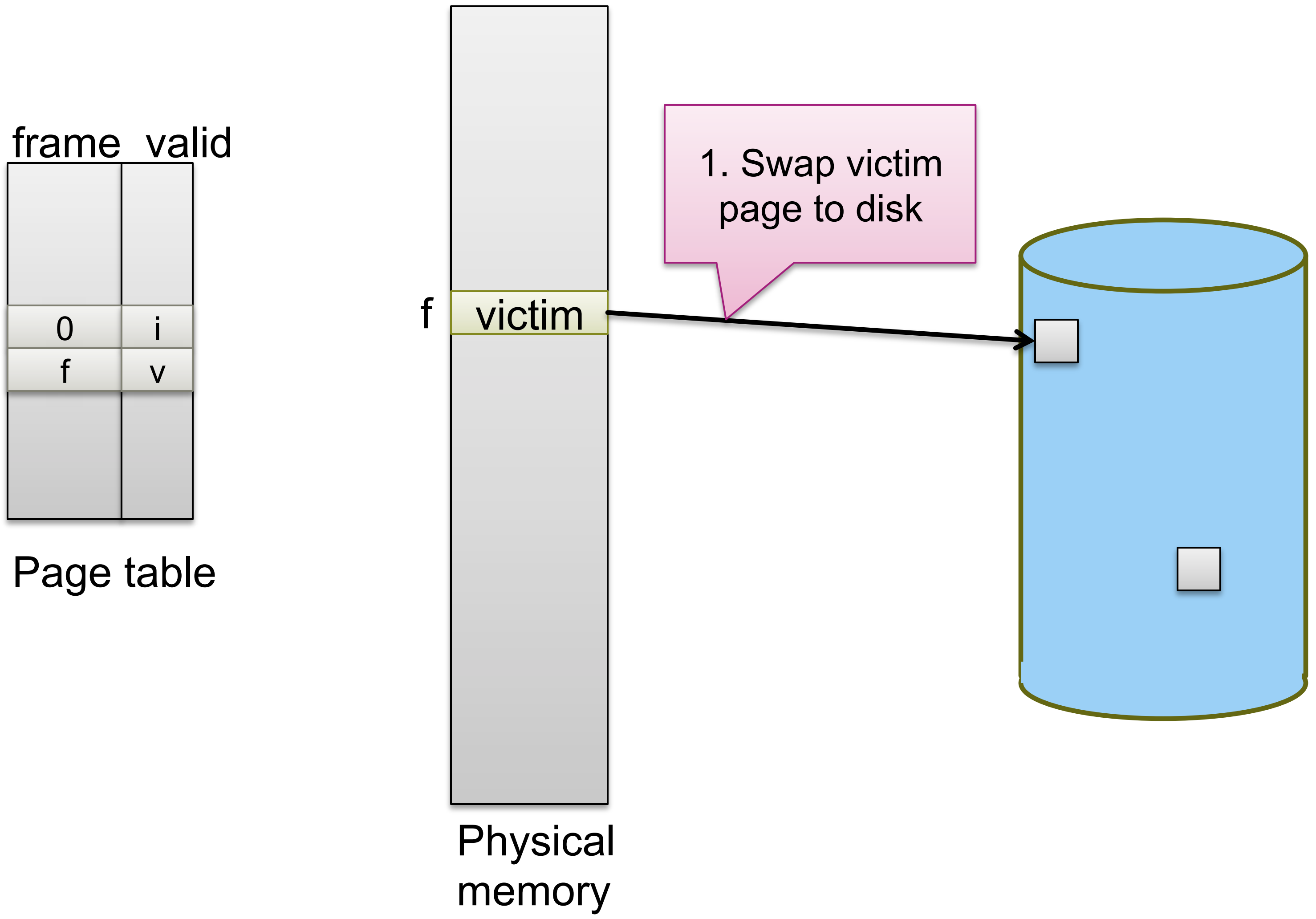
Page replacement

- **Try to pick a victim page which won’t be referenced in the future**
  - Various heuristics – but ultimately it’s a guess
- **Use “modify” bit on PTE**
  - Don’t write “clean” (unmodified) page to disk
  - Try to pick “clean” pages over “dirty” ones
    (save a disk write)

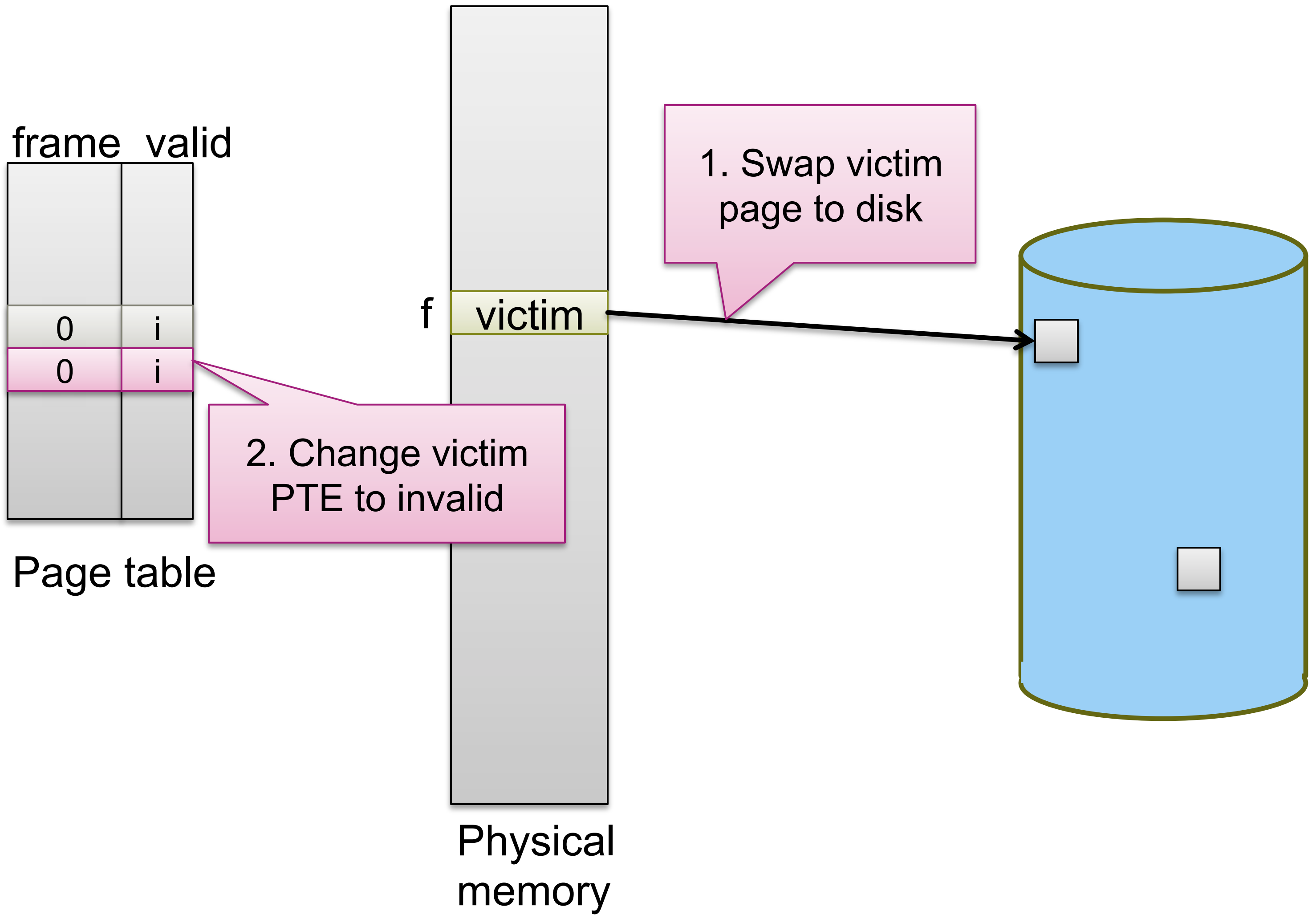
Page replacement



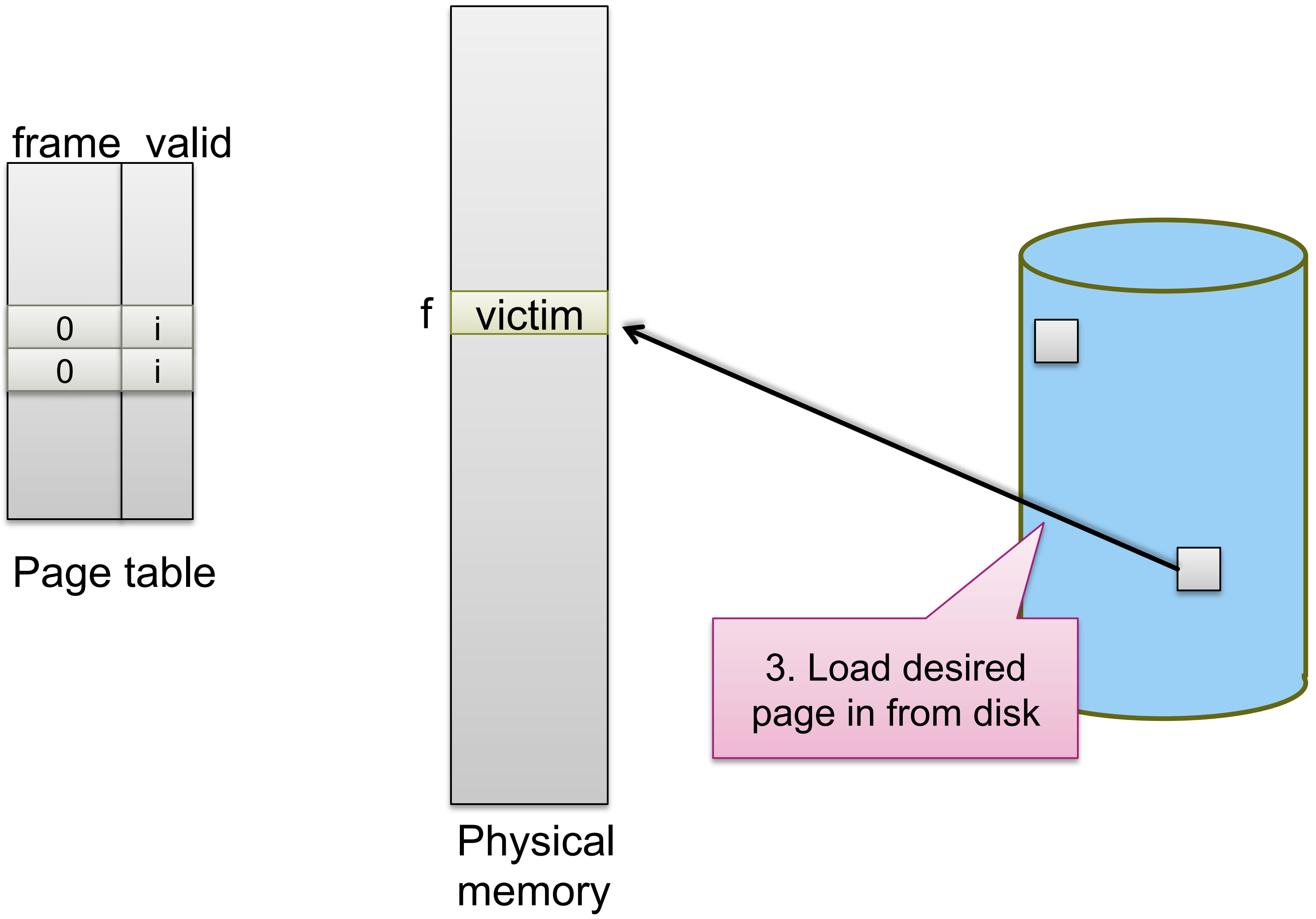
Page replacement



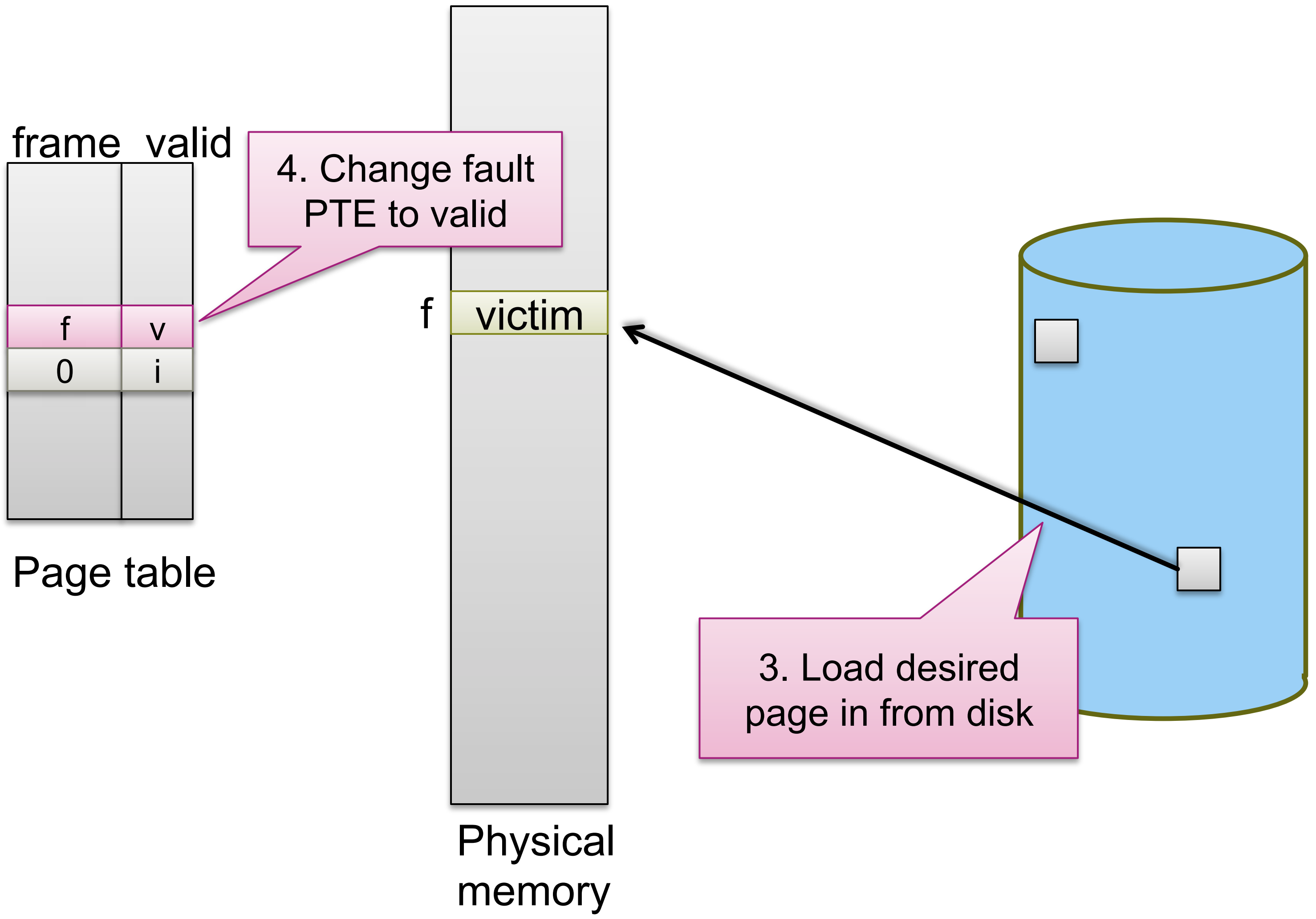
Page replacement



Page replacement



Page replacement

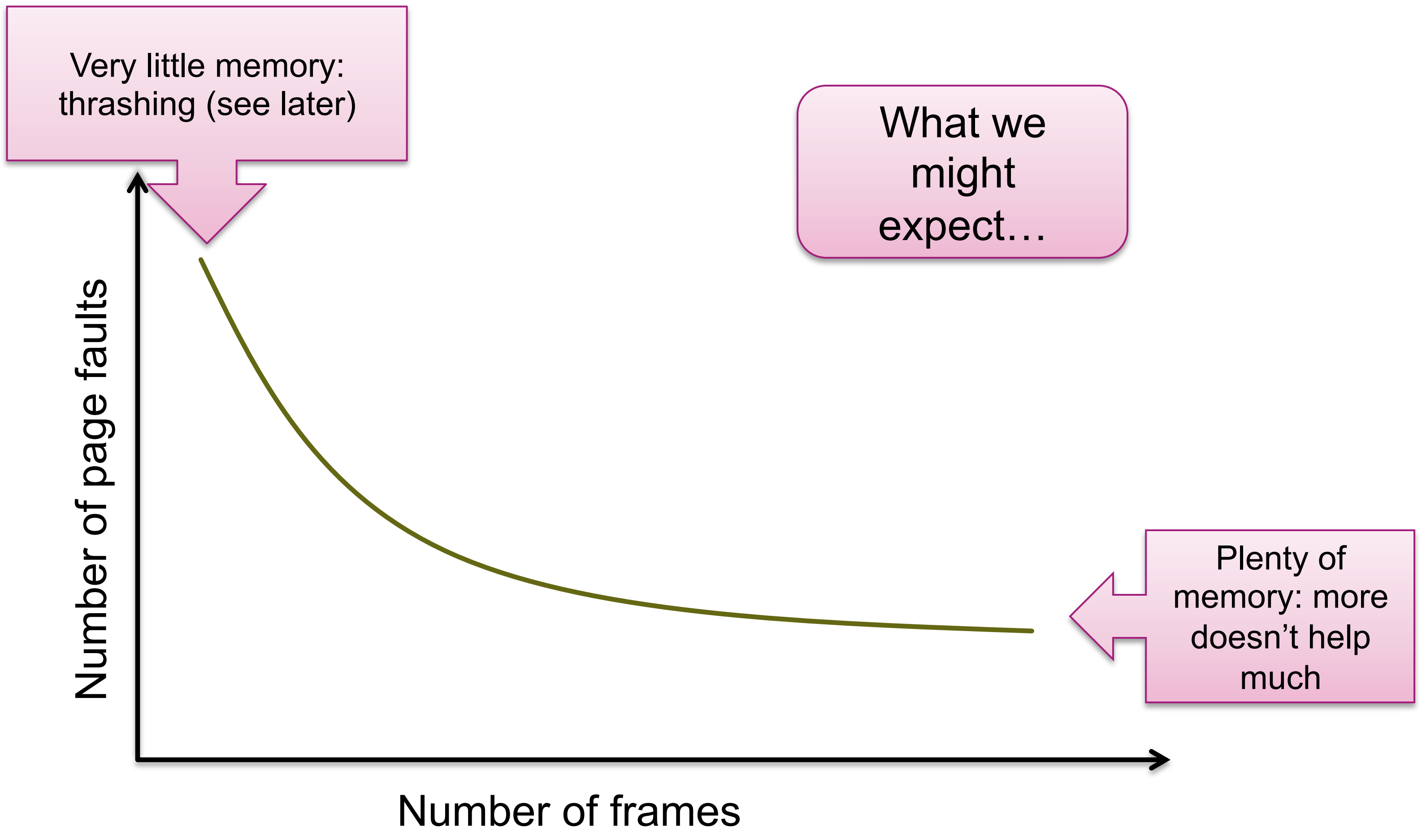

frame
valid
f
v
0
i
Page table
4. Change fault PTE to valid
f
victim
Physical memory
3. Load desired page in from disk

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
- E.g.

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

Page faults vs. number of frames



FIFO (First-In-First-Out) page replacement

reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

page frames:

FIFO (First-In-First-Out) page replacement

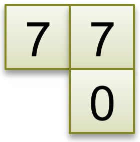
reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

page frames: $\boxed{7}$

FIFO (First-In-First-Out) page replacement

reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

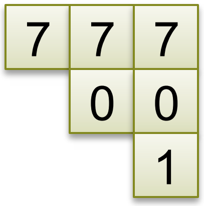
page frames:



FIFO (First-In-First-Out) page replacement

reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

page frames:



FIFO (First-In-First-Out) page replacement

reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

page frames:

7	7	7	2
	0	0	0
		1	1

FIFO (First-In-First-Out) page replacement

reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

reference string:	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
page frames:	7	7	7	2		2														
		0	0	0		3														
			1	1		1														

FIFO (First-In-First-Out) page replacement

reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

page frames:

7	7	7	2
	0	0	0
		1	1

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

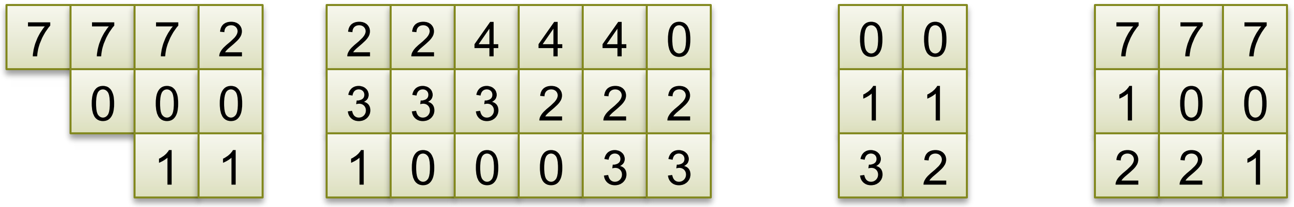
FIFO (First-In-First-Out) page replacement

reference string:	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
page frames:	7	7	7	2		2	2	4	4	4	0			0	0					
		0	0	0		3	3	3	2	2	2			1	1					
			1	1		1	0	0	0	3	3			3	2					

FIFO (First-In-First-Out) page replacement

reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

page frames:



Here, 15 page faults.

More memory is better?

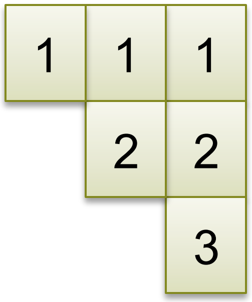
Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

-

More memory is better?

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

- **3 frames (3 pages can be in memory):**

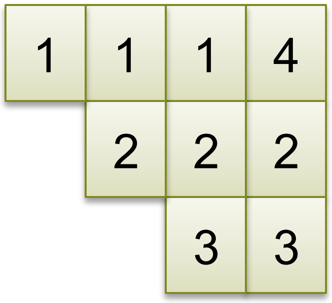


-

More memory is better?

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

- **3 frames (3 pages can be in memory):**



-

More memory is better?

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

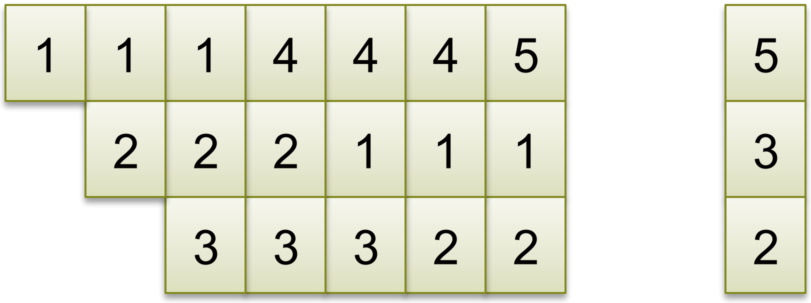
- **3 frames (3 pages can be in memory):**

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

More memory is better?

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

- **3 frames (3 pages can be in memory):**



-

More memory is better?

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

- **3 frames (3 pages can be in memory):**

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

9 page faults

-

More memory is better?

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

- **3 frames (3 pages can be in memory):**

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

5	5
3	3
2	4

9 page faults

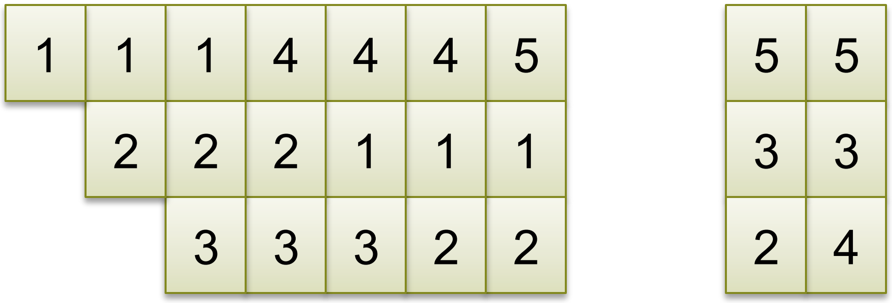
- **4 frames:**

1	1	1	1
	2	2	2
		3	3
			4

More memory is better?

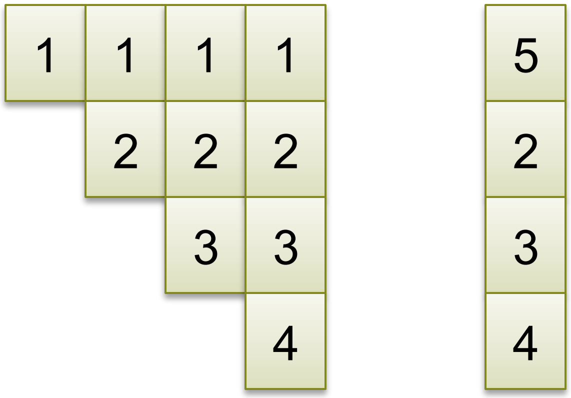
Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- **3 frames (3 pages can be in memory):**



9 page faults

- **4 frames:**



More memory is better?

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

- **3 frames (3 pages can be in memory):**

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

9 page faults

- **4 frames:**

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

10 page faults!

More memory is better?

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

- **3 frames (3 pages can be in memory):**

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

5	5
3	3
2	4

9 page faults

- **4 frames:**

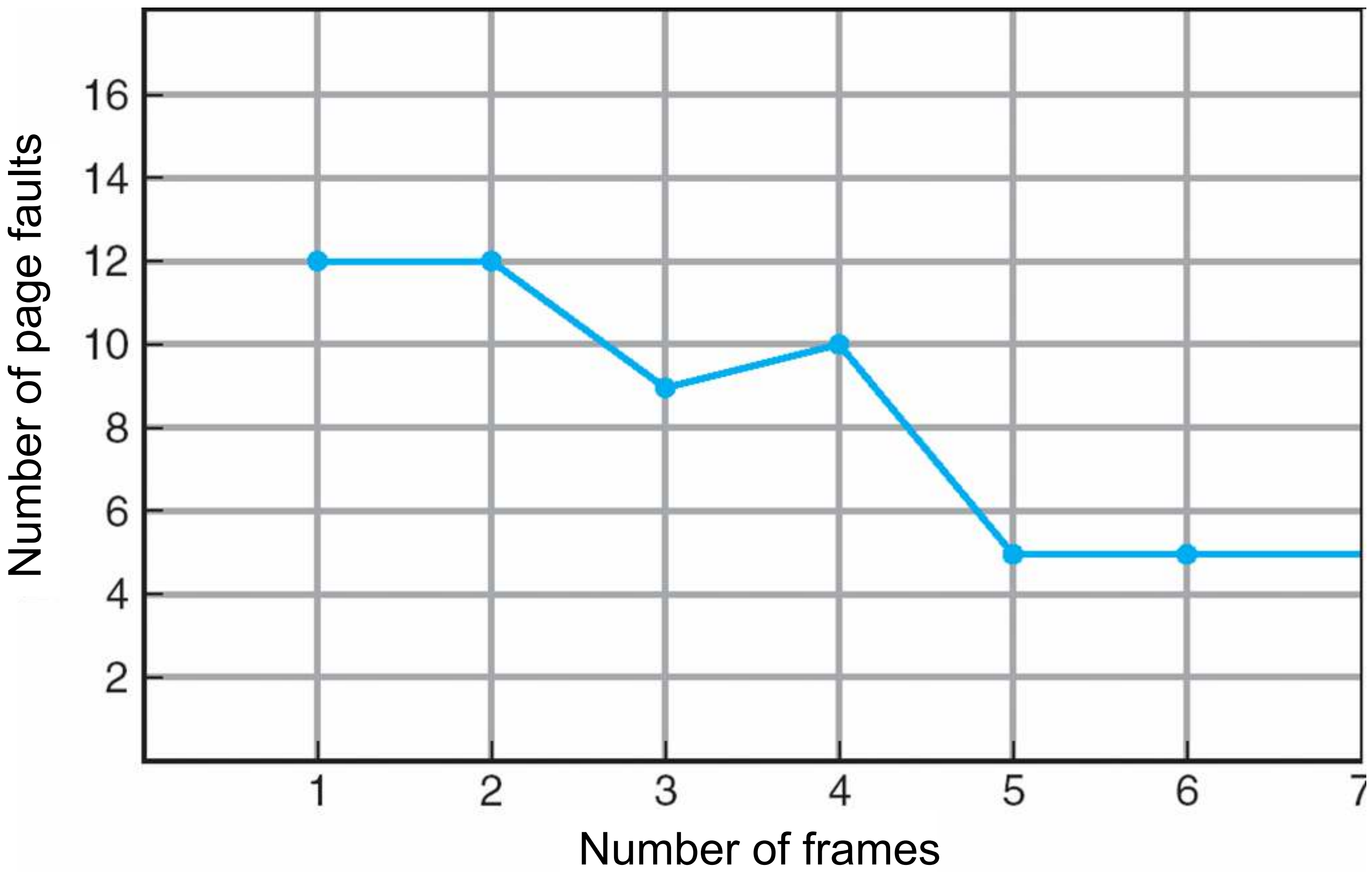
1	1	1	1
	2	2	2
		3	3
			4

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

10 page faults!

Belady's Anomaly: more frames ⇒ more page faults

FIFO showing Belady's Anomaly

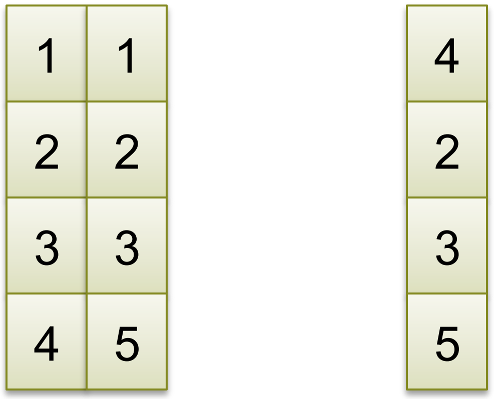


Optimal algorithm

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

4 frames example:

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



⇒ 6 page faults

How do you know this? – you can't!

Used for measuring how well your algorithm performs

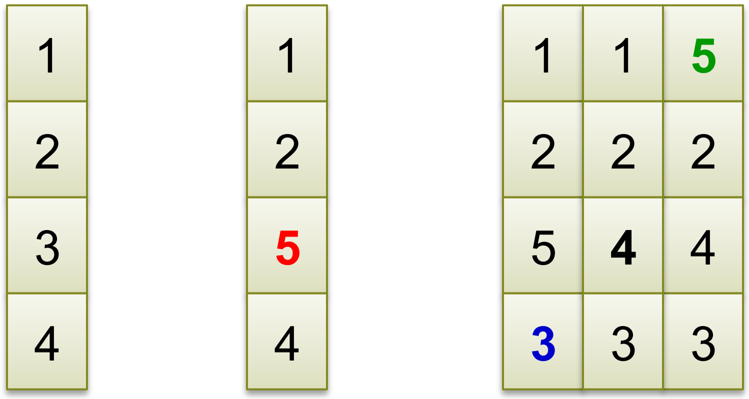
Optimal page replacement

reference string:	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
page frames:	7	7	7	2		2		2			2			2				7		
		0	0	0		0		4			0			0				0		
			1	1		3		3			3			1				1		

Here, 9 page faults.

Least Recently Used (LRU) algorithm

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



- **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

LRU page replacement

reference string:	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
page frames:	7	7	7	2		2		4	4	4	0			1		1		1		
		0	0	0		0		0	0	3	3			3		0		0		
			1	1		3		3	2	2	2			2		2		7		

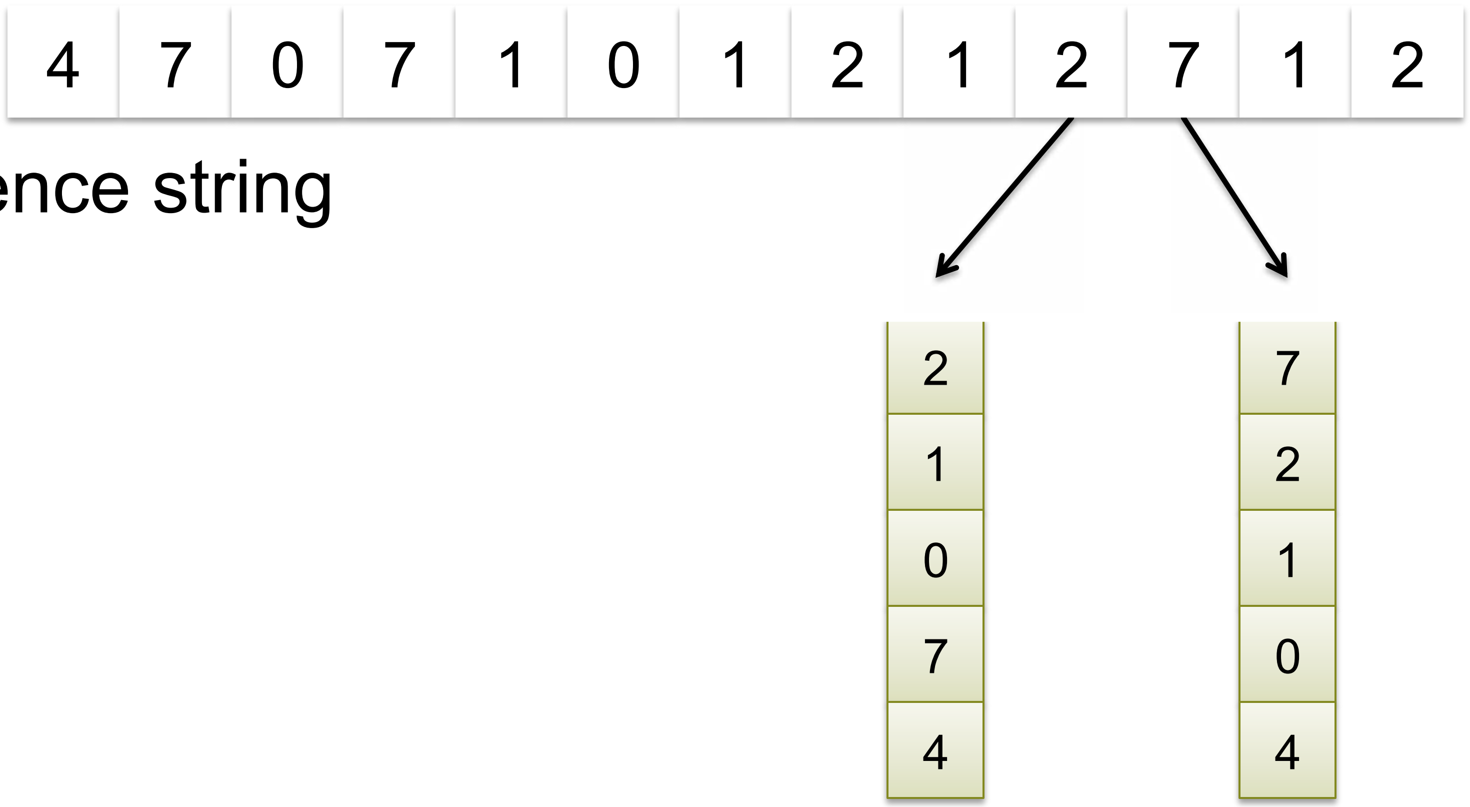
Here, 12 page faults.

LRU algorithm

- **Stack implementation – keep a stack of page numbers in a double link form:**
  - Page referenced:
    move it to the top
    requires 6 pointers to be changed
  - No search for replacement

- **General term: *stack algorithms***
  - Have property that adding frames always reduces page faults (no Belady's Anomaly)

Use a stack to record most recent page references

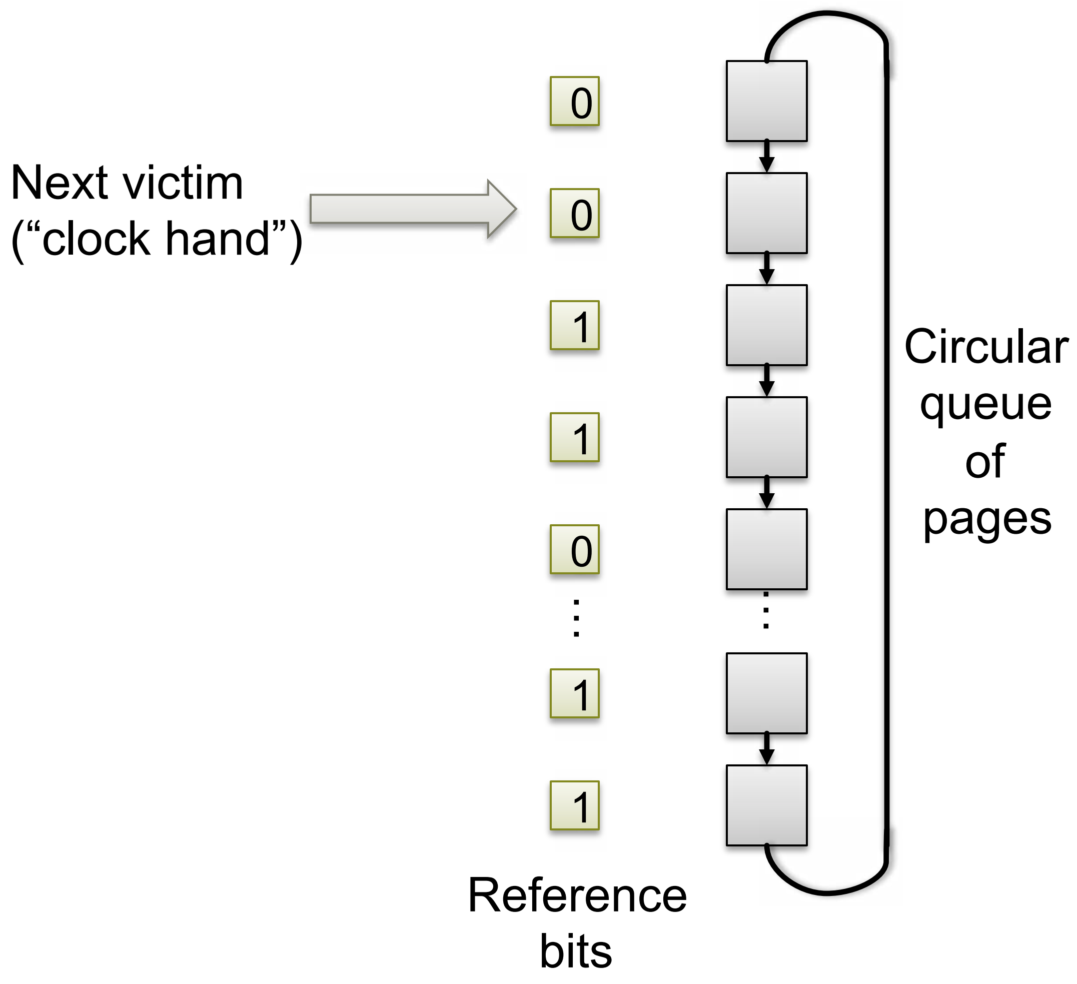


Reference string

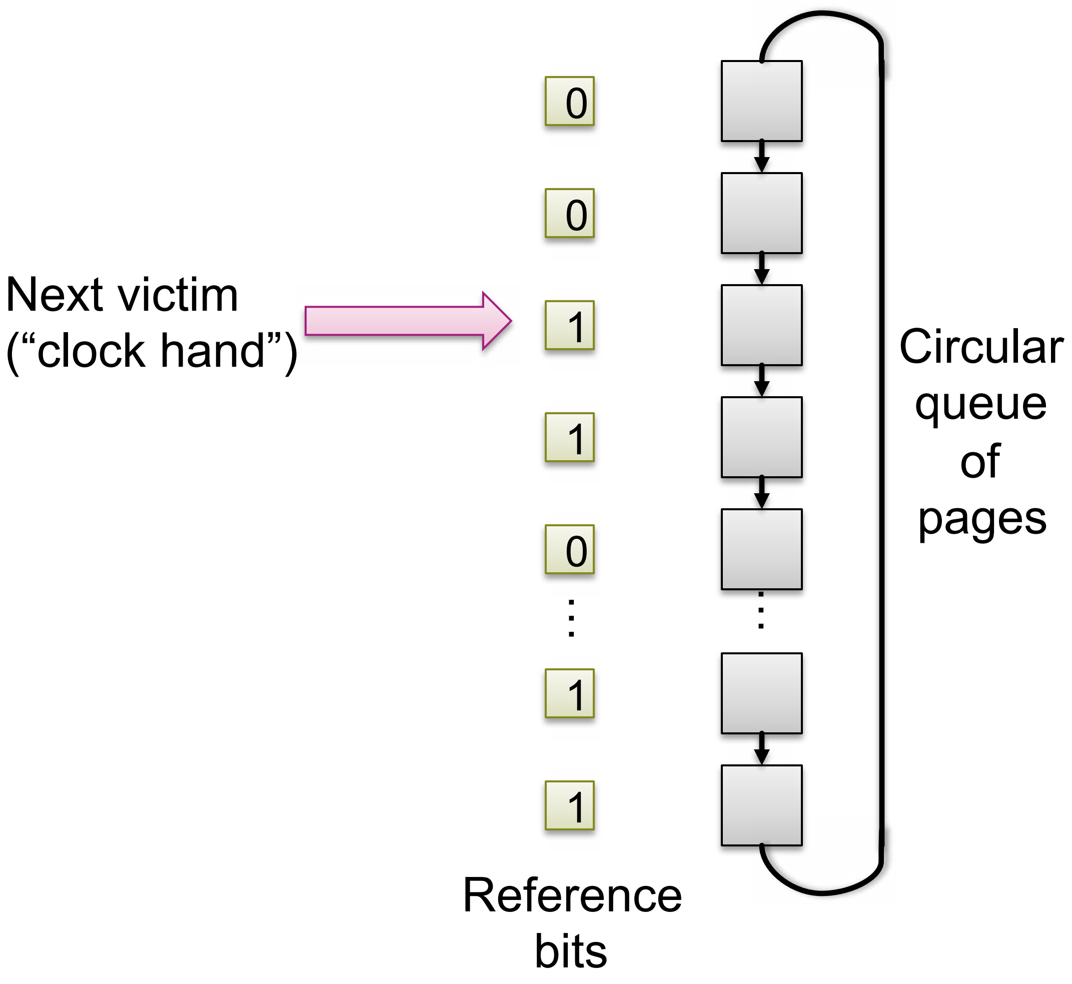
LRU approximation algorithms

- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace a page which is 0 (if one exists)
    We do not know the order, however
- **Second chance**
  - Need reference bit
  - Clock replacement
  - If page to be replaced (in clock order) has reference bit = 1 then:
    set reference bit 0
    leave page in memory
    replace next page (in clock order), subject to same rules

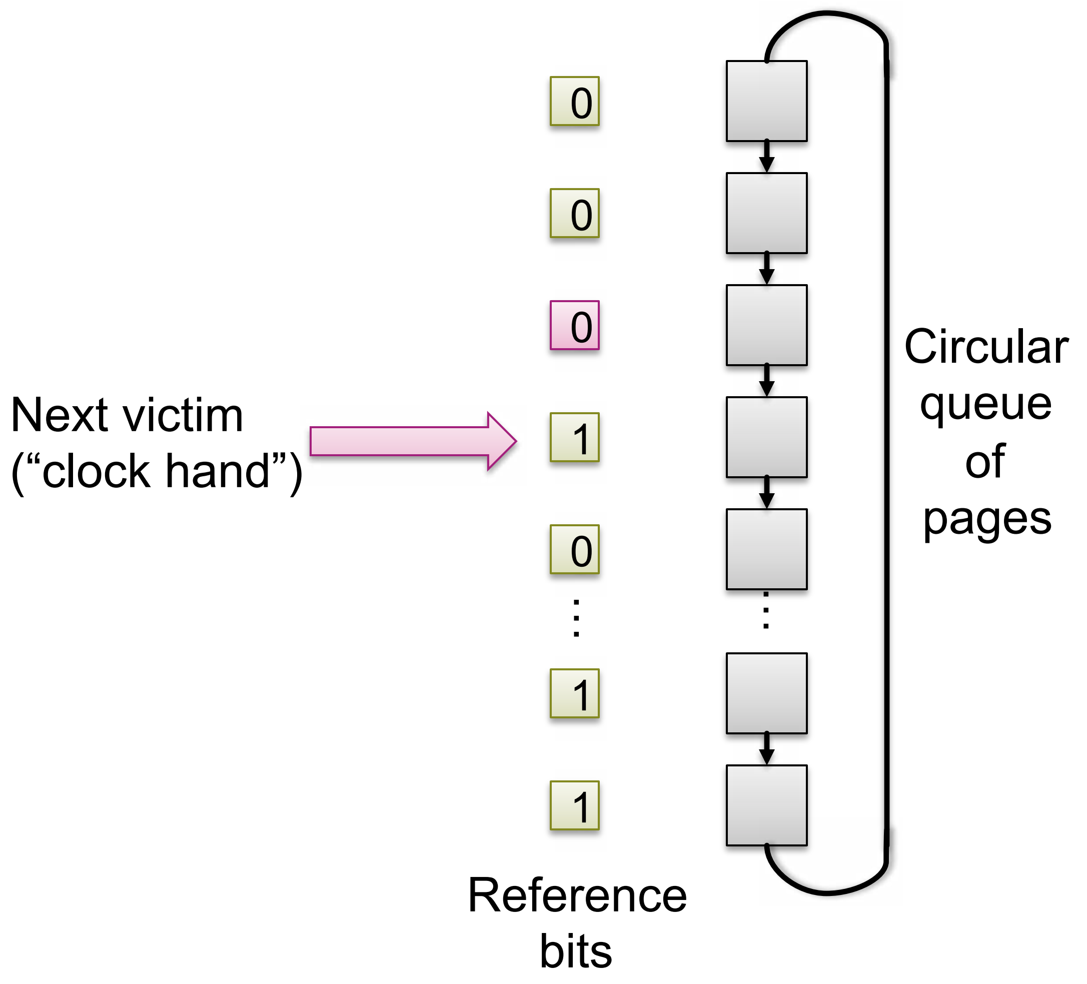
Second-chance (clock) page replacement algorithm



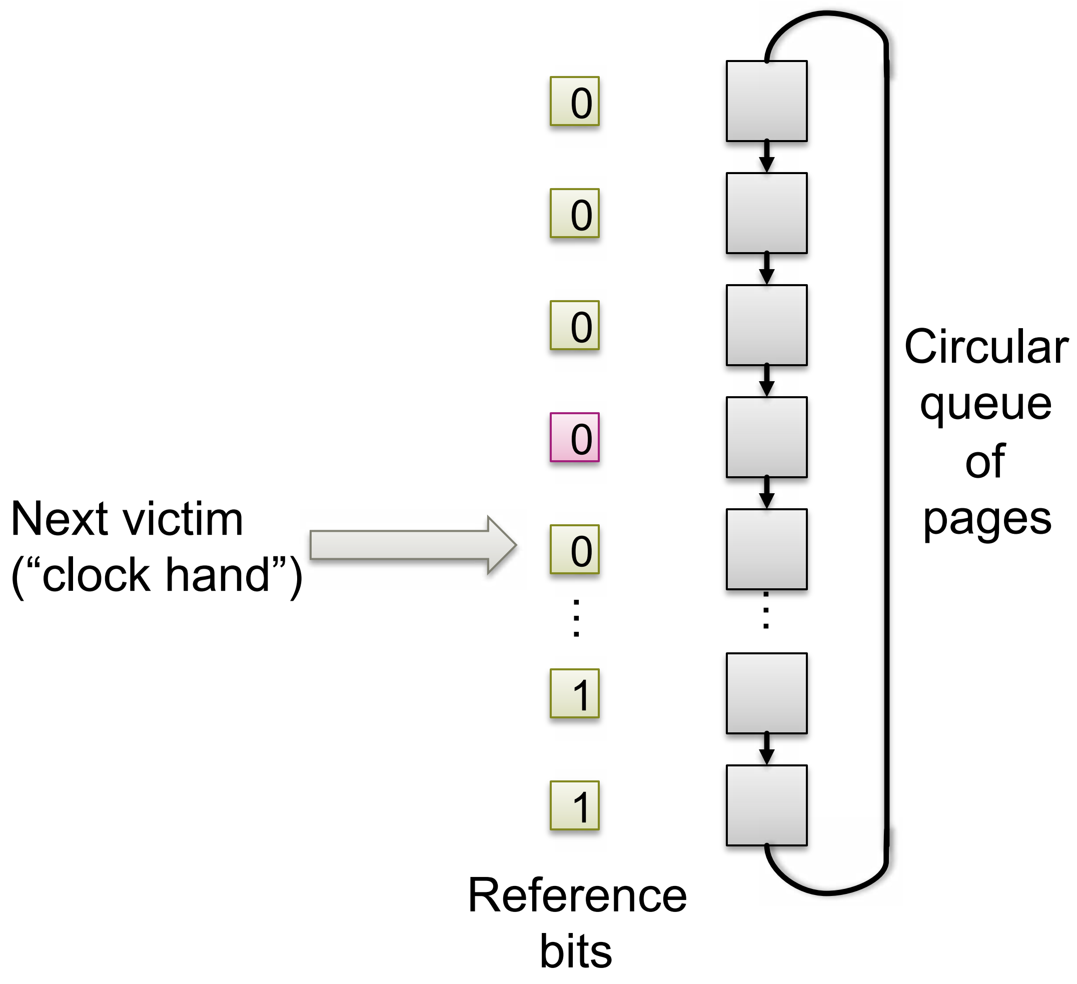
Second-chance (clock) page replacement algorithm



Second-chance (clock) page replacement algorithm



Second-chance (clock) page replacement algorithm



Frame allocation policies

Allocation of frames

- **Each process needs minimum number of pages**
- **Example: IBM 370 – 6 pages to handle SS MOVE instruction:**
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle from
  - 2 pages to handle to
- **Two major allocation schemes**
  - fixed allocation
  - priority allocation

Fixed allocation

- **Equal allocation**
  - all processes get equal share
- **Proportional allocation**
  - allocate according to the size of process

$$s_i = \text{size of process } p_i$$

$$S = \sum s_i$$

$$m = \text{total number of frames}$$

$$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$$

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 64 \approx 5$$

$$a_2 = \frac{127}{137} \times 64 \approx 59$$

Priority allocation

- **Proportional allocation scheme**
- **Using priorities rather than size**

- **If process P_i generates a page fault, select:**
  1. one of its frames, or
  2. frame from a process with lower priority

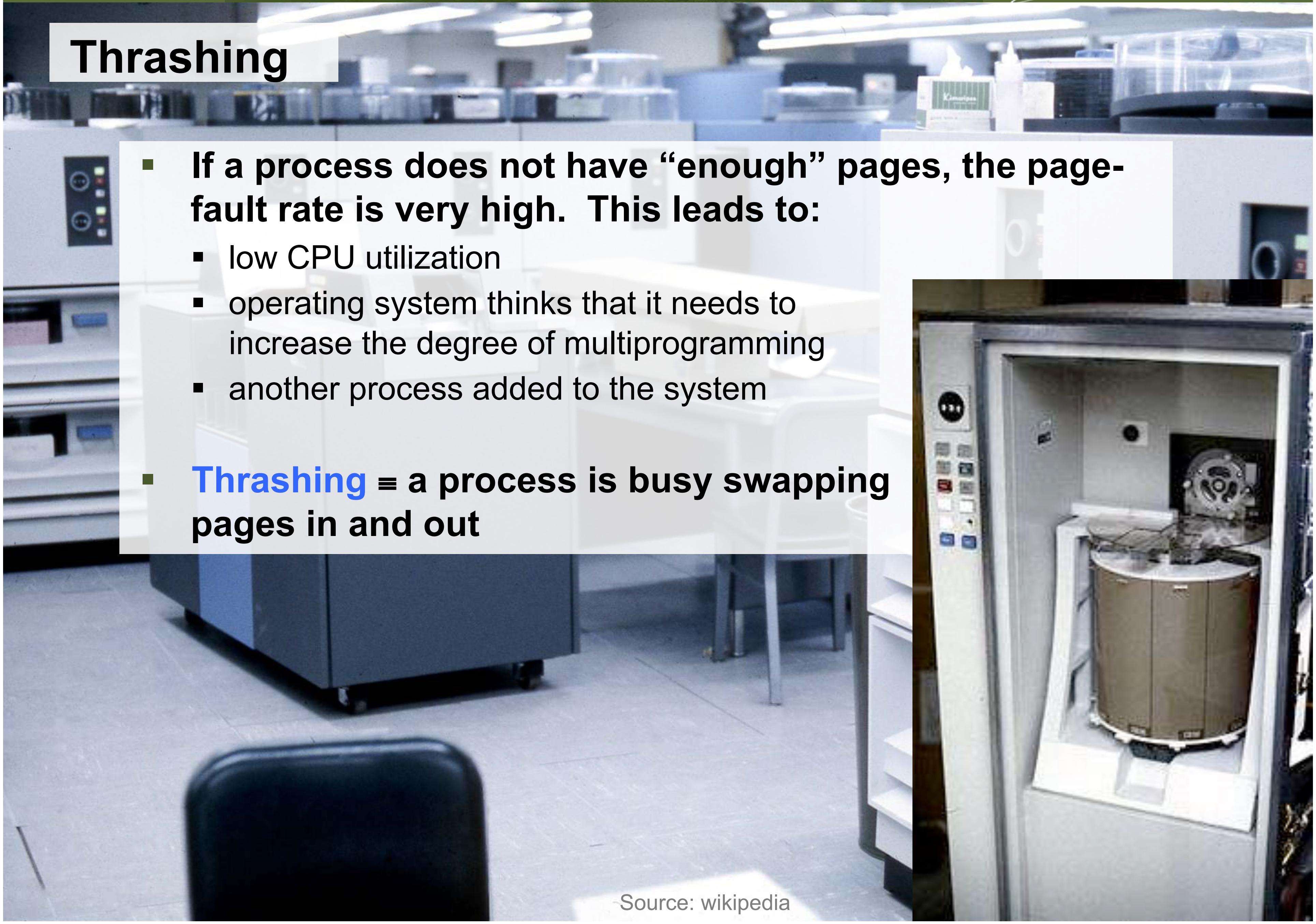
Global vs. local allocation

- **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**

Thrashing

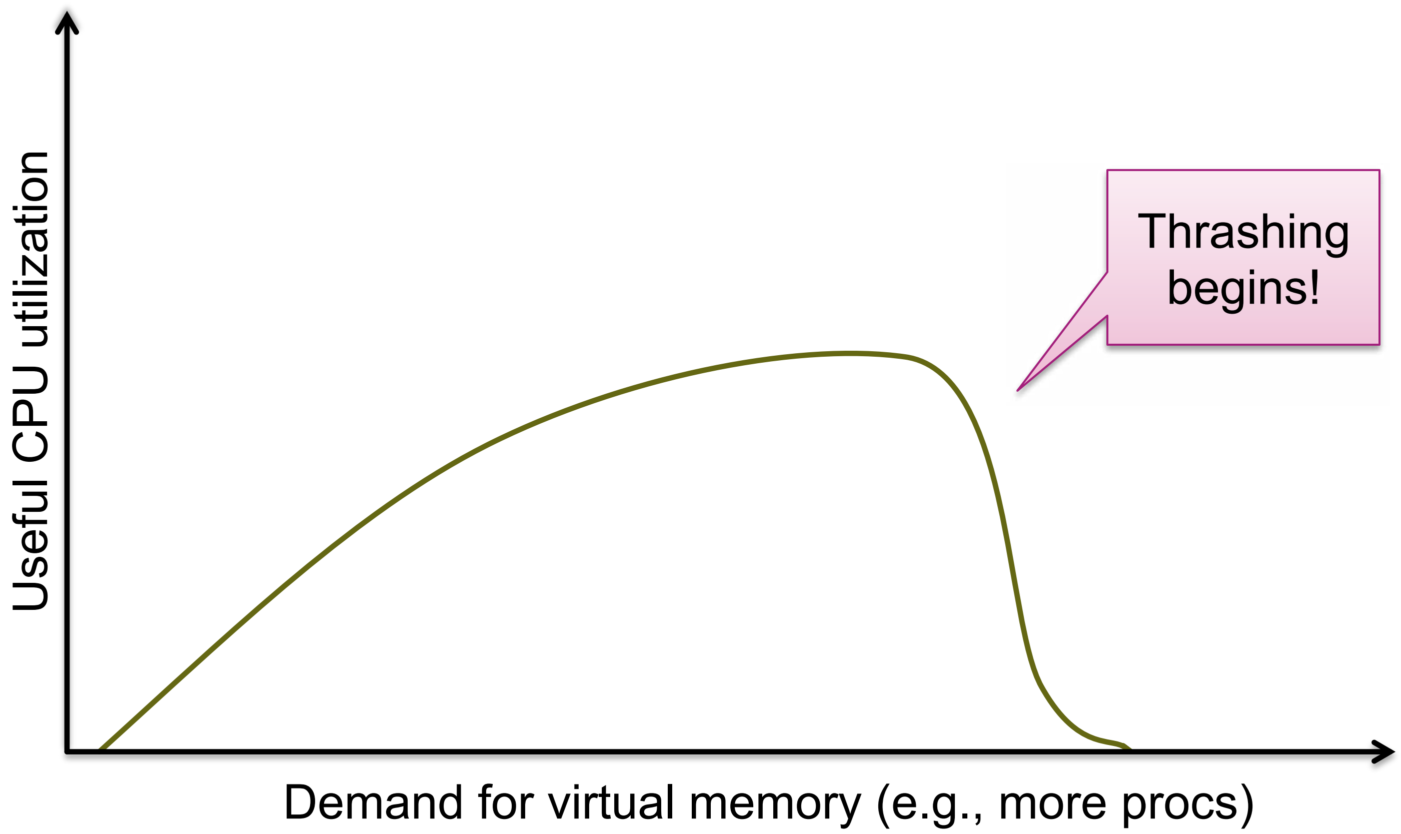
- **If a process does not have "enough" pages, the page-fault rate is very high. This leads to:**
  - low CPU utilization
  - operating system thinks that it needs to increase the degree of multiprogramming
  - another process added to the system

- **Thrashing ≡ a process is busy swapping pages in and out**



Source: wikipedia

Thrashing

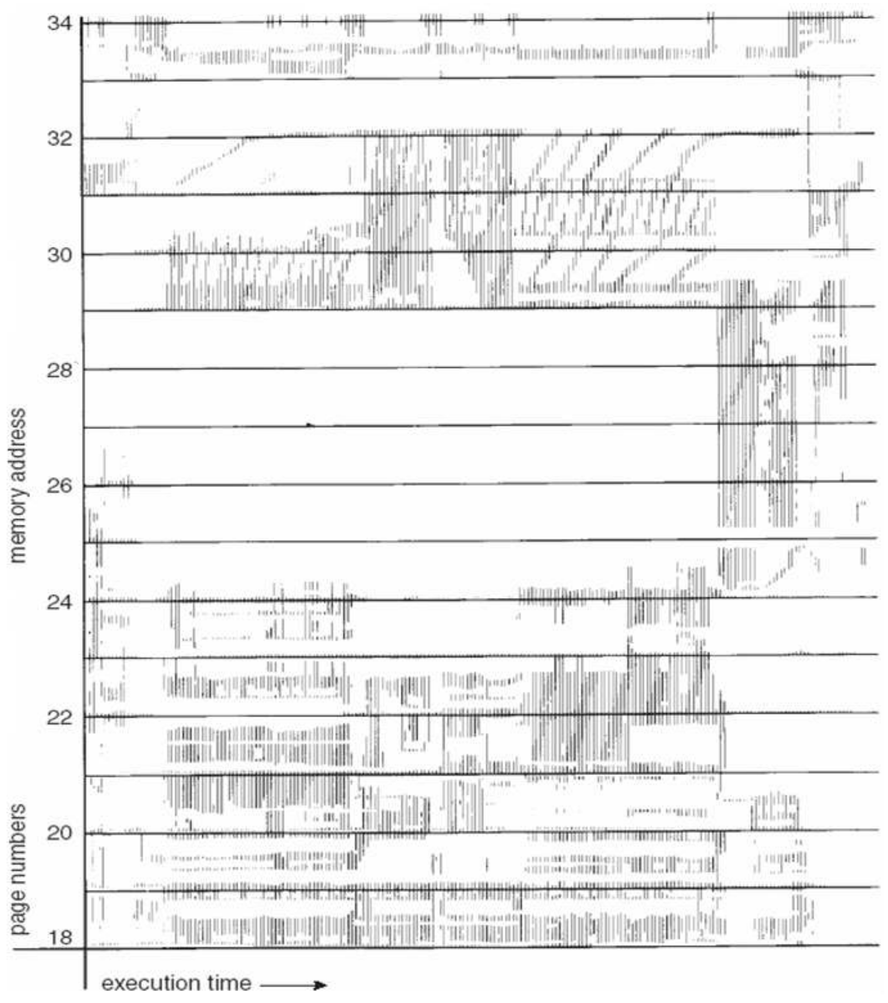

Useful CPU utilization
Thrashing begins!
Demand for virtual memory (e.g., more procs)

Demand paging and thrashing

- **Why does demand paging work?**
  Locality model
  - Process migrates from one locality to another
  - Localities may overlap

- **Why does thrashing occur?**
  Σ size of localities > total memory size

Locality in a memory reference pattern



Working-set model

- **Δ ≡ working-set window**
  ≡ a fixed number of page references
  - Example: 10,000 instruction

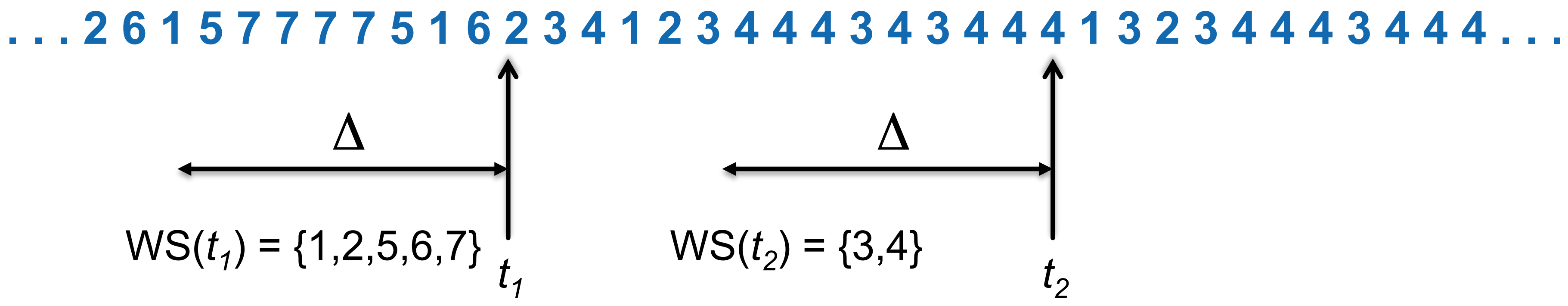
- **WSS_i (working set of Process P_i) = total number of pages referenced in the most recent Δ (varies in time)**
  - Δ too small $\Rightarrow$ will not encompass entire locality
  - Δ too large $\Rightarrow$ will encompass several localities
  - $\Delta = \infty \Rightarrow$ will encompass entire program

Allocate *demand frames*

- **$D = \Sigma\ WSS_i \equiv$ total demand frames**
  - Intuition: how much space is really needed
- **$D > m \Rightarrow$ Thrashing**
- **Policy: if $D > m$, suspend some processes**

Working-set model

Page reference string:



Keeping track of the working set

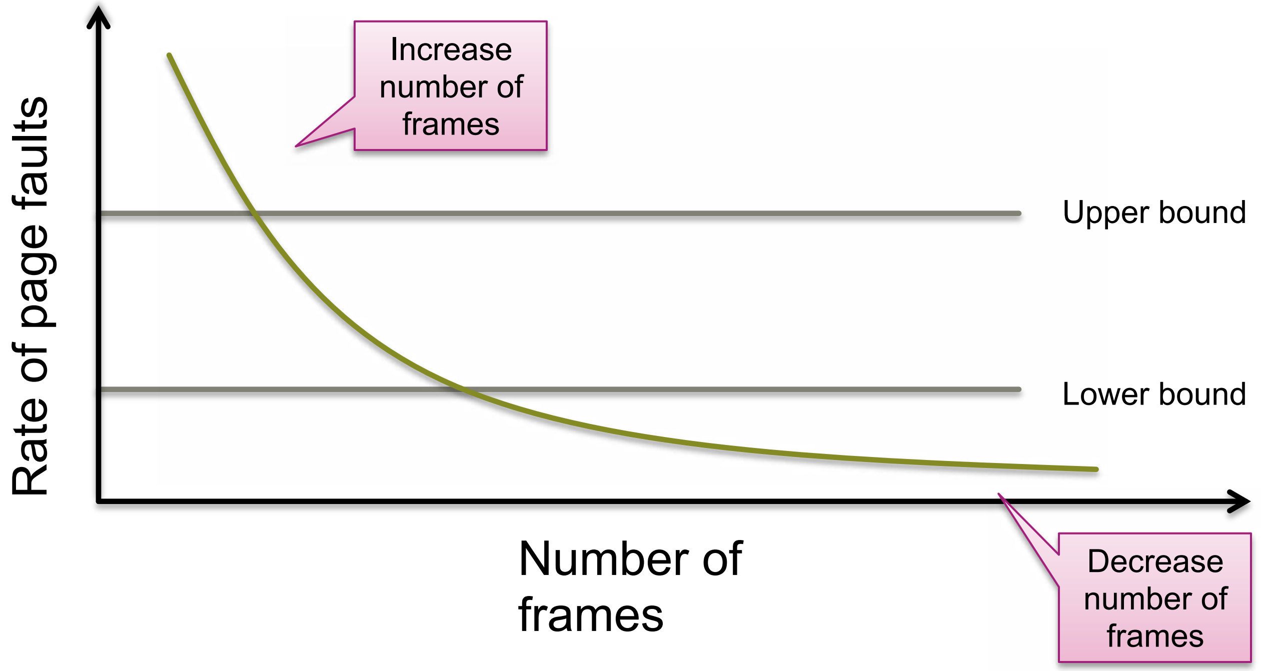
- **Approximate with interval timer + a reference bit**
- **Example: Δ = 10,000**
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts shift+copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 ⇒ page in working set
- **Why is this not completely accurate?**
  - Hint: Nyquist-Shannon!

Keeping track of the working set

- **Approximate with interval timer + a reference bit**
- **Example: Δ = 10,000**
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts shift+copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 ⇒ page in working set
- **Why is this not completely accurate?**
- **Improvement = 10 bits and interrupt every 1000 time units**

Page-fault frequency scheme

- **Establish "acceptable" page-fault rate**
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame


Increase number of frames
Upper bound
Lower bound
Rate of page faults
Number of frames
Decrease number of frames