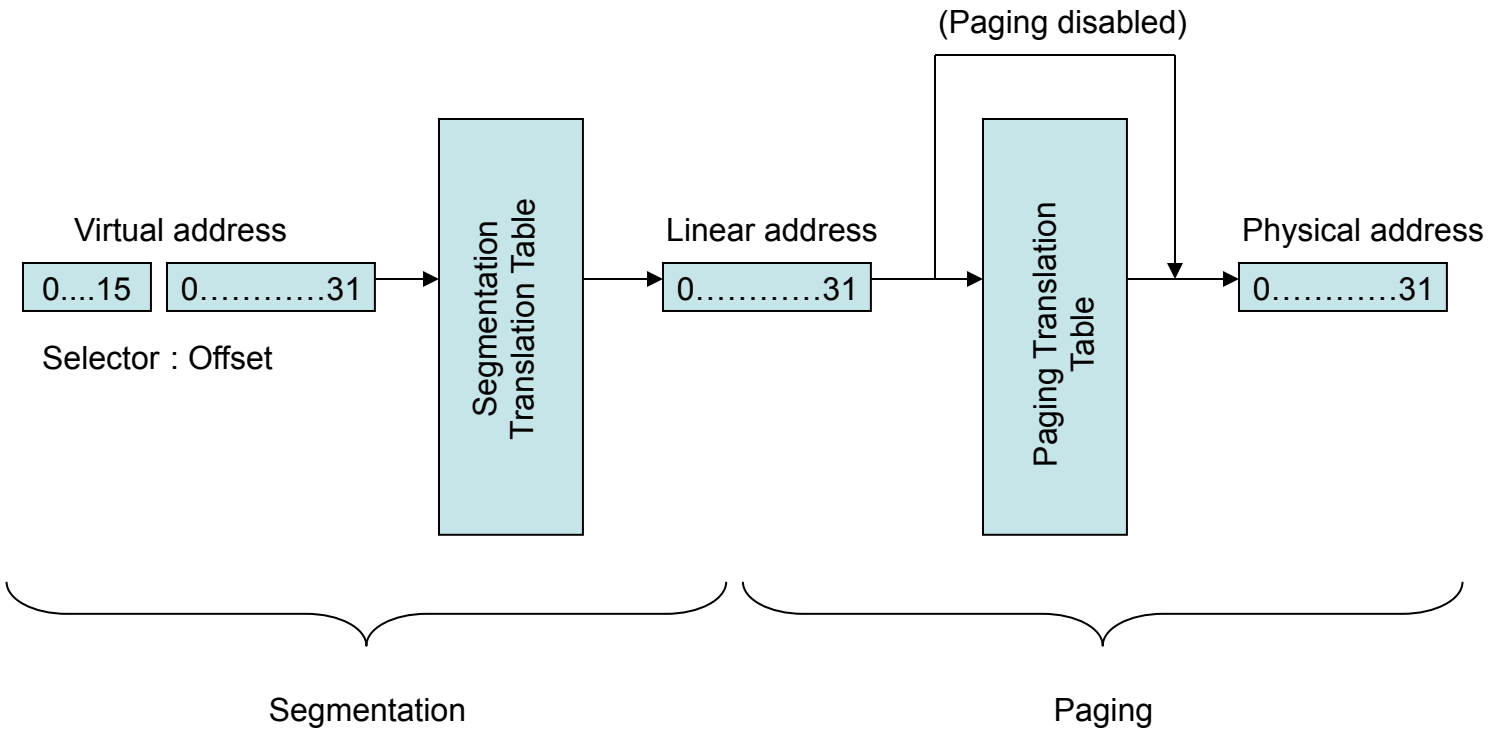
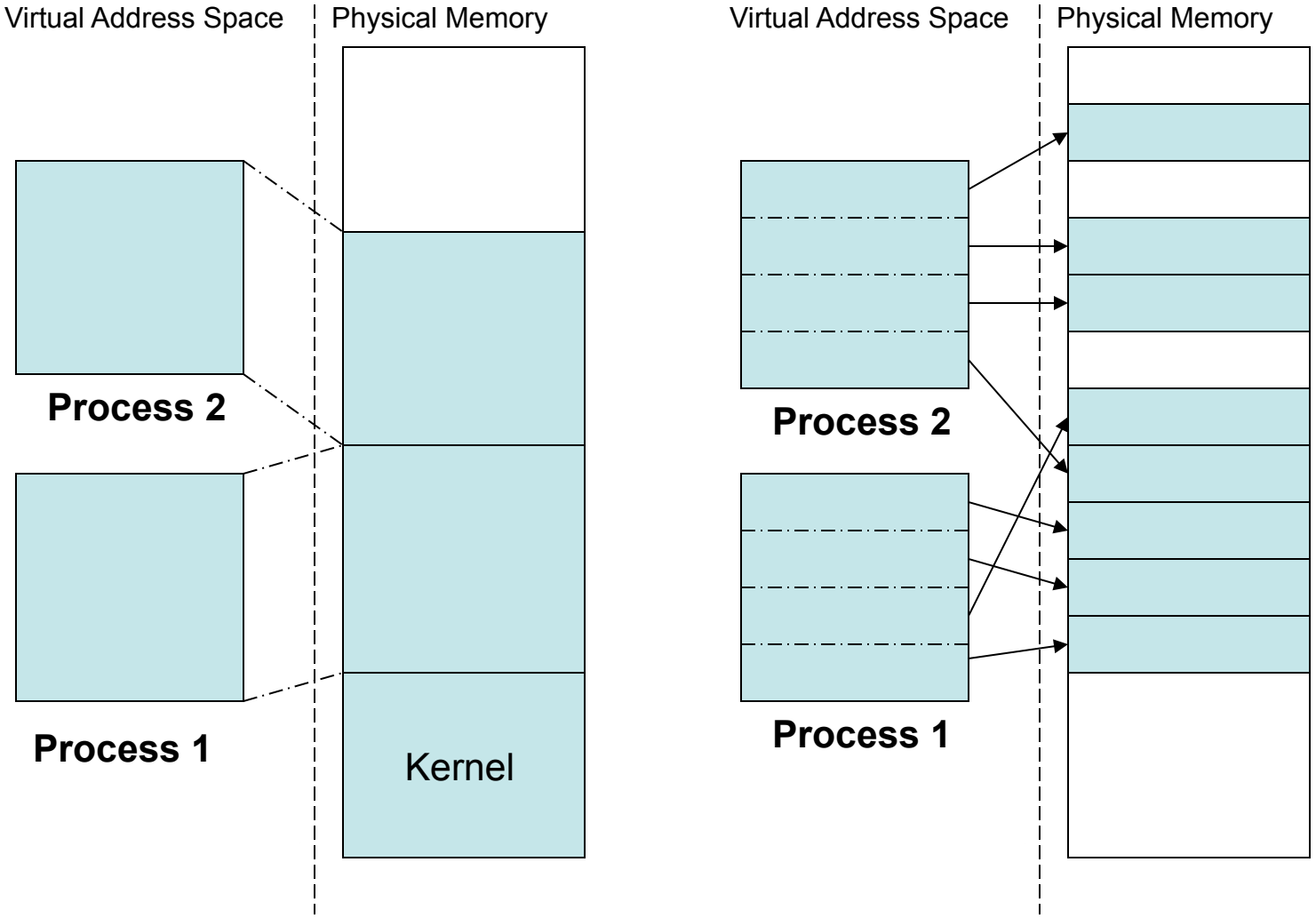
Virtual Memory

x86 Address Translation



Raspberry pi: no segmentation

Segmentation vs Paging

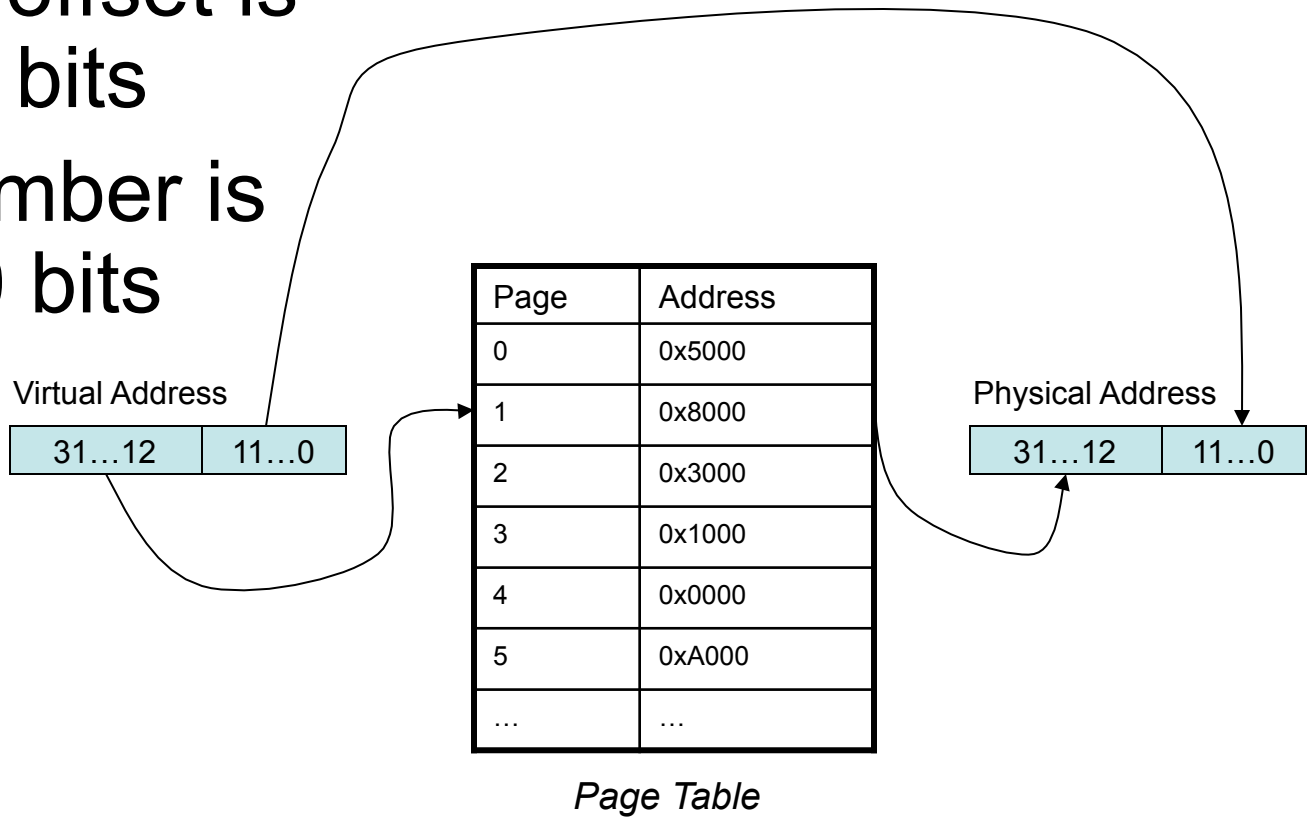


Paging

- An address space is divided into fixed-size chunks called *pages*.
- A data structure called a *page table* maps virtual page numbers to physical page numbers.
- When paging is enabled, every memory reference is translated from virtual to physical through the page table.

Paging Example

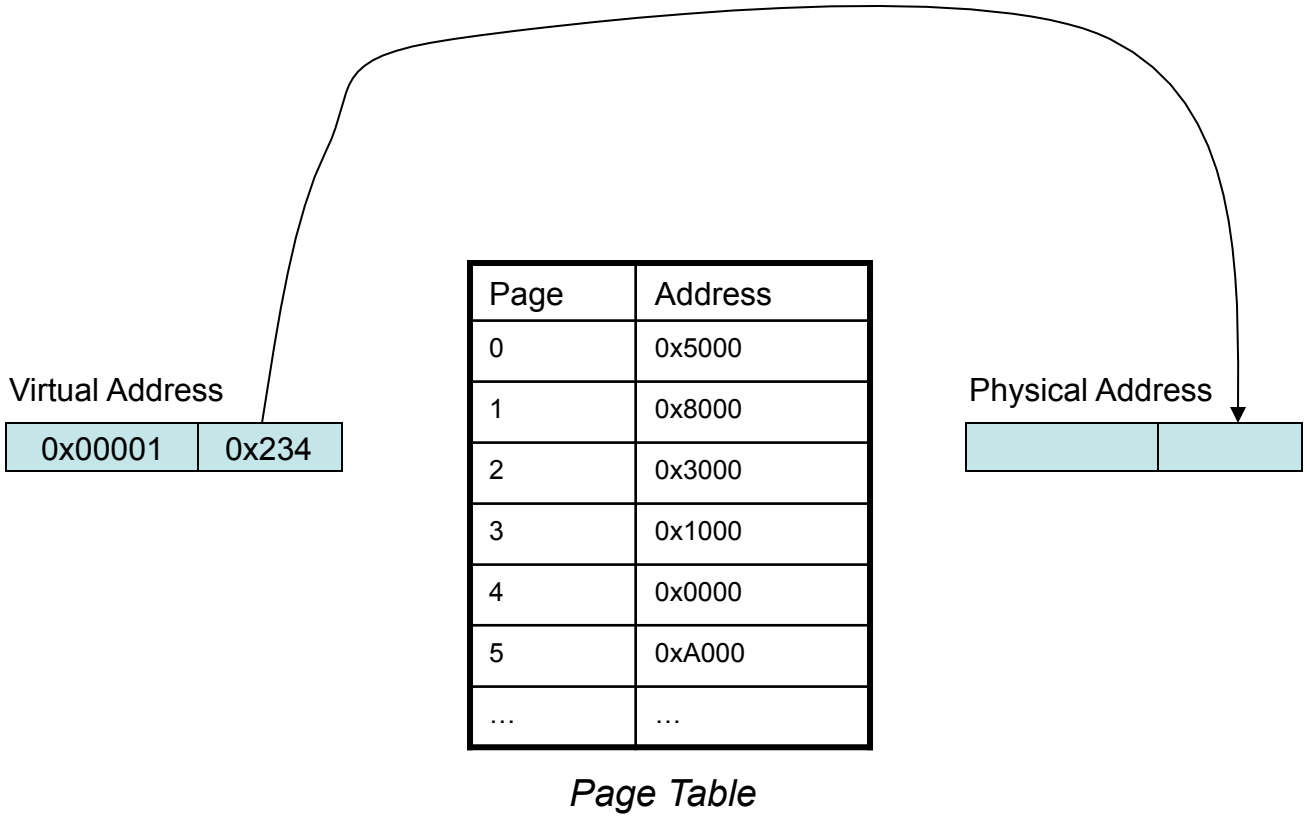
- Page size: 4 KB
- Address offset is lower 12 bits
- Page number is upper 20 bits



Page Table

Paging Example

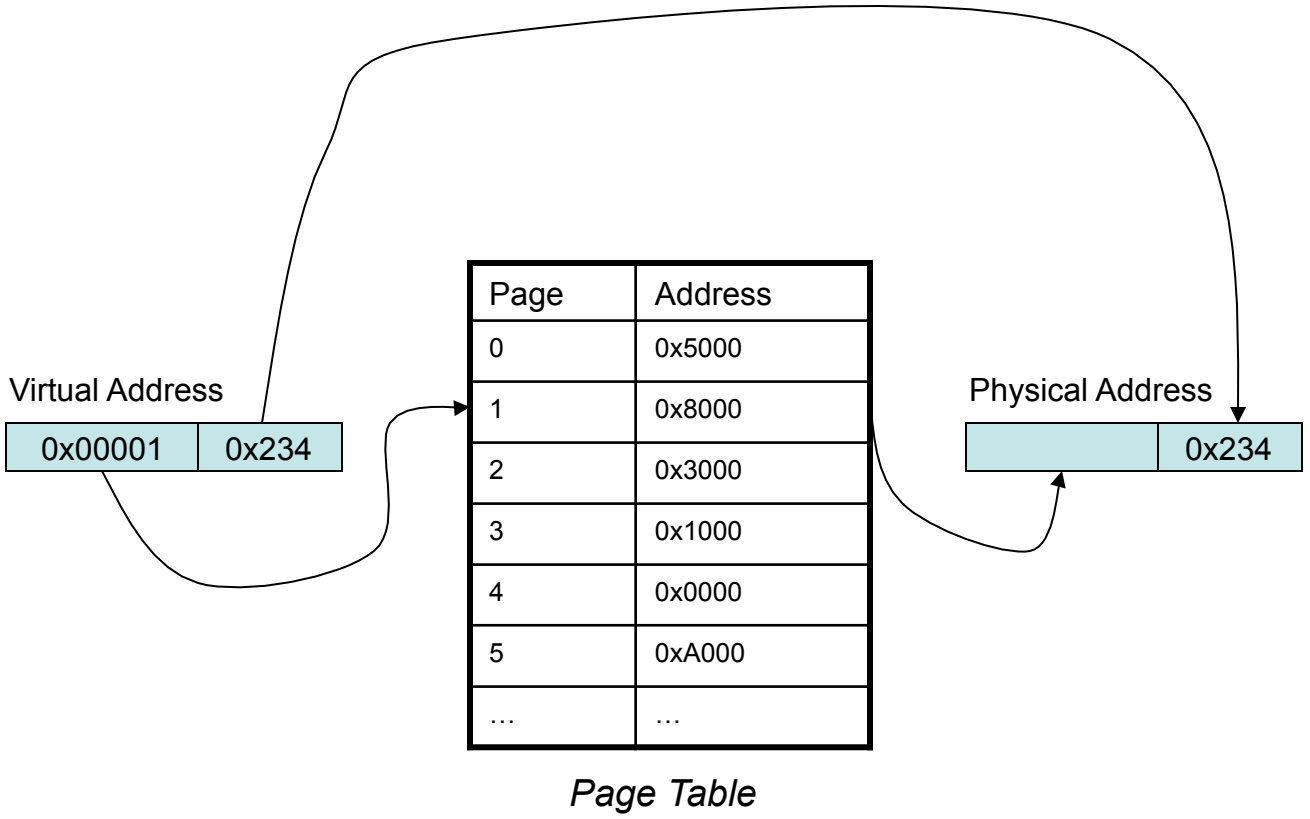
- Translate virtual address 0x1234



Page Table

Paging Example

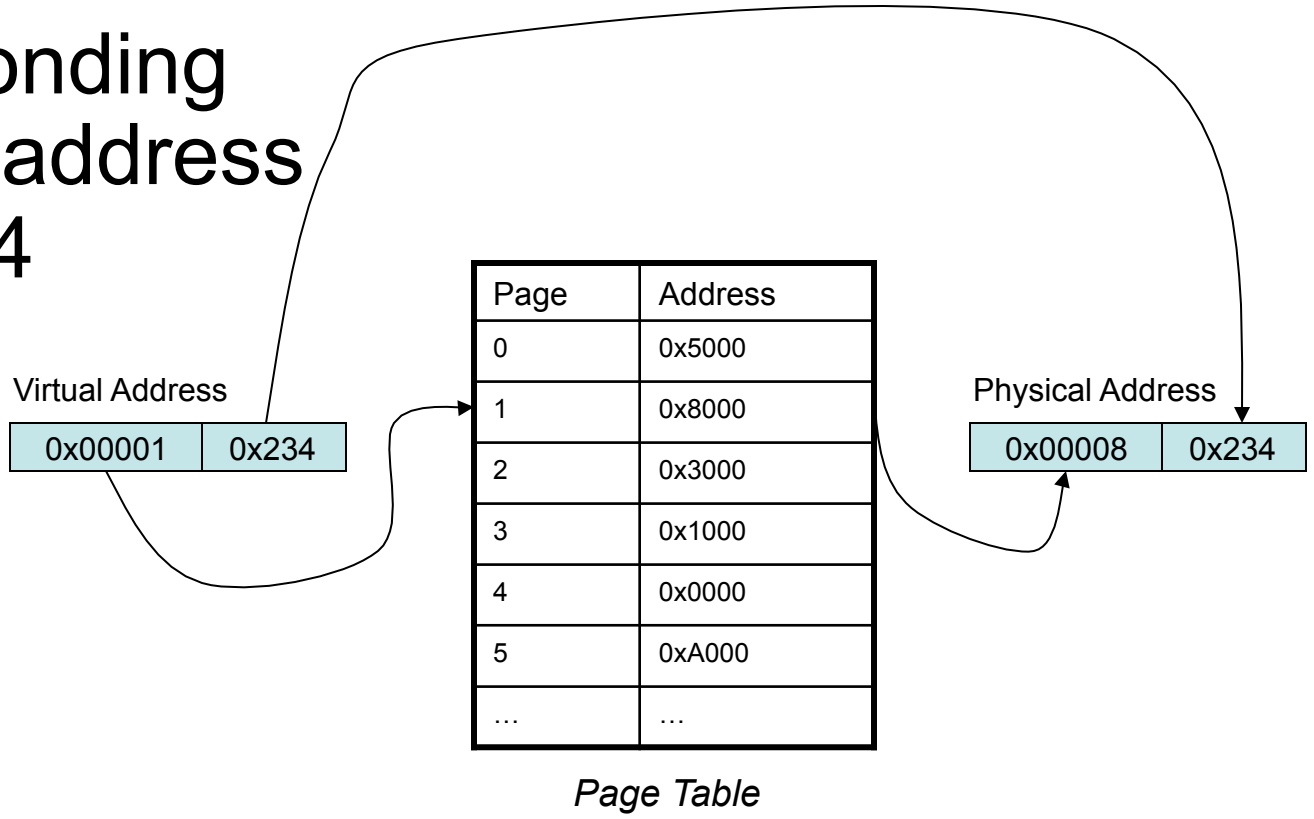
- Translate virtual address 0x1234



Page Table

Paging Example

- Translate virtual address 0x1234
- Corresponding physical address is 0x8234



Page Table

Segmentation vs Paging

- State required per address space:
  - Segmentation: base, offset
  - Paging: full page table
- Paging uses more state, but consider flexibility:
  - Expanding an address space is much easier with paging
  - Paging simplifies other features (sharing, loading on demand)

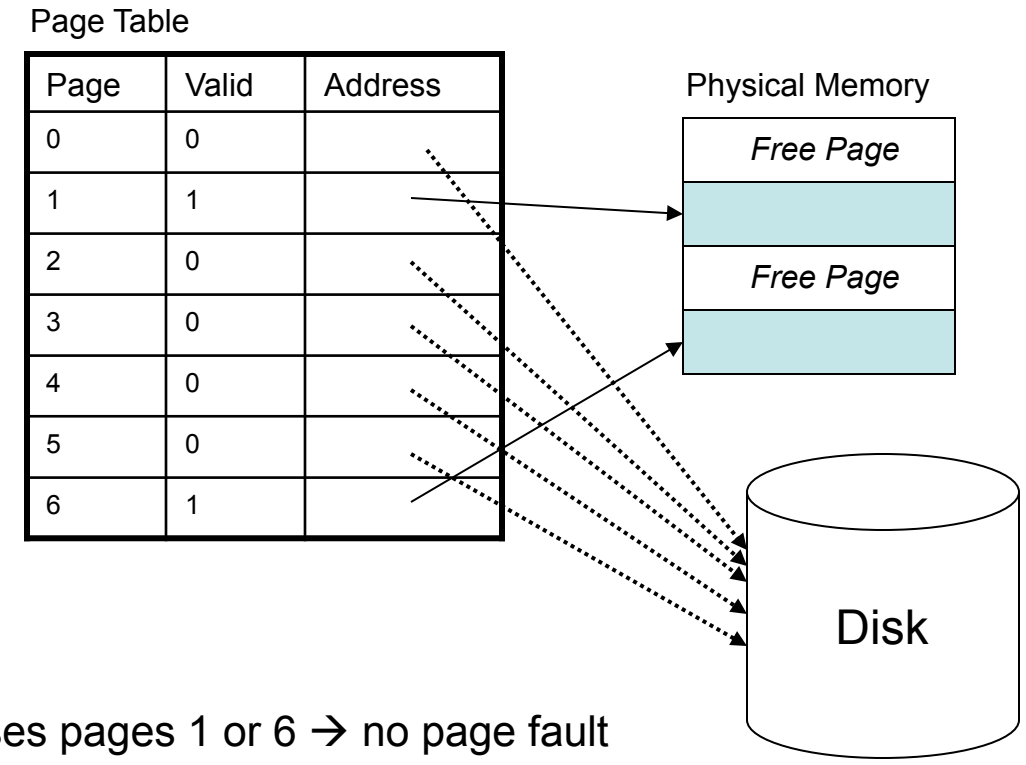
Virtual Memory

- Paging also provides a convenient way to implement virtual memory:
  - Key idea: entire address space does not need to be in physical memory at the same time!
- Implemented with an additional valid bit in each page table entry:
  - If valid bit is 1, the page is in memory
  - If valid bit is 0, the page is not in memory

Implementing Virtual Memory

- For pages in memory, translation proceeds as usual
- For pages not in memory, hardware detects an invalid page table entry and generates a *page fault*
- A page fault is an exception, just like an illegal instruction or divide-by-zero
- Unlike those exceptions, a page fault is not necessarily fatal!
- On the x86, exceptions can be handled similar to an interrupt.

Virtual Memory with Paging

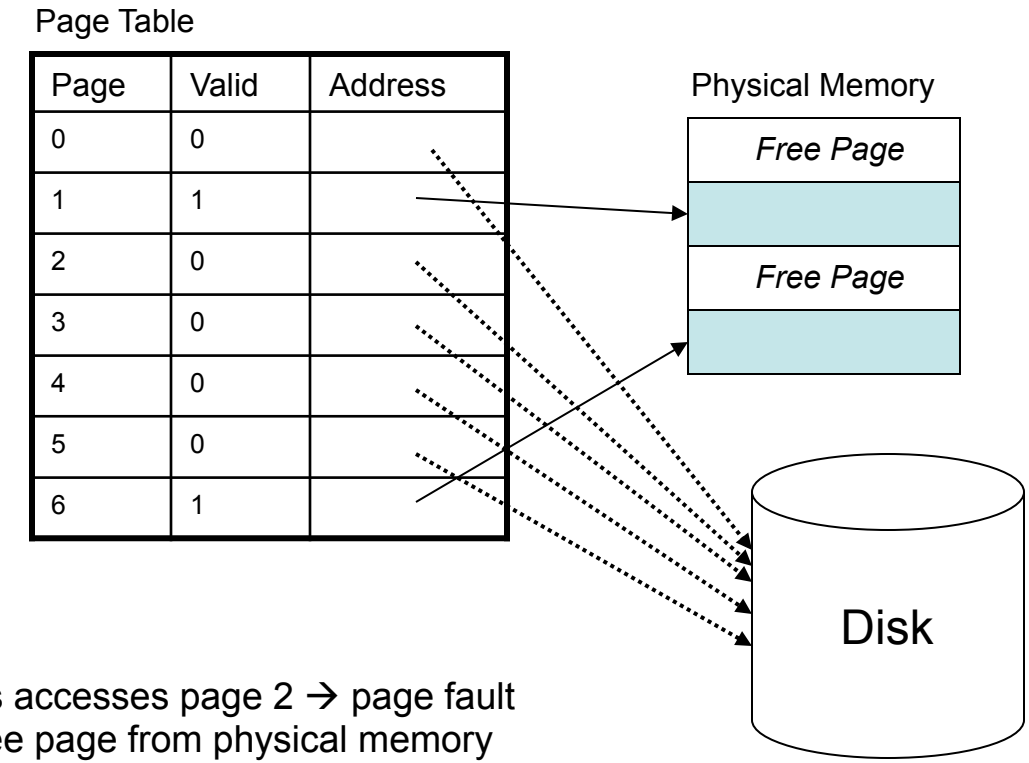


- If process accesses pages 1 or 6 → no page fault
- If process accesses pages 0, 2, 3, 4, or 5 → page fault

Handling Page Faults

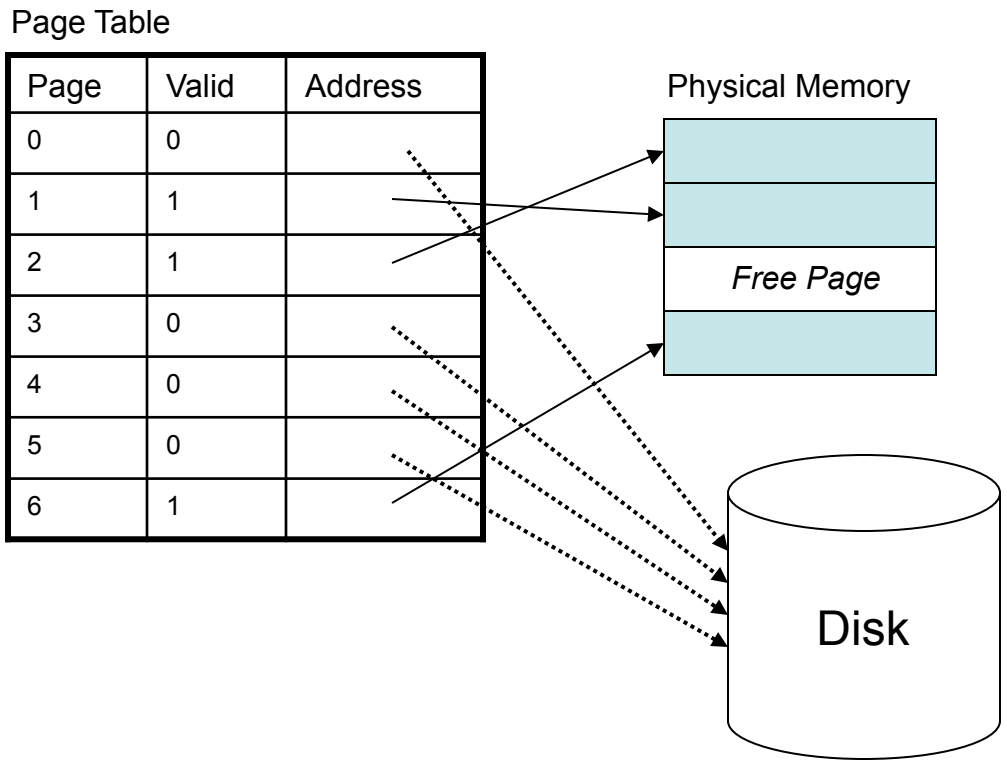
- If a memory reference is for a valid address that is not in memory at the time, the OS must:
  - Read the appropriate page into memory
  - Restart the faulting instruction when the I/O is complete
- Note that the I/O takes a long time (~10 ms) -- paging will be inefficient if a lot of I/O is required!

Virtual Memory with Paging



- Example: process accesses page 2 → page fault
- OS allocates a free page from physical memory
- OS loads page from disk and updates page table
  (see next slide for the result of these operations)

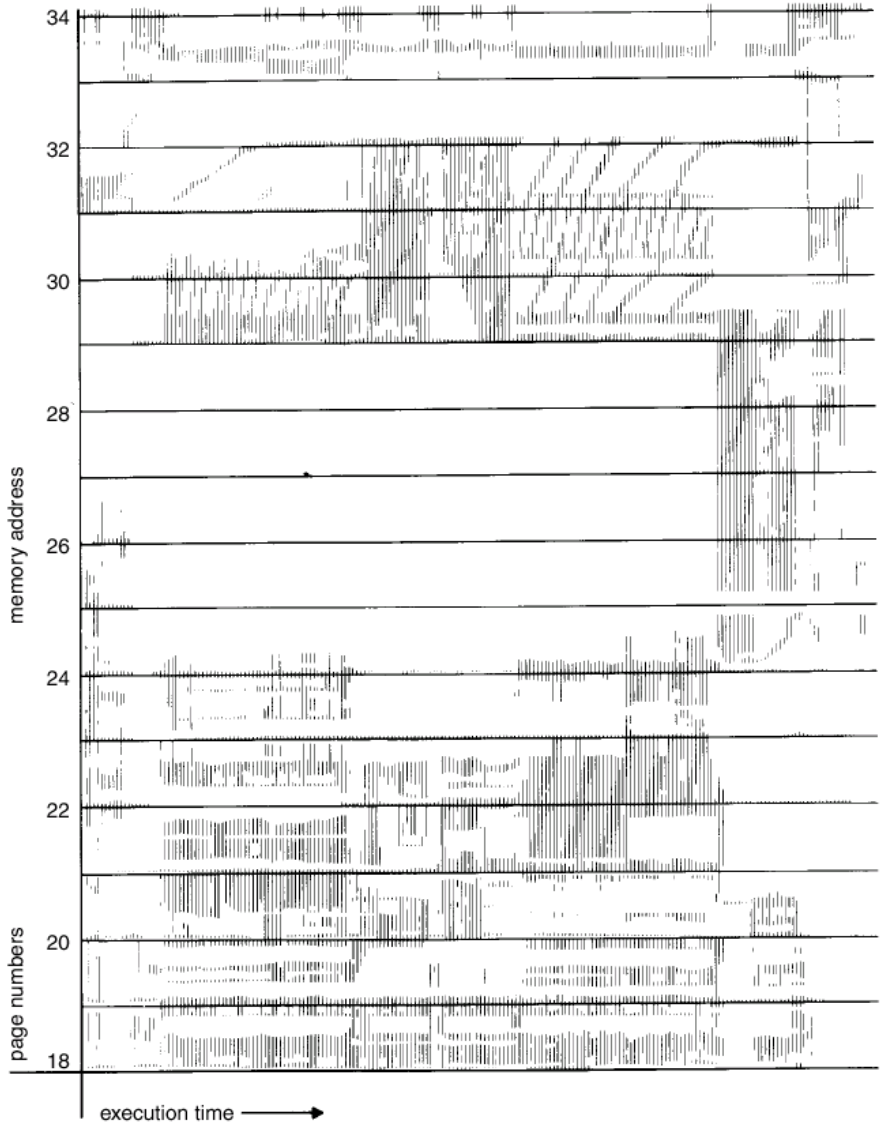
Virtual Memory with Paging



Paging Efficiency

- Virtual Memory exploits *locality* in applications -- programs tend to access the same addresses repeatedly (e.g., for-loop)
- Given an application with locality of memory references, only a small fragment of the full address space is used at any moment
- Goal of a good VM system is to keep these addresses (the *working set*) in physical memory, while leaving unused addresses on disk.

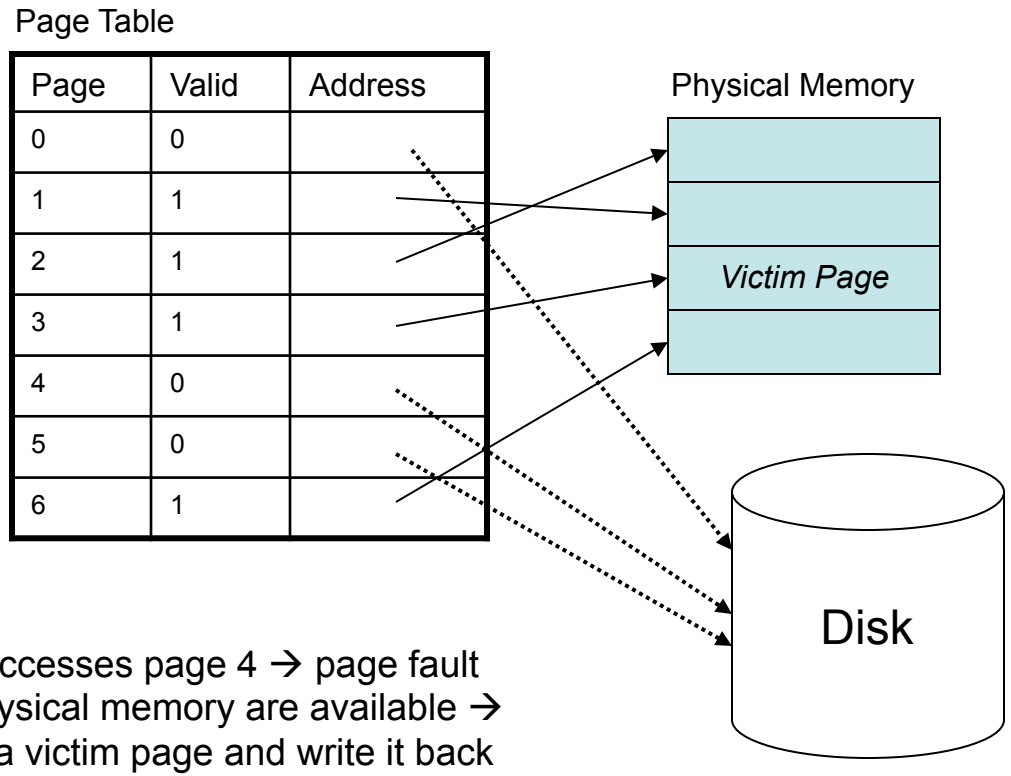
Memory Locality



Page Replacement

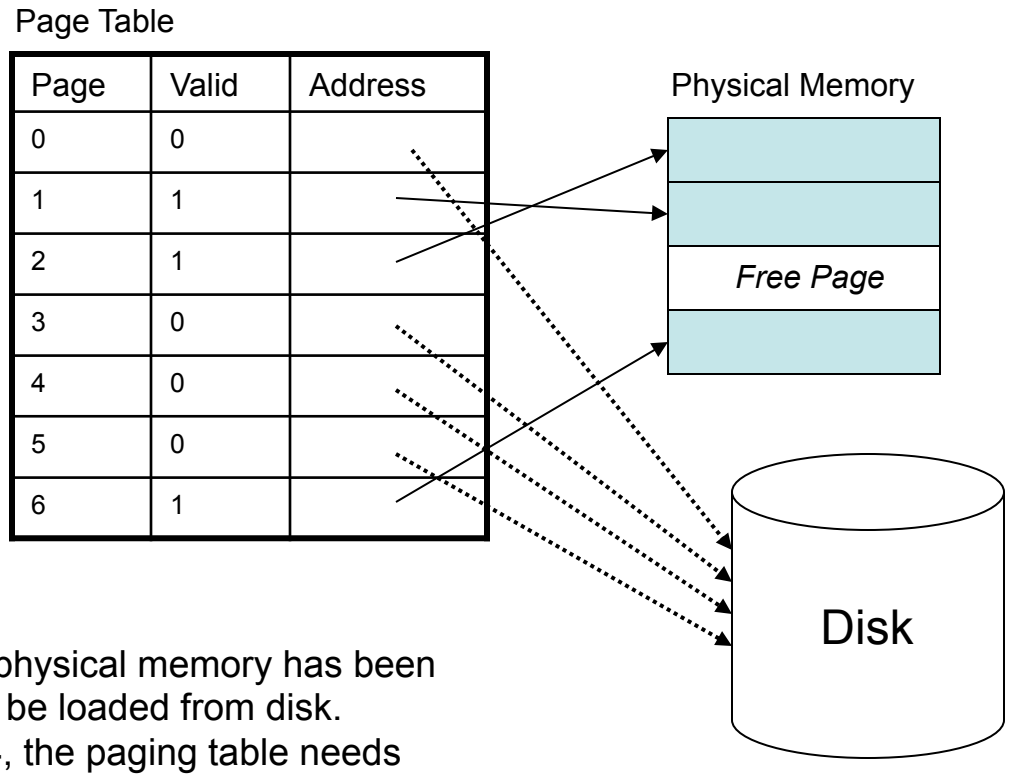
- When handling a page fault, what if there is no free physical page available?
- Need a page replacement algorithm to choose an existing page to evict.
- This page is also referred to as the *victim page*.

Page Replacement



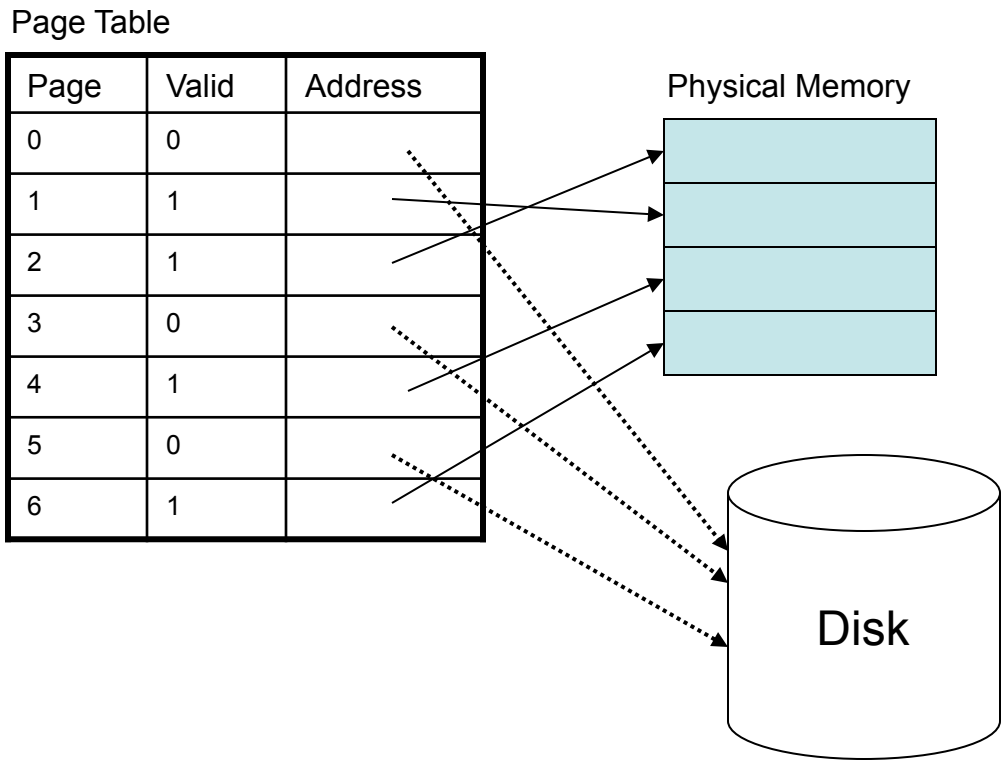
- Example: process accesses page 4 → page fault
- No free pages in physical memory are available → OS needs to select a victim page and write it back to disk (page out)
- OS selects by some algorithm page 3 as the victim page.
- Next slide shows the effect of paging out the victim page.

Page Replacement



- Now that a page in physical memory has been cleared, page 4 can be loaded from disk.
- After loading page 4, the paging table needs to be updated.
- The effect is shown on the next slide.

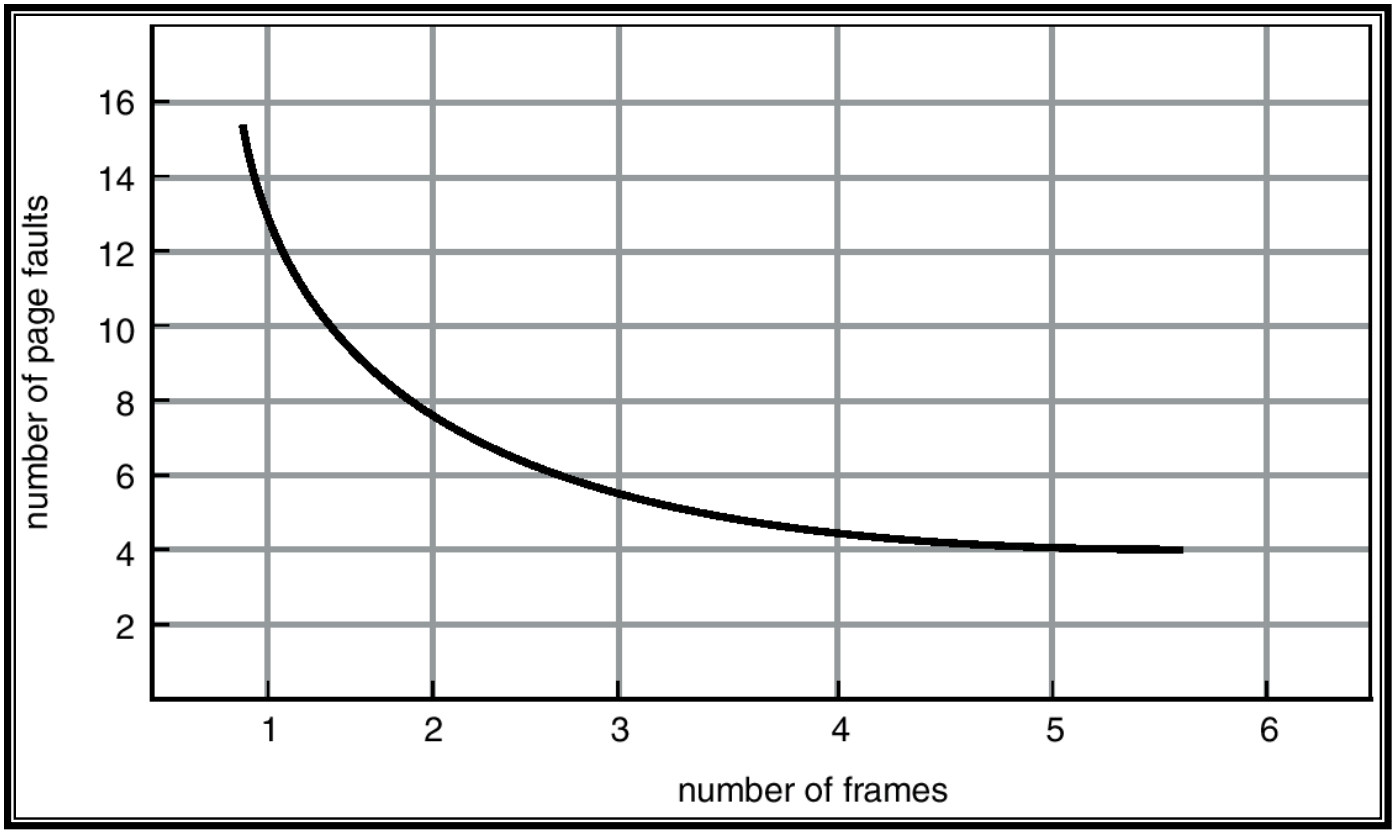
Page Replacement



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., reference string “7, 0, 1, …” means that the process first accesses page 7, then page 0, then page 1, etc.

Graph of Page Faults Versus The Number of Frames

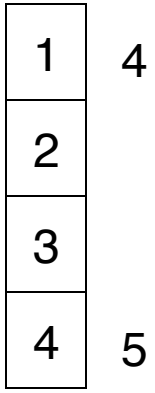


This is the ideal situation one wants: increasing physical memory (number of frames) decreases number of page faults.

Optimal Algorithm

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

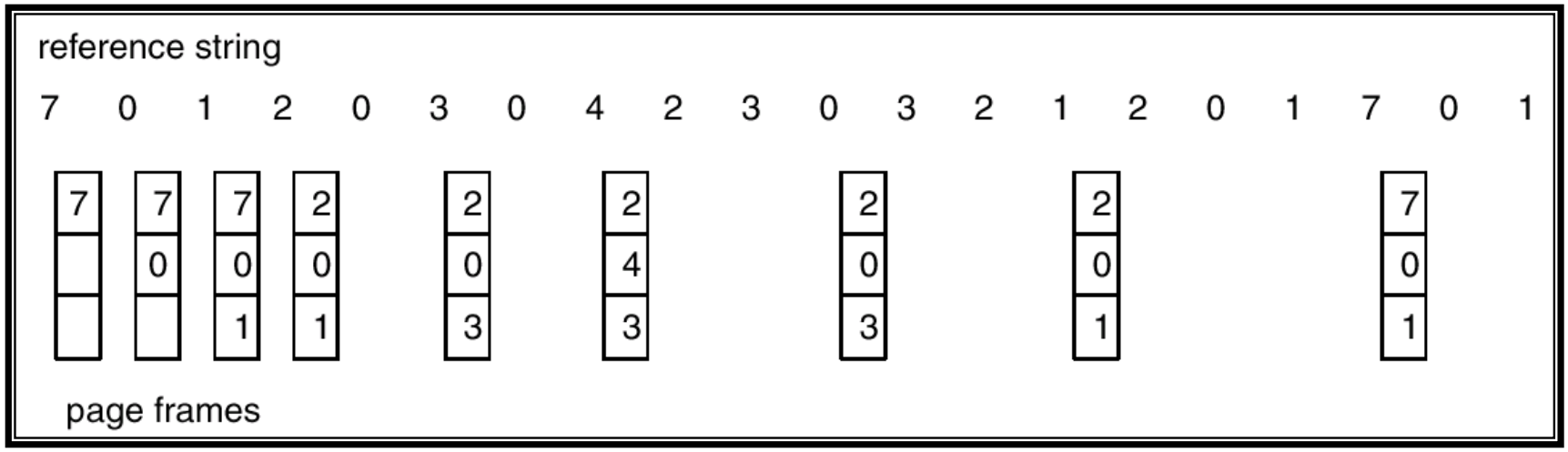
  Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



6 page faults

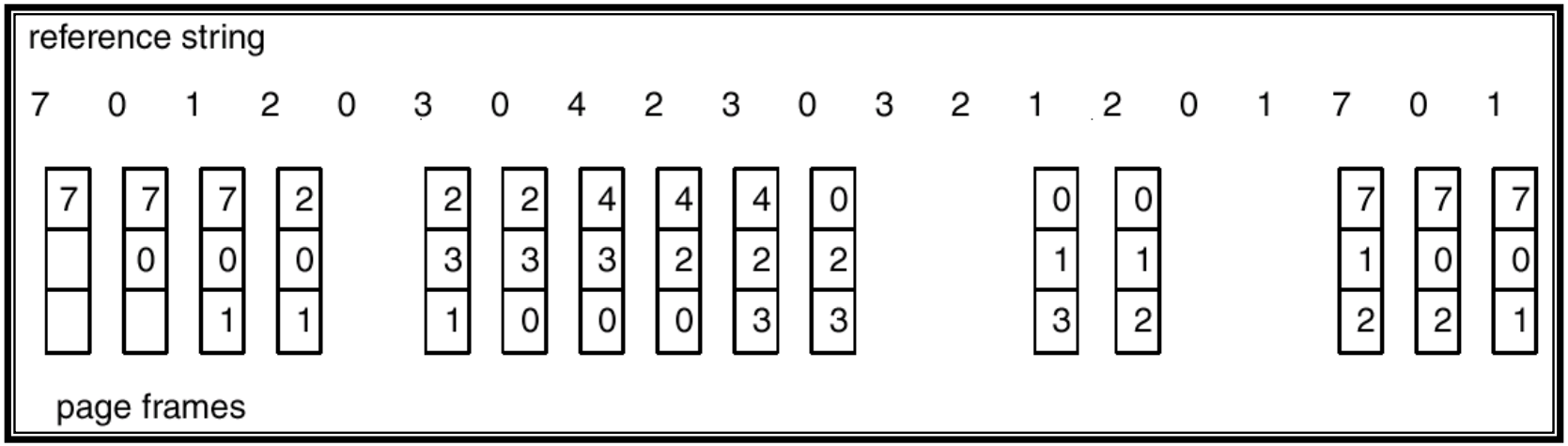
- How do you know this?
- Used for measuring how well your algorithm performs.

Optimal Page Replacement



The reference string denotes the sequence in which a process is accessing pages in virtual memory. In the example above, the process first accesses page 7, then page 0, etc. The assumption in the example above is that physical memory only consists of 3 pages.

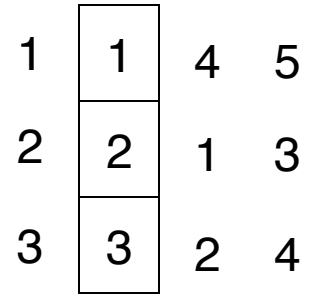
FIFO Page Replacement



With the FIFO page replacement algorithm, the first page to be paged-in will also be the first page to be selected as the next victim page. In the example above, page 7 is the very first page to be paged in, therefore it will also be the first victim page when page 2 needs to be paged in.

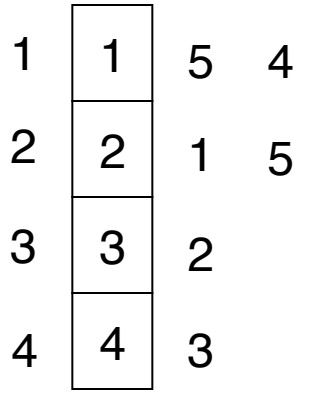
Belady's Anomaly

- 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)



9 page faults

- 4 frames

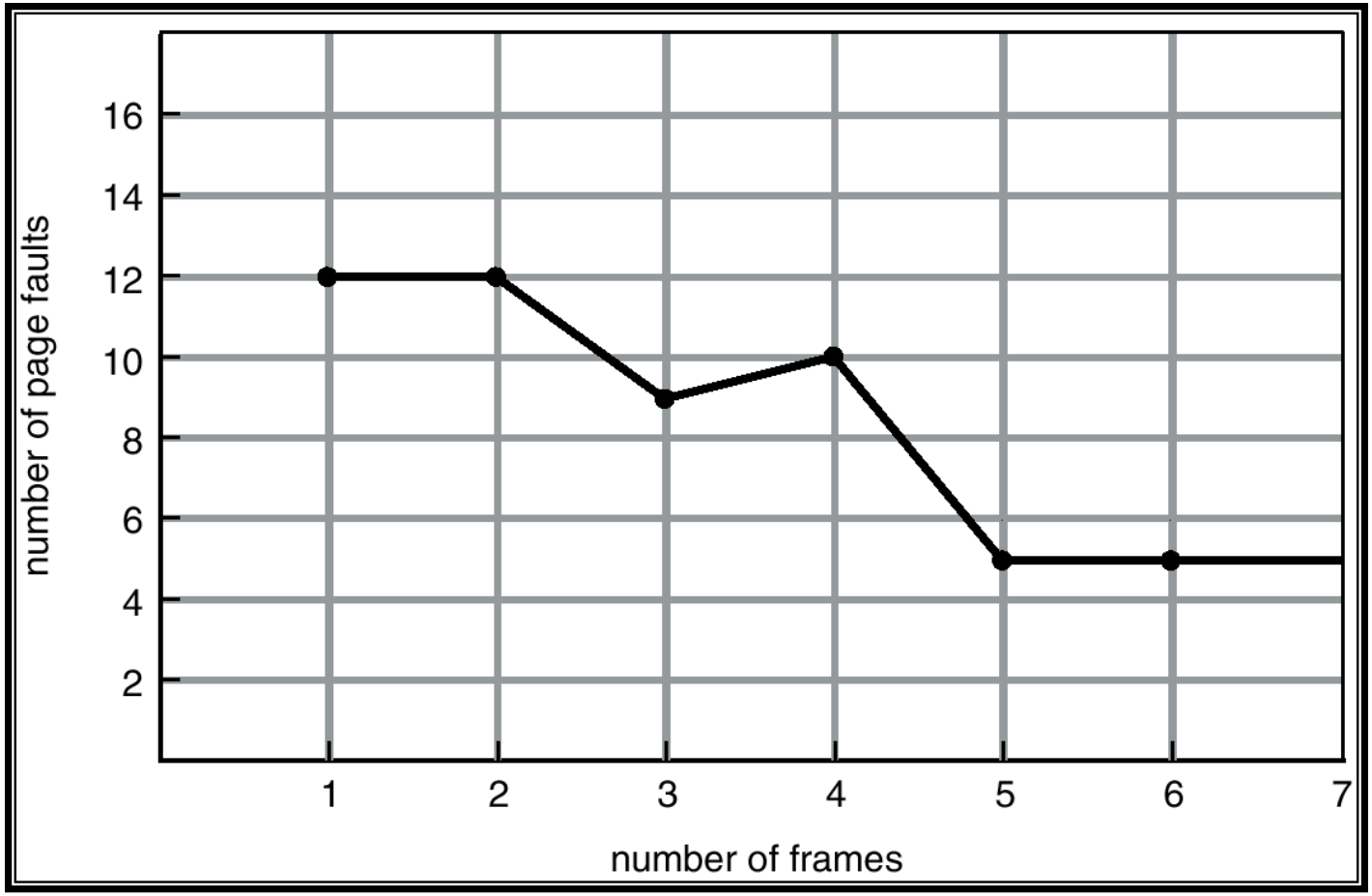


10 page faults

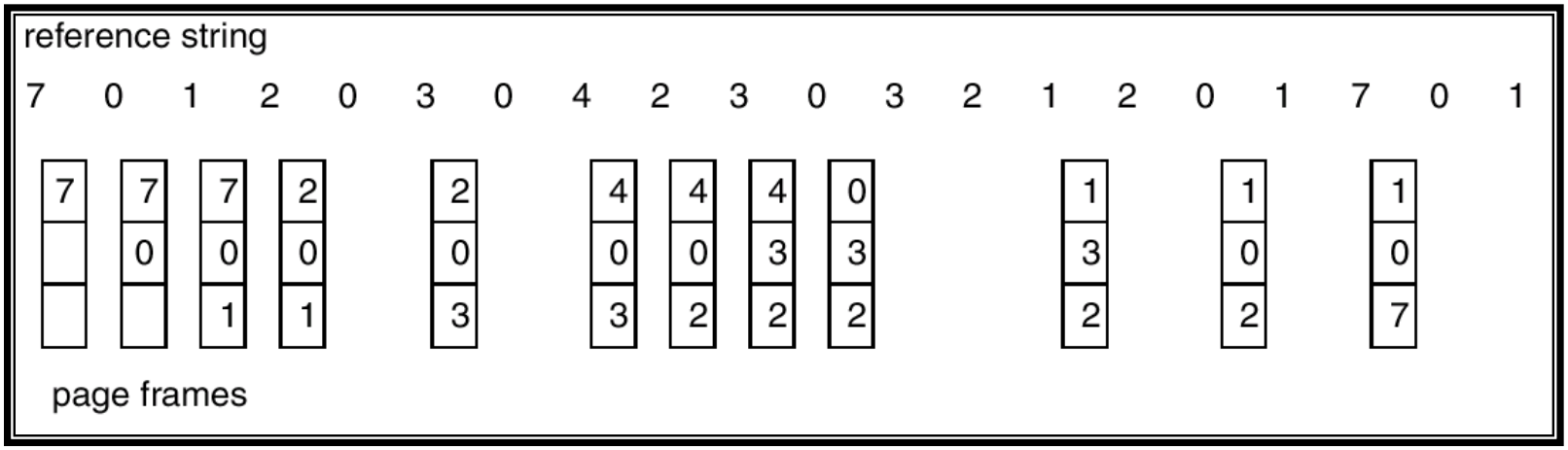
FIFO Replacement – Belady's Anomaly

– more frames ⇒ more page faults!

FIFO Illustrating Belady's Anomaly



LRU Page Replacement

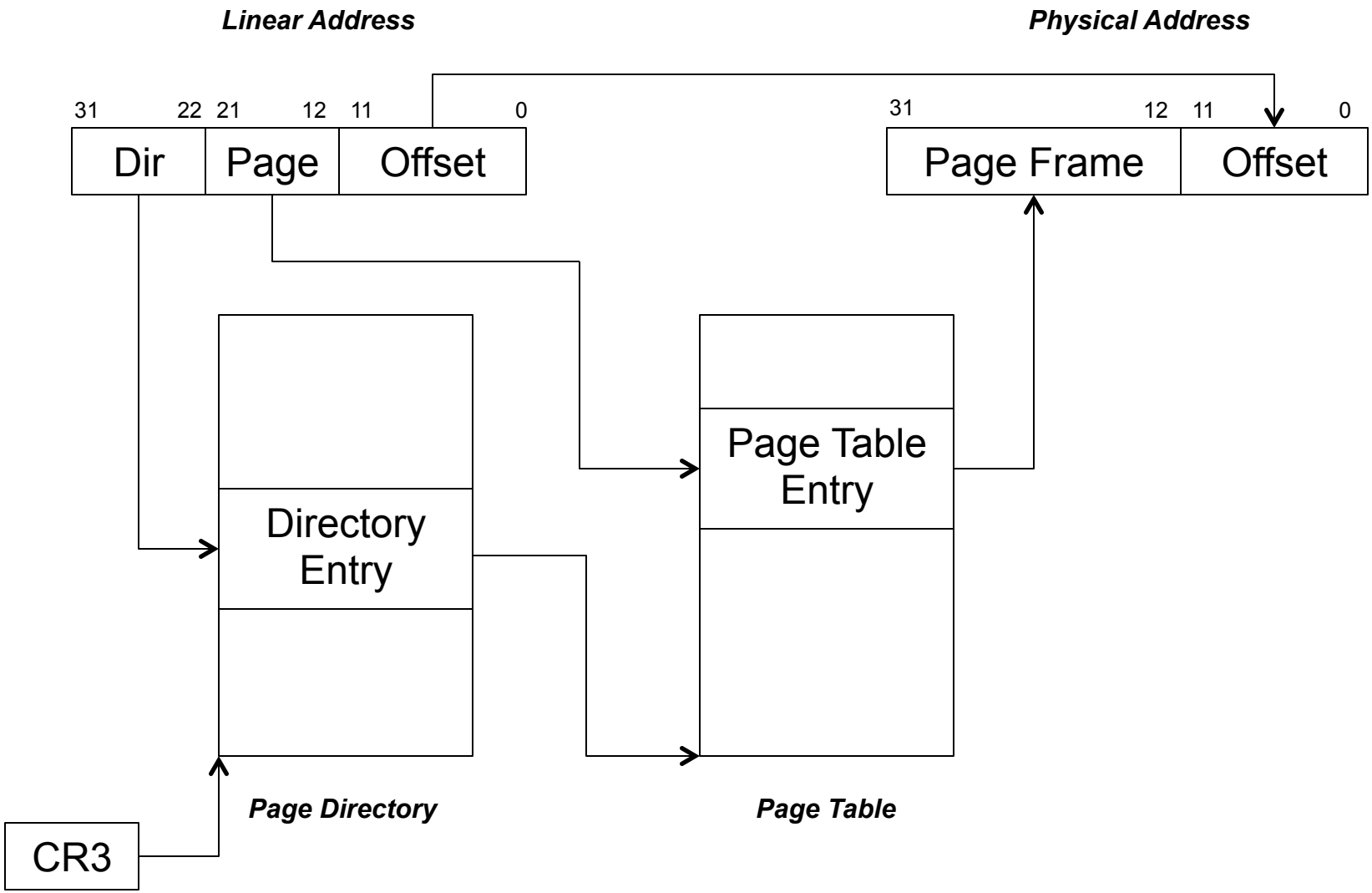


With the LRU (Least Recently Used) page replacement algorithm, the page that has not been used the least (the longest ago) will be selected as the victim page. In the example above, when page 3 is referenced, page 1 will be selected as the victim page since it was accessed the least at this point in time.

x86 Paging Mechanism

- Intel supports paging since the 80386 CPU.
- Paging can be enabled or disabled (controlled by the PG-bit of the CR0 register).
- x86 uses 4K byte page size that are aligned at 4K boundaries.
  → Paging divides the 2^{32} (4G) linear address space into 2^{20} pages each 2^{12} bytes in size.
- Each process has its own page table:
  - Register CR3 defines base address of page table.
  - CR3 is part of the context and needs to be saved in the PCB during a context switch.
- x86 page table:
  - Dictionary with 2^{20} entries mapping linear to physical addresses.
  - Problem: if one entry is 4 bytes, this dictionary would occupy 4M of continuous memory!
  - Solution: Two-level page table structure.

Two-level Page Table Structure



Page Directory/Table Entry Format

- P: Present bit
- R/W: Read/Write bit. If 0, page can be read and executed but not written to.
- U/S: User/Supervisor bit. If 0, page can only be accessed in supervisor privilege level.
- A: Access bit. If 1, page has been accessed. CPU will never set this bit to 0.
- D: Dirty bit. If 1, page has been written to.
- AVL: Available for use by an operating system.

Global and Local Page Tables