Memory Basics

CIS-4020
Vermont Technical College
Peter C. Chapin

References

- *Linux Kernel Development*
  - Chapters 12, 15, and 16.
- Excerpt from *Programming the 80386*
  - Passed out in class.
  - Describes the paging system of the x86 architecture.
- IA-32 Documentation from Intel
  - On the class web site (under Linux Kernel Information)
  - Note especially the *System Programming* document, Chapters 3 and 4.
- This information is **very** detailed!

IA-32

- Intel Architecture, 32 bit (80386 and up)
  - We will focus on this popular architecture.
  - Other architectures are different in many details.
  - The basic concepts are the same regardless.
- Simple CPUs have fewer issues.
  - Advanced memory management requires hardware support.
    - Simple CPUs lack this support.
    - Thus OS software has less to worry about.

Segmentation

- IA-32 supports memory segmentation.
  - We will de-emphasize this feature.
    - It is a "throw back" to the days of the 16 bit 8086.
    - Not used by most operating systems.
      - Not supported by most architectures so using it "locks" an operating system to IA-32.
  - Briefly...
    - Memory divided into variable sized "segments."
      - Size ranges from 1 byte to 4 GiB.
    - Code, data, stack, stored in separate segments.
    - Program manipulates segments expliclity
    - OS decides where segments are actually located.

Pages

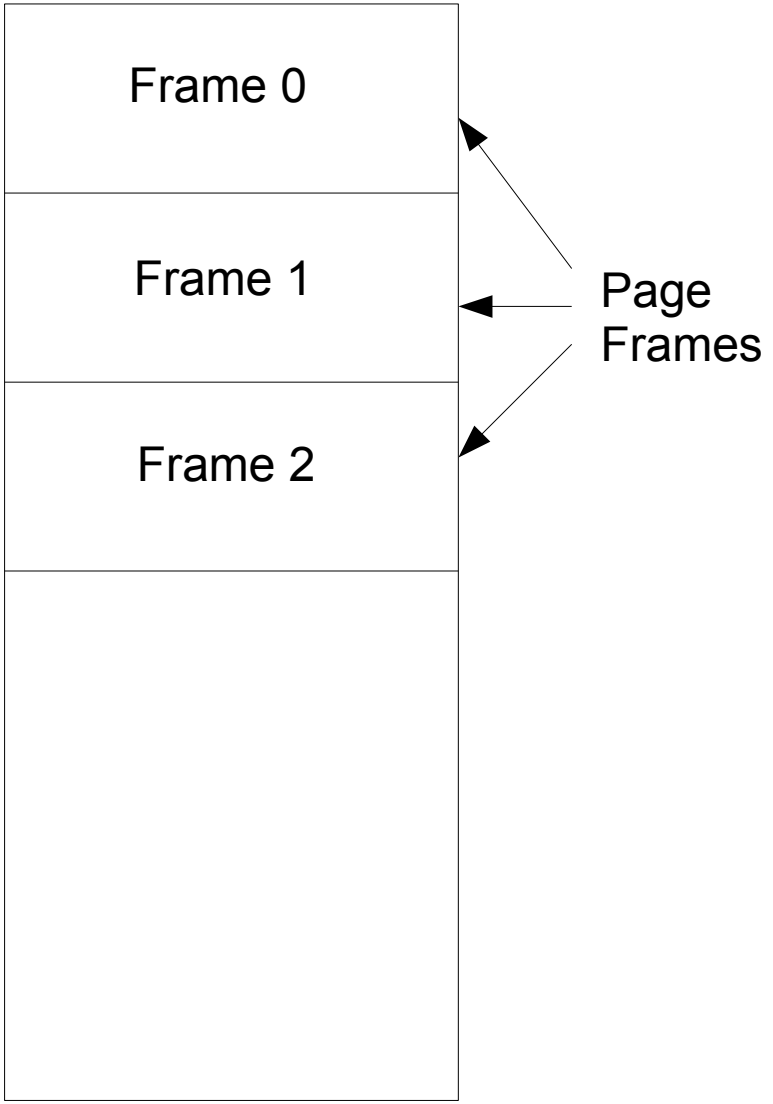
- Pages are better.
  - We will focus on this.
    - Linux, Windows, etc, use paging almost exclusively.
    - Concept supported by all advanced processors.
  - Briefly...
    - Memory divided in equal sized pages.
      - Size often fixed. It is 4 KiB on IA-32
        - Later IA-32 devices also have special "large" pages of 4 MiB.
      - Different architectures have different page sizes.
      - Some architectures allow the OS to configure the page size.
      - The size is always a power of two!

Physical Memory

- The memory you actually buy.
  - Memory that stores data and consumes power.
    - And costs money.
  - On IA-32 physical memory addresses are 32 bits.
    - Limits physical address "space" to 4 GiB.
      - Later IA-32 processors allow 64 GiB physical memory.
      - OS must use special methods to access it. Programs don't know.
    - Upper 20 bits of the address is the "frame number."
    - Lower 12 bits of the address is the page offset.

Physical Memory Organization

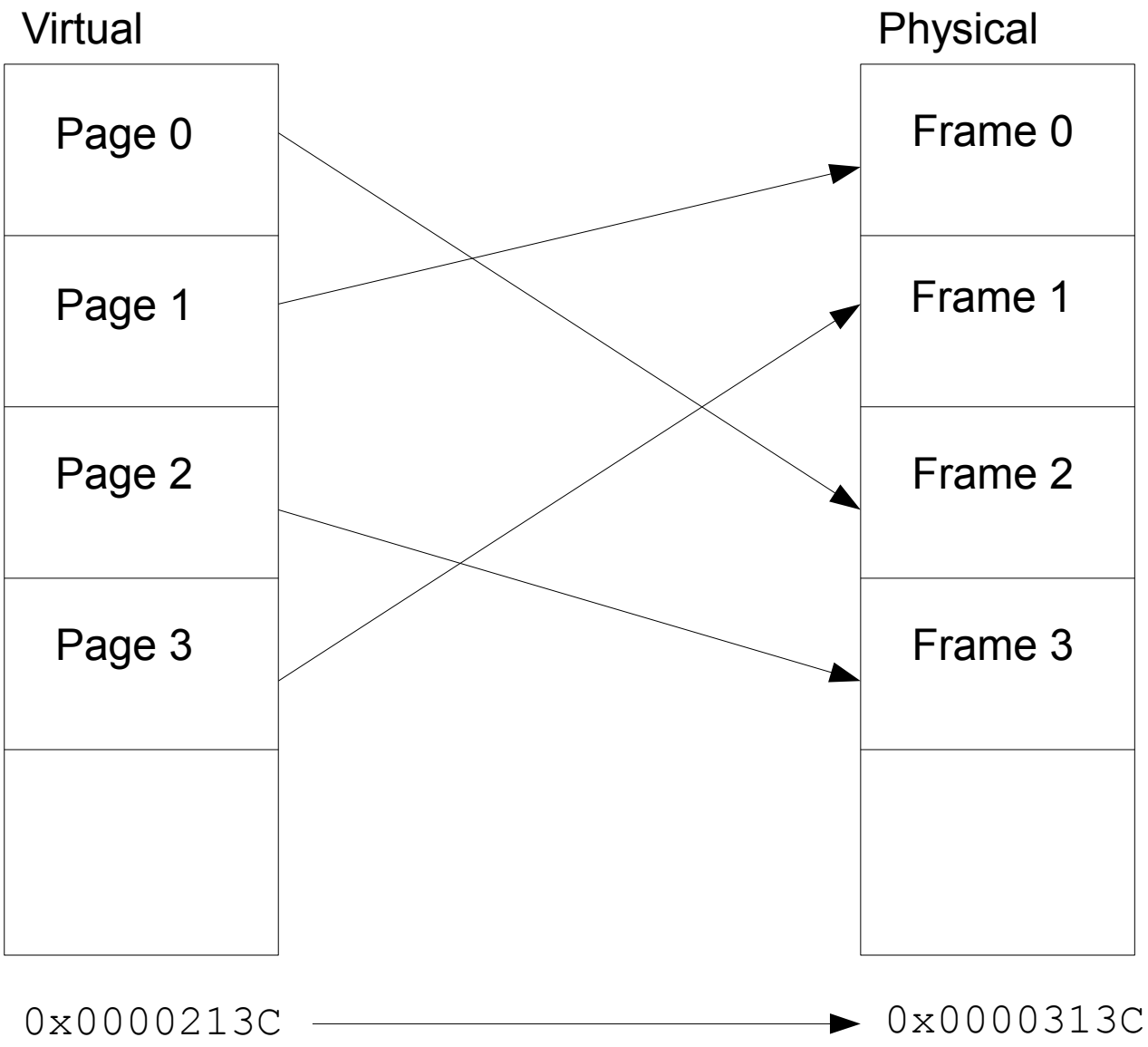
- Each frame is the same size (4 KiB)
- Offsets run from 0x000 to 0xFFF.
- Frame #0 starts at address: 0x00000000
- Frame #1 starts at address: 0x00001000
- Frame #2 starts at address: 0x00002000
- etc...



Virtual Memory

- The memory your programs think they have.
  - Memory that your programs use and address.
  - On IA-32 virtual addresses are also 32 bits (normally)
    - Virtual address space limited to 4 GiB as well.
      - This is per process. On a machine with > 4 GiB of physical memory the OS can allocate 4 GiB to multiple processes at once.
    - Virtual addresses are divided into page numbers (20 bits) and page offsets (12 bits).
      - Note distinction: "page number" vs "page frame number."
      - *A frame is physical concept. A page is a virtual concept.*
      - Many people do not bother distinguishing them.

Memory Mapping



Memory Mapping

- From Virtual to Physical
  - Each virtual page maps to a physical frame.
    - Page number replaced by frame number.
    - Page offset remains the same.
    - 0x0000213C...
      - Page 2, offset 0x13C
    - ... maps to 0x0000313C
      - Frame 3, offset 0x13C
  - Central job of the OS memory manager:
    - Maintain tables that define these mappings.
    - Different for each process!

Page Directory

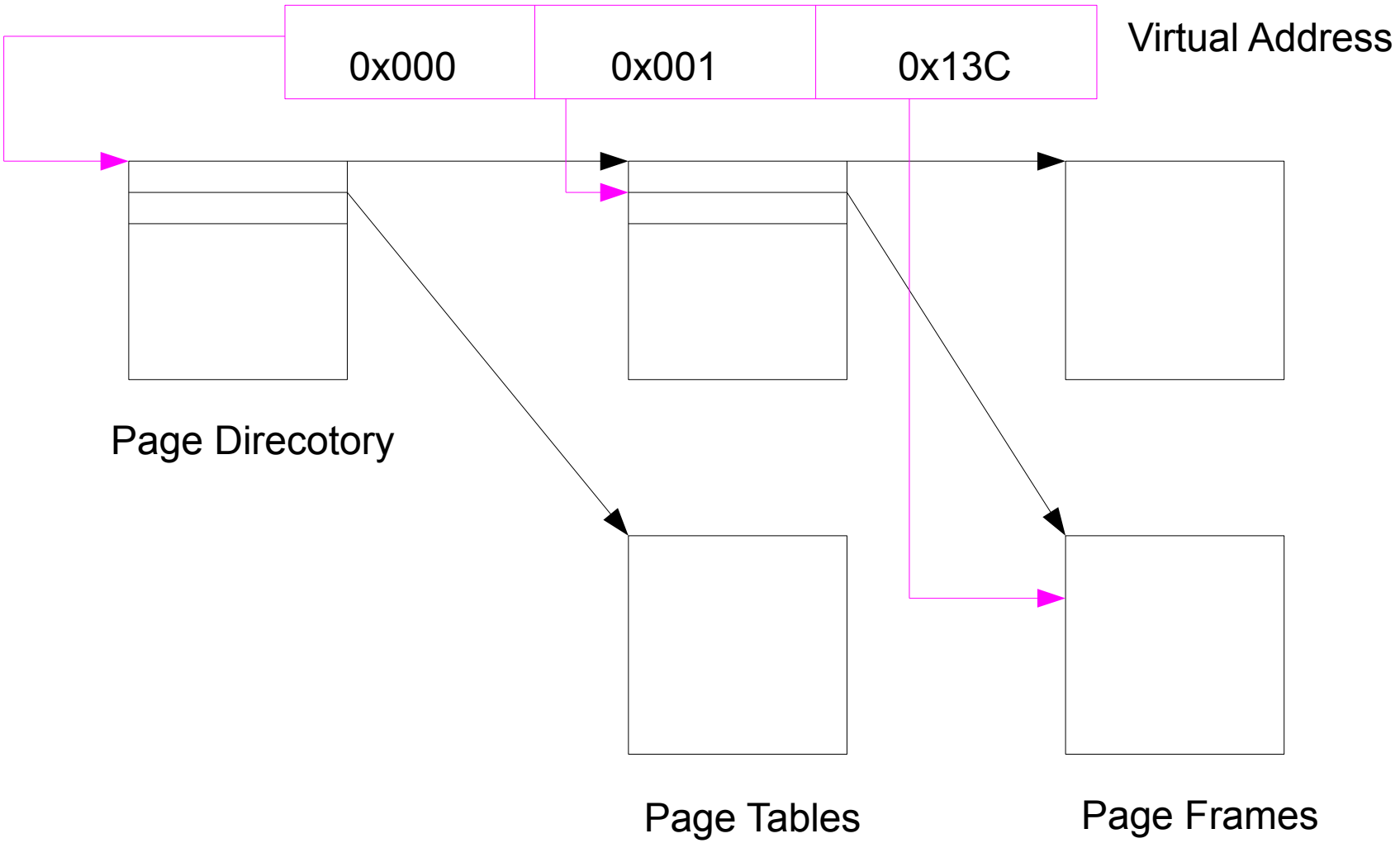
10 bit directory index	10 bit page table index	12 bit page offset

- Page directory is one page (4 KiB)
  - Contains 1024, 4 byte entries.
  - Top 10 bits of virtual address specifies directory entry.
  - Upper 20 bits of directory entry specifies a page frame where the corresponding page table is located.
  - A single page directory thus defines up to 1024 page tables.

Page Table

- Page tables are also one page (4 KiB)
  - Contains 1024, 4 byte entries
  - Middle 10 bits of virtual address specifies an entry.
  - Upper 20 bits of the entry specifies a page frame where the physical memory associated with the virtual address is located.
  - A single page table specifies up to 1024 pages (4 MiB) of program virtual address space.

Page Directories and Tables



n-Level Paging

- IA-32 uses "two level" paging
  - One page directory and then page tables.
- Other CPUs do other things.
  - Alpha processors use three level paging.
    - Top level directory, then subdirectories, then page tables.
    - Necessary because addresses are 64 bits
      - Also uses 8 KiB pages, requires 13 bit page offsets.
    - Arrangment consumes 3*10 + 13 = 43 address bits.
  - Quiz: *What does x86_64 do?*

Efficiency?

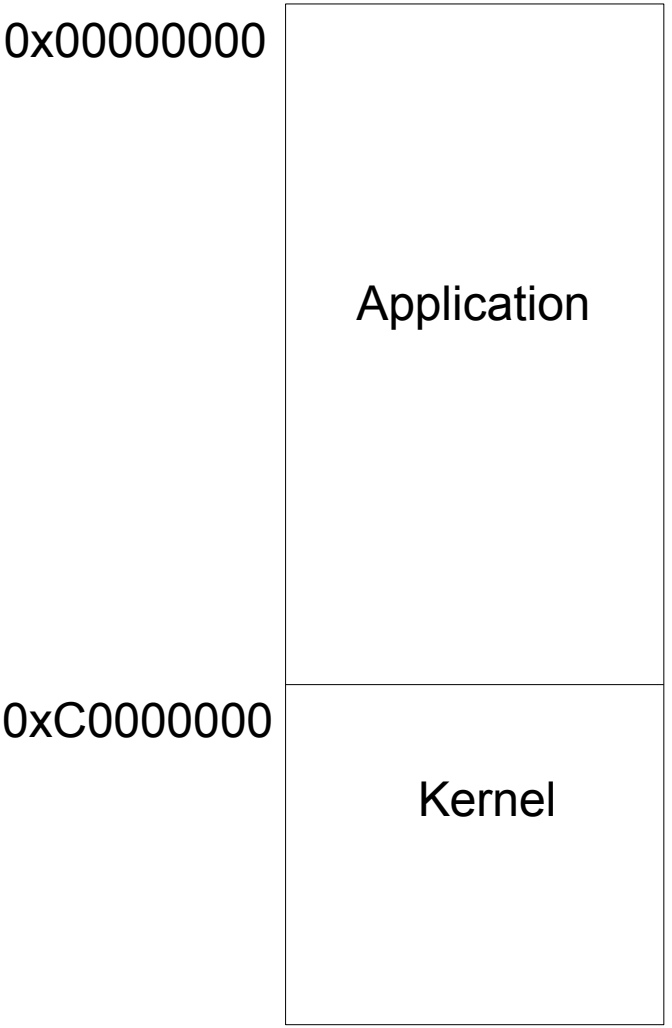
- Each memory access must go through page translation. Isn't that horribly slow?
  - Page directory and page tables are located in memory.
  - One virtual memory access causes 3 real memory operations?
- Not so bad...
  - Translations are cached in the CPU
    - "Translation look-aside buffer" (TLB).
    - Yes, there is overhead as TLB is first filled.
    - During normal operation most translations are satisfied by the TLB and execution is fast.

Where is the Page Directory?

- To start the translation process, the processor needs to know the location of the page directory
  - Physical page frame holding page directory is pointed at by a processor register.
  - When a context switch occurs the register is reloaded.
    - Each process can have its own virtual mappings.
    - Processes often share mappings by sharing page tables in their respective page directories.
  - Many "tricks" are possible.

Basic Linux Memory Map

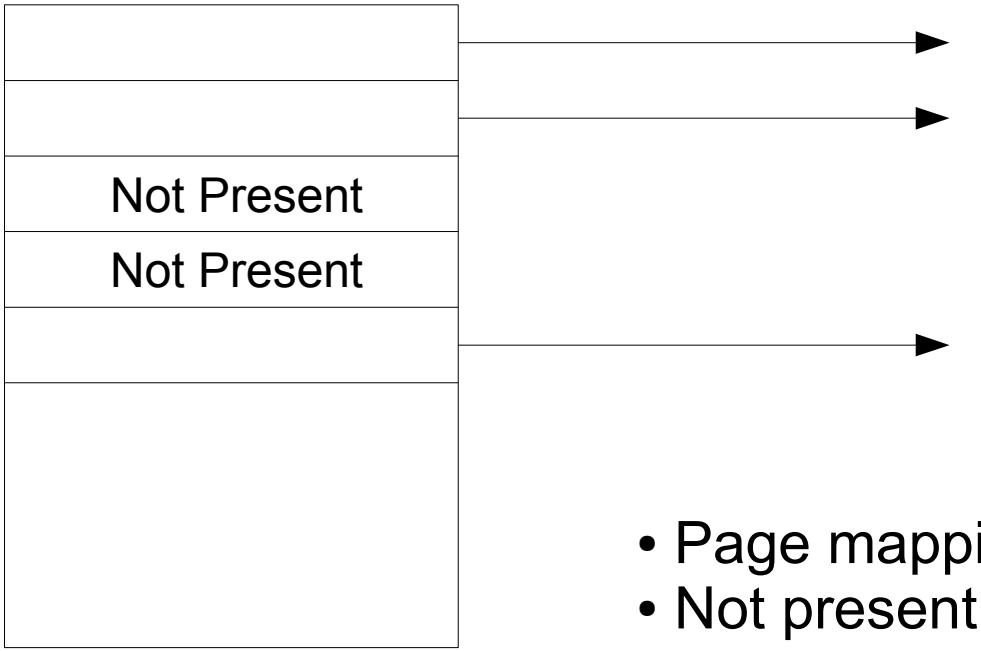
- Kernel mapping...
  - Starts at 0xC0000000
  - Same for all processes
  - Contains image of physical memory
    - Quiz: *What about large memory machines?*
- Application mapping...
  - The rest of memory
  - Remapped for each process (except for shared memory segments).



Not Present

- 1024 page directory entries
  - Implies 1024 page tables (4 MiB total)
  - Implies 1024*1024 pages (4 GiB total)
- Not all processes need that much memory!
  - Each table entry is 32 bits.
    - ... but only 20 bits are needed to hold a frame number.
    - What of the other 12 bits?
  - One of those bits is the "Not Present" bit.
    - When set it means the translation is *invalid*

Not Present



- Page mapping can have holes.
- Not present directory entries invalidates an entire 4 MiB of address space.
- Not present page table entries invalidates just 4 KiB of address space.

Where are Not Present Pages?

- They might not exist at all.
  - An attempt to access them generates a page fault.
    - Operating system function invoked automatically.
    - Sends a SIGSEGV ("segment violation") signal to the program.
    - Program is usually booted out of memory.
- They might exist on disk.
  - An attempt to access them generates a page fault.
    - Operating system function locates page and swaps it into physical memory.
    - Program continues where it left off!

Common Starting Address

- It is possible (common) to map processes so they start at the same virtual address.
  - Physical memory is distinct.
- Often address zero is left unmapped
  - NULL pointer dereference thus causes a page fault.
    - A useful testing/debugging feature.

Shared Memory

- The same page frames can be mapped into different address spaces.
  - Can be mapped to the same address in both
    - Allows consistent pointers in both processes.
      - Consider linked data structures.
      - Consider shared code libraries (call addresses, etc)
  - Can be mapped into different addresses in both
    - More flexible since a particular range of addresses might already be allocated in one of the processes.

Shared Code

- Two processes executing the same program can share a single image of the code.
  - Processes must have separate data pages (since they are independent)
  - Saves memory
  - Improves cache performance
  - BUT... disallows self-modifying programs.
  - *Note that the shared pages are marked as "read-only."*

Fork

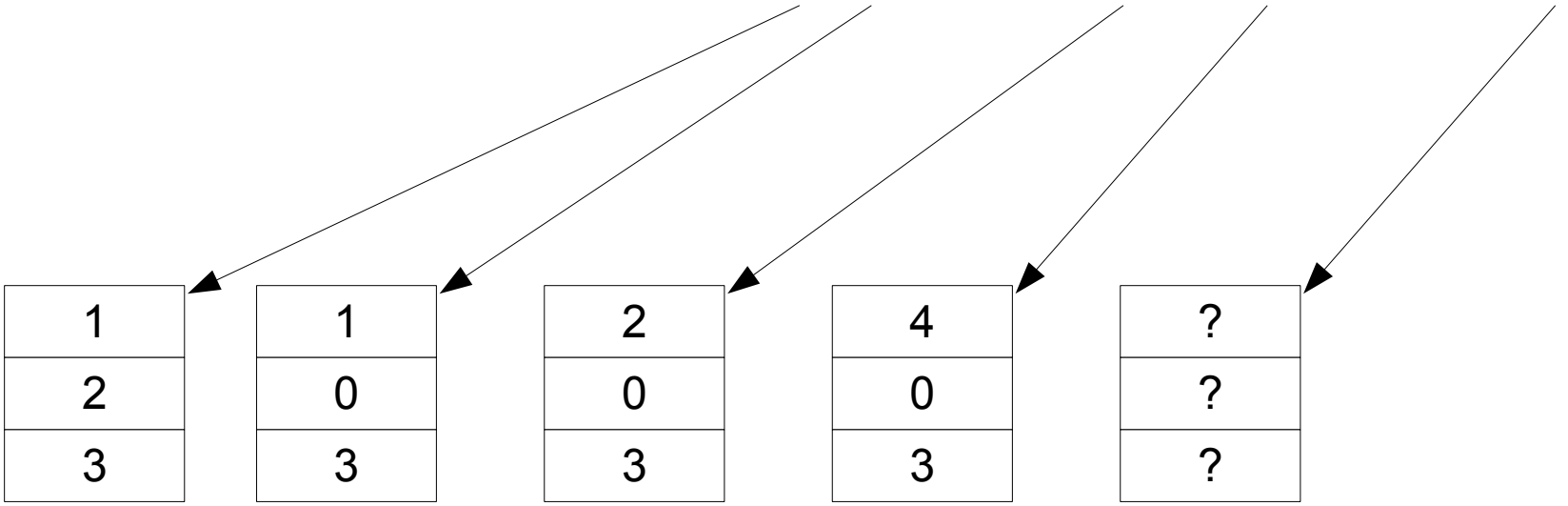
- When a process forks, the page tables of the child refer to the same page frames as used by the parent.
  - Pages all marked as read-only.
    - When either partent or child writes to them a page fault occurs.
    - OS makes a copy of the page, modifies both parent and child page tables, and marks pages read/write.
- Only the memory that is actually modified is copied, and then only when necessary.
    - *Called "copy-on-write" (COW)*

Page Replacement

- When a "not present" page is brought in from disk, an existing page must (often) be swapped out.
  - How do we decide which page to swap? There are several methods
    - Optimal
    - FIFO
    - LRU
    - Clock

Optimal Page Replacement

- *Replace the page that will be next referenced the furthest in the future.*
  - Example: 3 frames, 8 pages.
    - Reference string 1, 2, 3, 0, 1, 3, 1, 2, 3, 4, 0, 0, 3, 1...

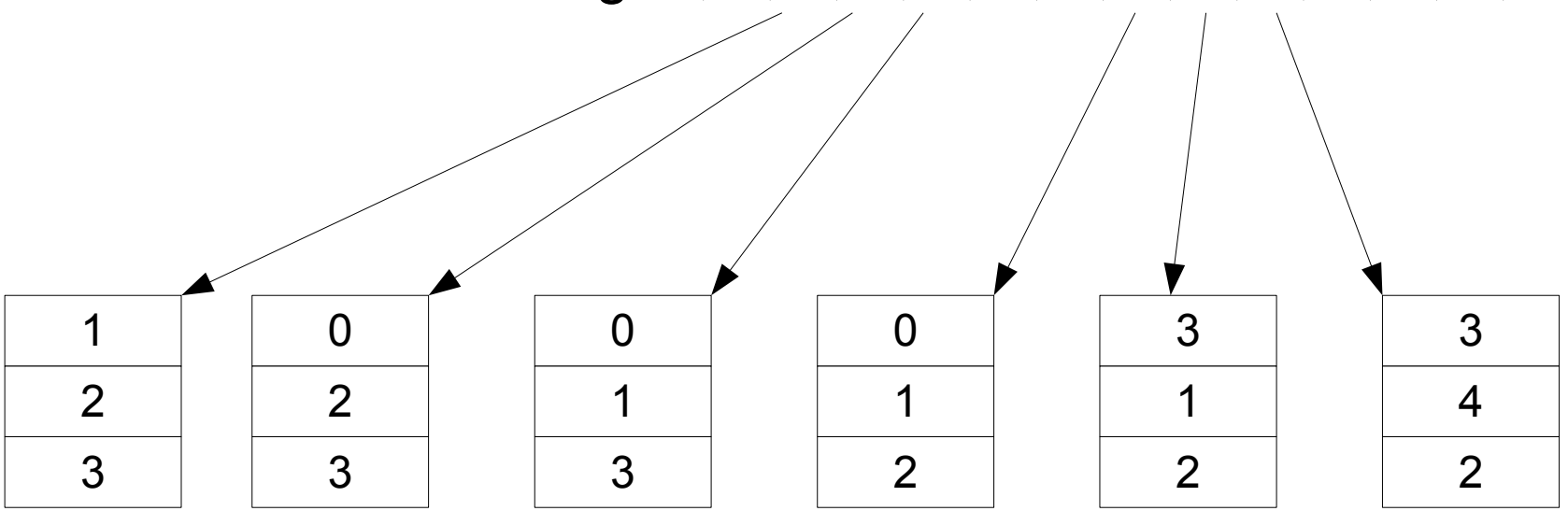


Optimal is Optimal

- It can be proved that the optimal algorithm minimizes the number of page faults.
  - But... it can't be implemented. It requires knowledge of the future.
  - Used as a basis of comparison to other page replacement methods.

FIFO

- *Replace the pages in the order in which they were loaded.*
  - Example: 3 frames, 8 pages
    - Reference String: 1, 2, 3, 0, 1, 3, 1, 2, 3, 4, 0, 0, 3, 1...

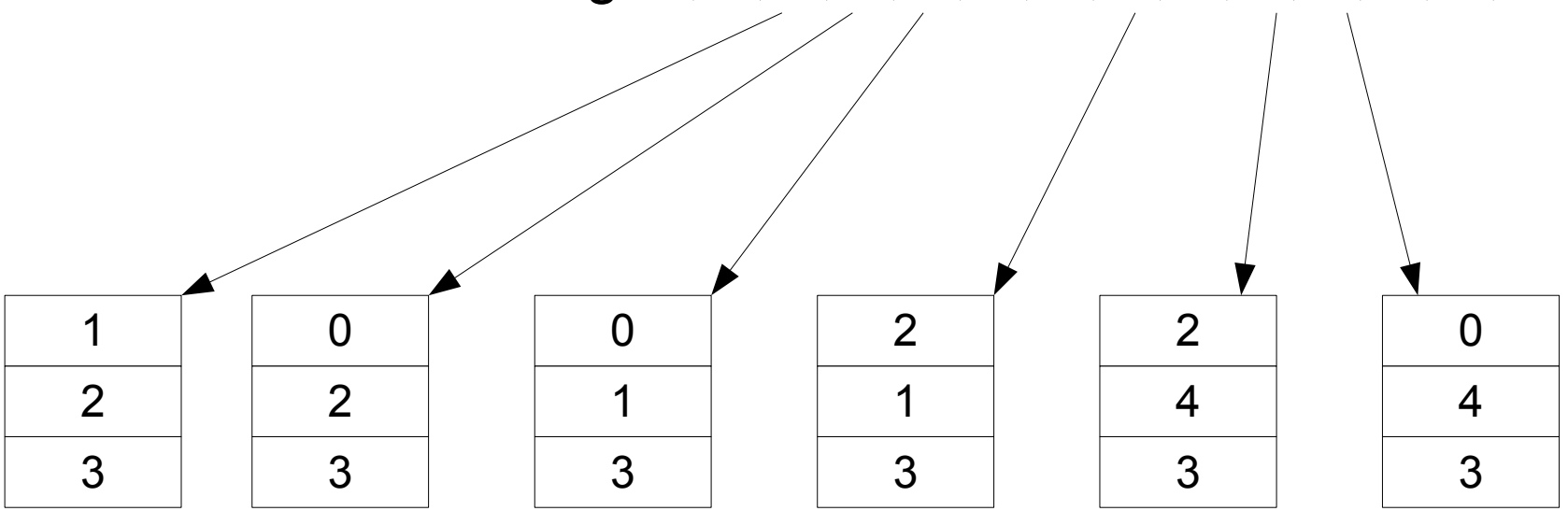


FIFO is Easy

- FIFO is easy to implement
  - Has low overhead
- BUT... FIFO is not very good.
  - Many page faults

LRU

- Least Recently Used: Replace the page that was referenced the longest in the past.
  - Example : 3 frames, 8 pages
    - Reference string: 1, 2, 3, 0, 1, 3, 1, 2, 3, 4, 0, 0, 3, 1...



LRU Works, But...

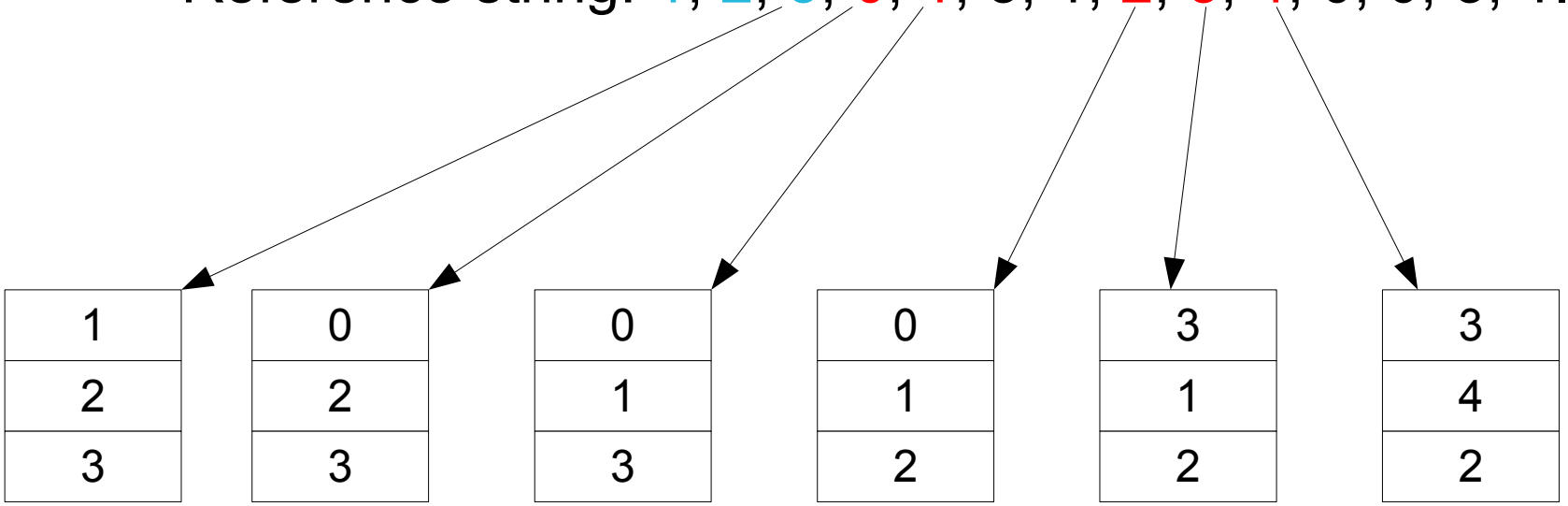
- LRU does well overall
  - Much better than FIFO in typical cases.
  - Reasonably close to optimal in many cases.
- BUT...
  - LRU is very difficult to implement.
    - Requires page frames be kept in a list so that the least recently used page can be easily found.
    - *List must be updated on every memory reference!*

Second Chance

- Like FIFO except that you skip over pages that have been accessed.
  - Requires hardware support:
    - Each time a page is accessed a flag is set in the PTE.
    - The OS searches linearly (FIFO) for the next frame that does not have the flag set (a frame that has not been accessed).
    - The OS clears the flag as it works. Eventually a free frame will be found (after possibly wrapping around).
  - Essentially FIFO with some LRU information.
    - Historic information simple. Easy for hardware to track.

Second Chance

- Like FIFO but modified with historic information.
  - Example: 3 frames, 8 pages
    - Reference string: 1, 2, 3, 0, 1, 3, 1, 2, 3, 4, 0, 0, 3, 1...



This example ends up being the same as FIFO; algorithm works better on larger examples