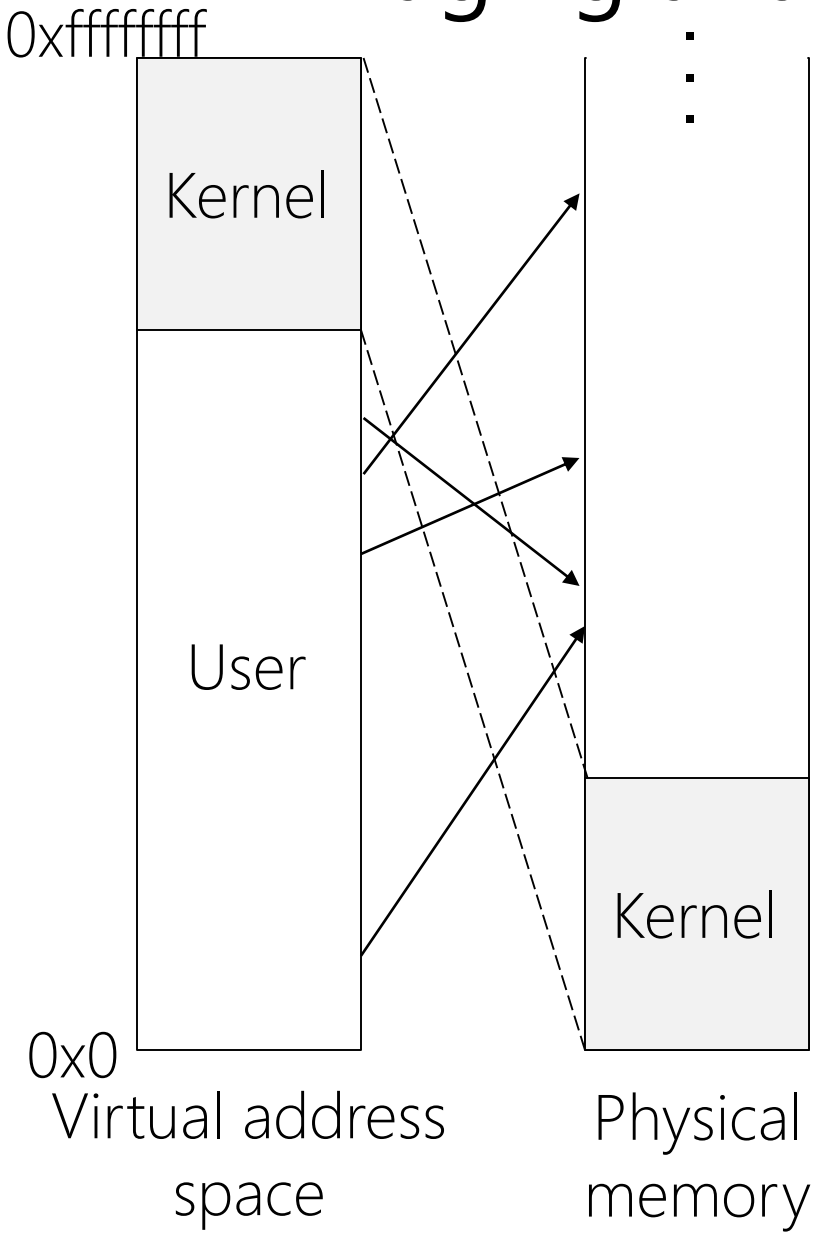
Virtual Memory, Part III

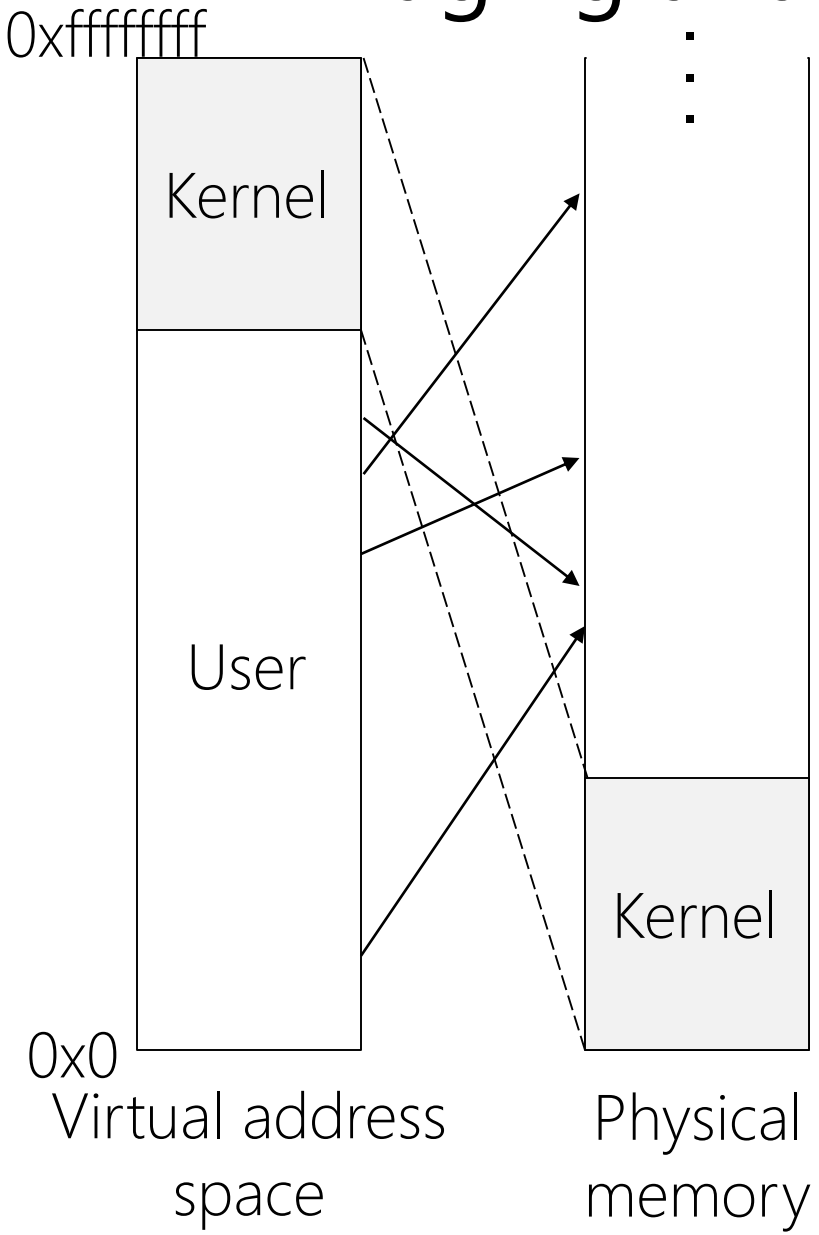
CS 161: Lecture 8
2/23/17

Kernel Memory Interactions with Paging and TLB: Linux on x86



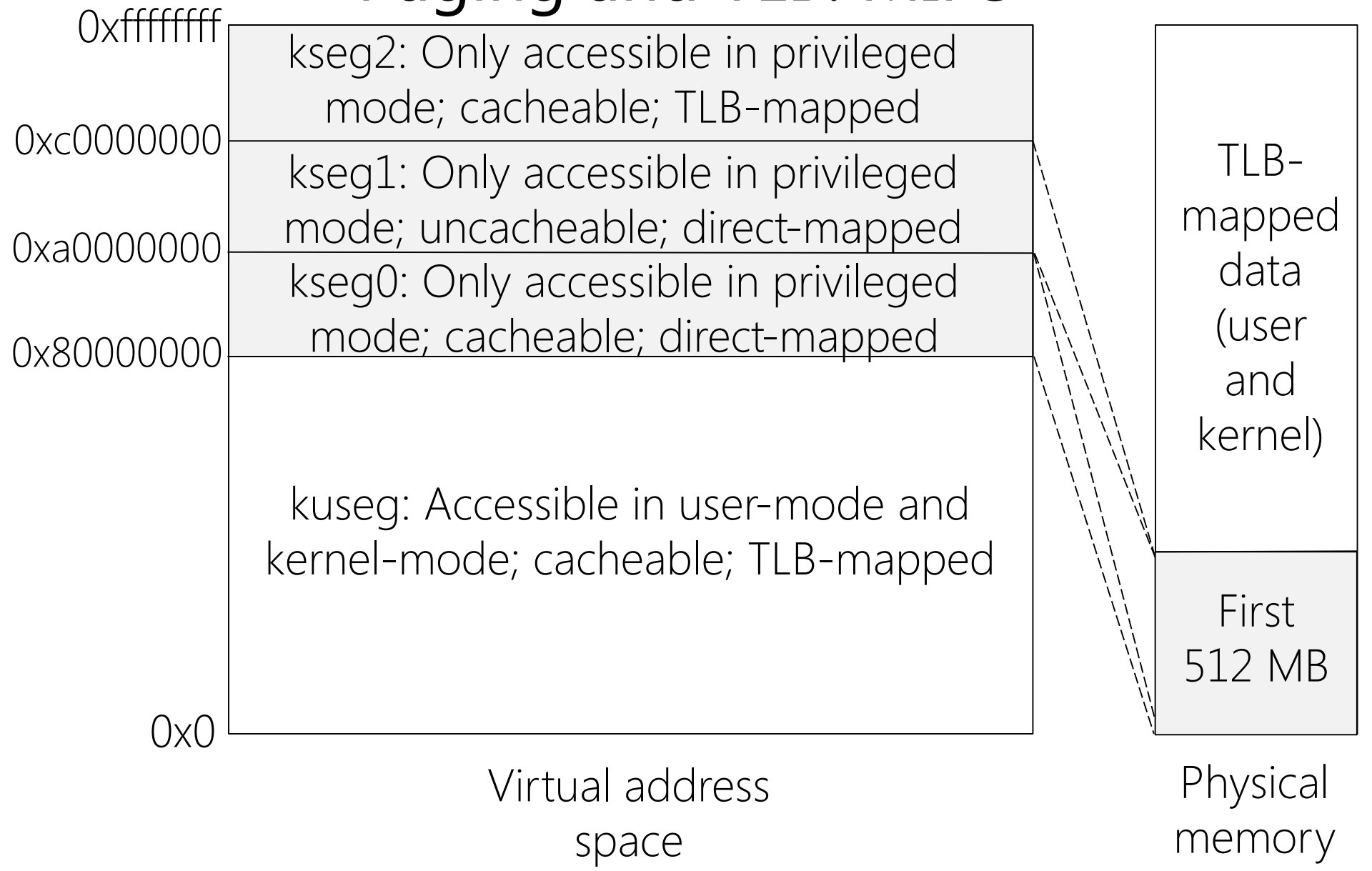
- Kernel places itself in lowest part of physical memory, but maps itself into highest part of each process' virtual address space
- Bottom part of process' address range:
  - 0x00000000—0xbfffffff are accessible by both user-mode and kernel-mode
  - Address range covered by first 768/1024 of the process' page directory; represents unique, per-process pages
- Top part of process' address range:
  - 0xc0000000—0xffffffff can only be accessed in kernel-mode
  - Address range covered by last 256 entries of page directory; "user-mode accessible?" bit is 0 in PDEs, so user-level code can't access those memory addresses!

Kernel Memory Interactions with Paging and TLB: Linux on x86



- All virtual memory addresses (include ones to kernel memory) go through TLB
  - Reference to kernel memory can cause TLB miss, but not page fault—kernel is always resident in RAM!
- A system call or interrupt doesn't require changing %cr3—kernel is mapped into address space already, but now the CPU runs in privileged mode, so high virtual memory addresses can be accessed!

Kernel Memory Interactions with Paging and TLB: MIPS



WHY ARE THERE SO MANY DIFFERENT SCHEMES?

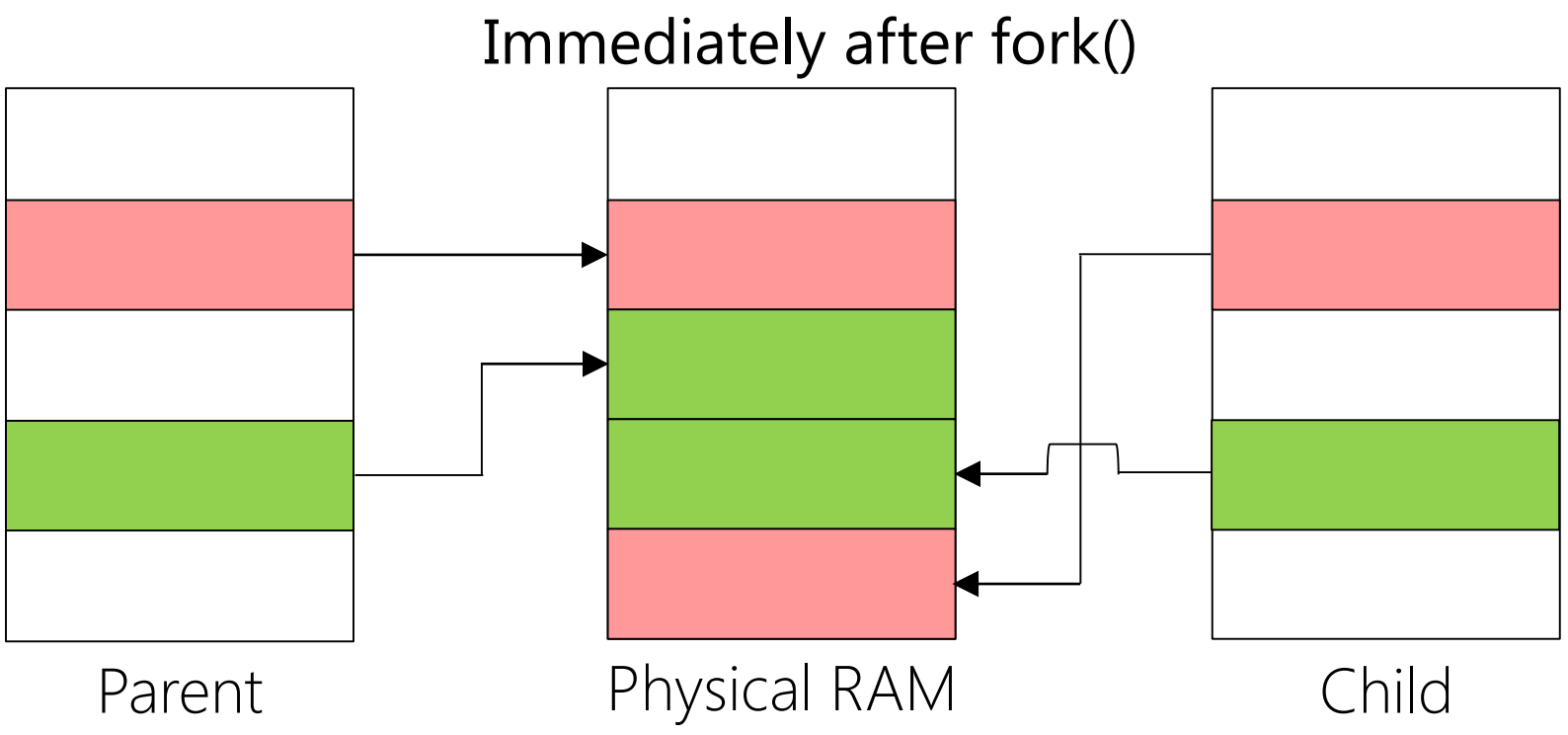
"The use of one memory management organization over another has not catapulted any architecture to the top of the performance ladder, nor has the lack of any memory management function been the leading cause of an architecture's downfall. So, while it may seem refreshing to have so many choices of VM interface, the diversity serves little purpose other than to impede the porting of system software."

B. Jacob and T. Mudge, "Virtual Memory in Contemporary Microprocessors." In *IEEE Micro*, Volume 18, Issue 4, July 1998.

Important job of OS: Hide processor quirks using abstraction!

Making fork() Efficient

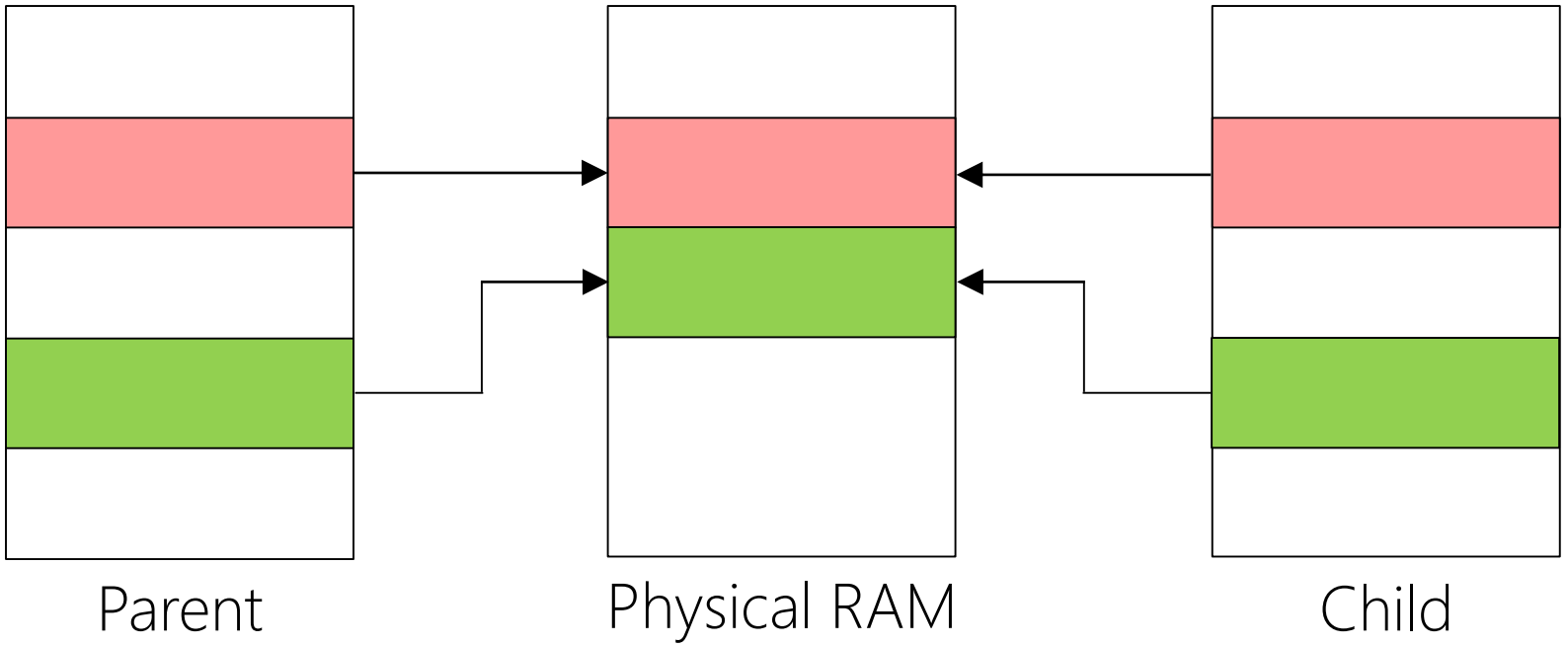
- fork() creates a new child process whose address space is a copy of the parent's address space
- Naïve implementation: synchronous copying
  - The OS gives the child a copy of the parent's page table . . .
  - . . . and then performs repeated, synchronous memcpy() operations to duplicate every allocated page in the parent's address space



Making fork() Efficient: Copy-on-Write

- OS gives the child a copy of the parent's page tables (i.e., the child gets a copy of the parent's PTEs), but the OS does not copy raw page data
- OS sets all PTEs in both page tables as "read-only"

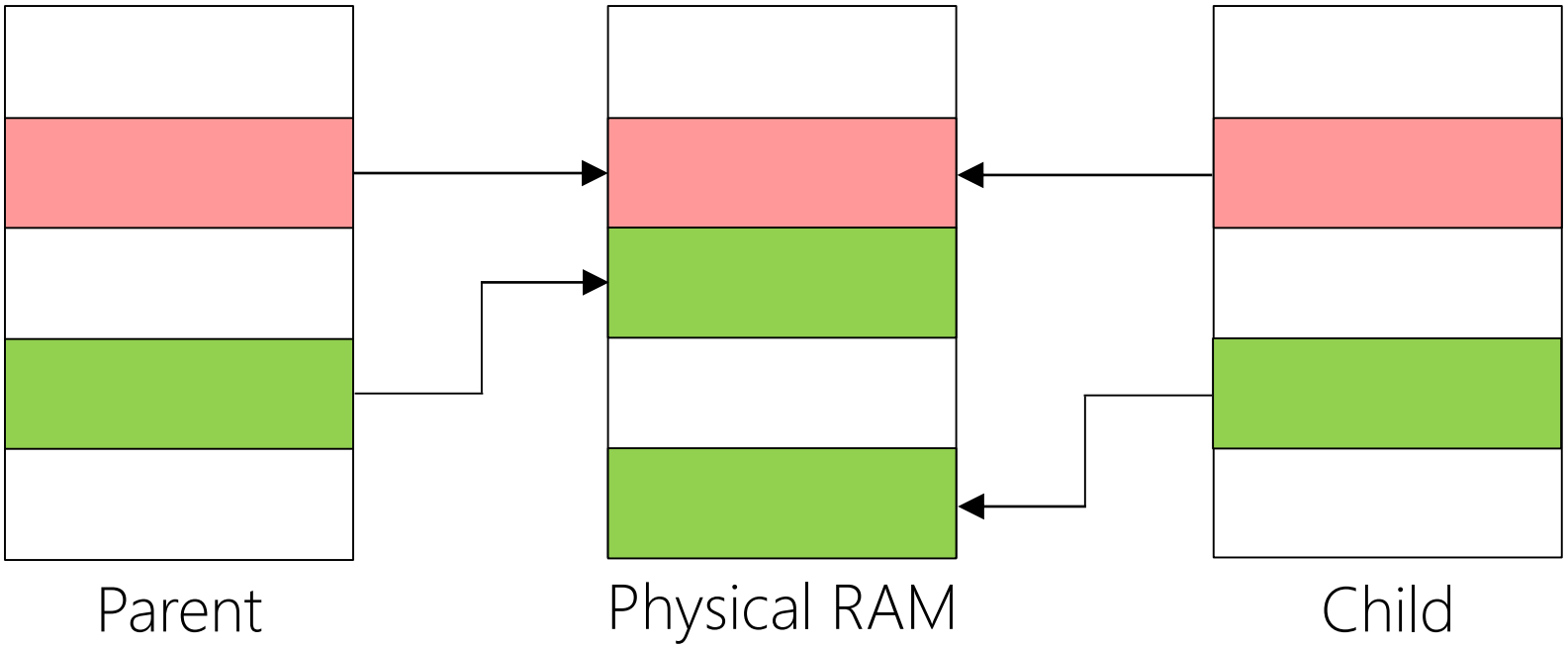
Immediately after fork()



Making fork() Efficient: Copy-on-Write

- When one of the processes tries to write a page, a trap occurs (e.g., "MOD" TLB read-only fault on MIPS), allowing the OS to:
  - create a copy of the page for the child
  - update the child's PTE to point to new page, and then
  - mark the associated PTE in both processes as RW

After fork(), after the child has modified the green page



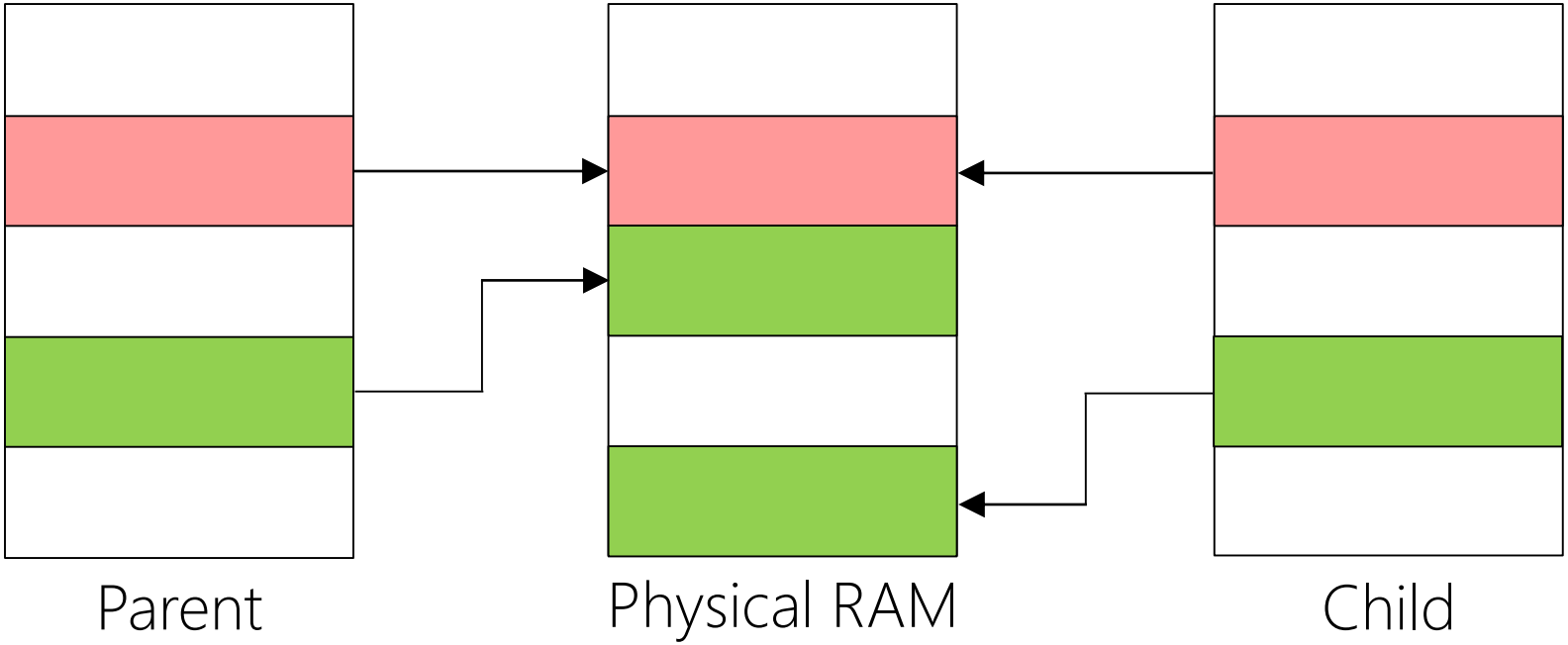
Making fork() Efficient: Copy-on-Write

Removes synchronous copy cost during fork()

Physical page sharing enables a fixed amount of RAM to contain more virtual pages

---

After fork(), after the child has modified the green page

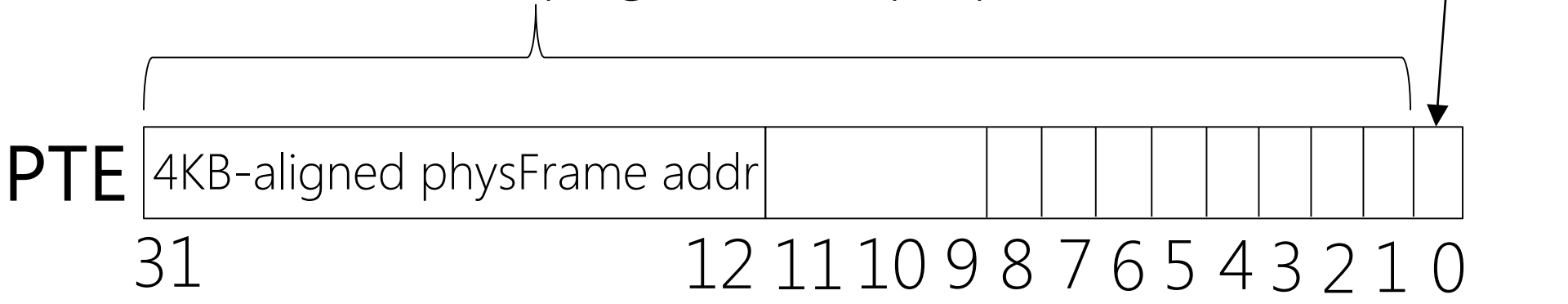


Swapping Pages Out

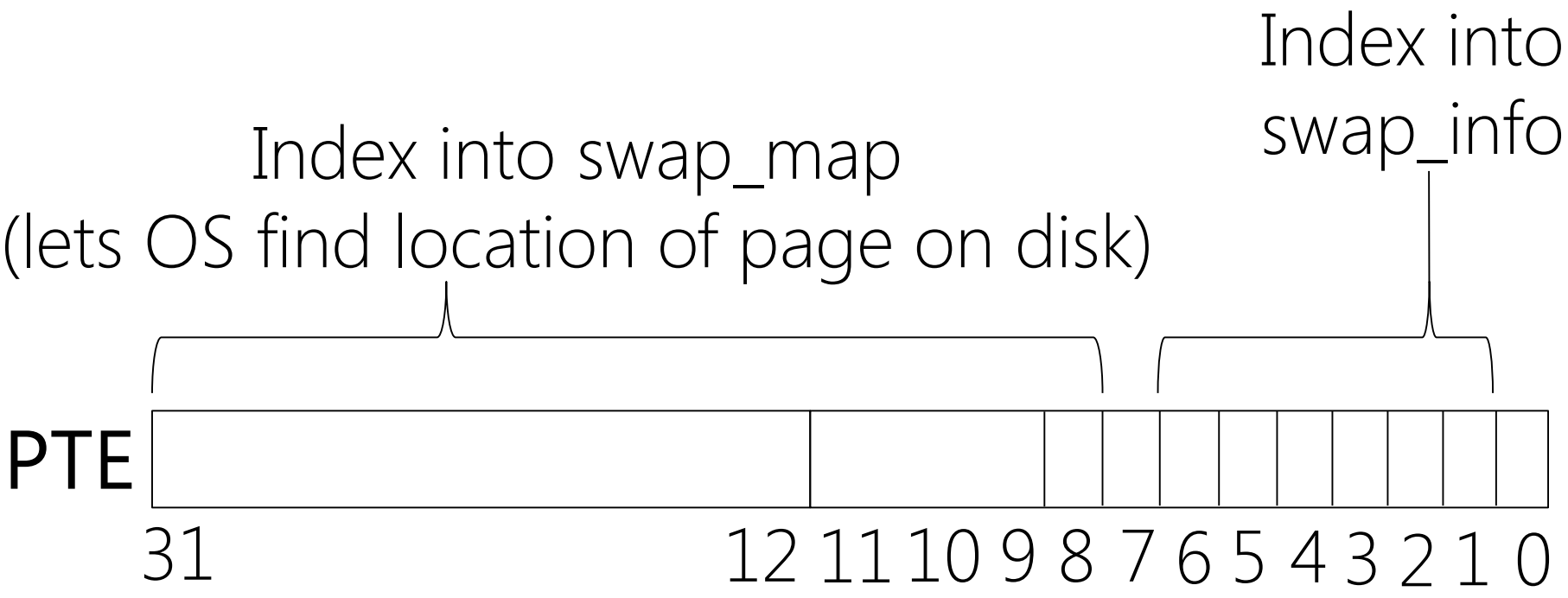
- Physical RAM may be oversubscribed: the aggregate number of pages in the active address spaces may be larger than the number of physical pages
- "Swapping" refers to the OS moving virtual pages from physical RAM to the swap device(s) and vice versa
- A swap device can be a hard disk, an SSD, or even remote network storage (<--maybe not a great idea)
- How does the OS associate a virtual page with its location in swap storage?
  - Case study: Linux on x86

```
struct swap_info_struct{
    struct file *swap_file;   /* Where the swap data lives on-disk */
    unsigned char *swap_map;  /* For each swapped-out page, stores a
                               * reference count of how many tasks
                               * use that page */
    unsigned int max;         /* Number of entries in swap_map */
    unsigned int inuse_pages; /* Number of swap entries that currently
                               * contain a virtual memory page */
    unsigned int lowest_bit;  /* index of first free element in
                               * swap_map */
    spinlock_t lock;
};
struct swap_info_struct *swap_info[MAX_SWAPFILES];
```

If Present?==0, can use this region to store location of page in swap space!



```
struct swap_info_struct{
    struct file *swap_file;   /* Where the swap data lives on-disk */
    unsigned char *swap_map;  /* For each swapped-out page, stores a
                               * reference count of how many tasks
                               * use that page */
    unsigned int max;         /* Number of entries in swap_map */
    unsigned int inuse_pages; /* Number of swap entries that currently
                               * contain a virtual memory page */
    unsigned int lowest_bit;  /* index of first free element in
                               * swap_map */
    spinlock_t lock;
};
struct swap_info_struct *swap_info[MAX_SWAPFILES];
```

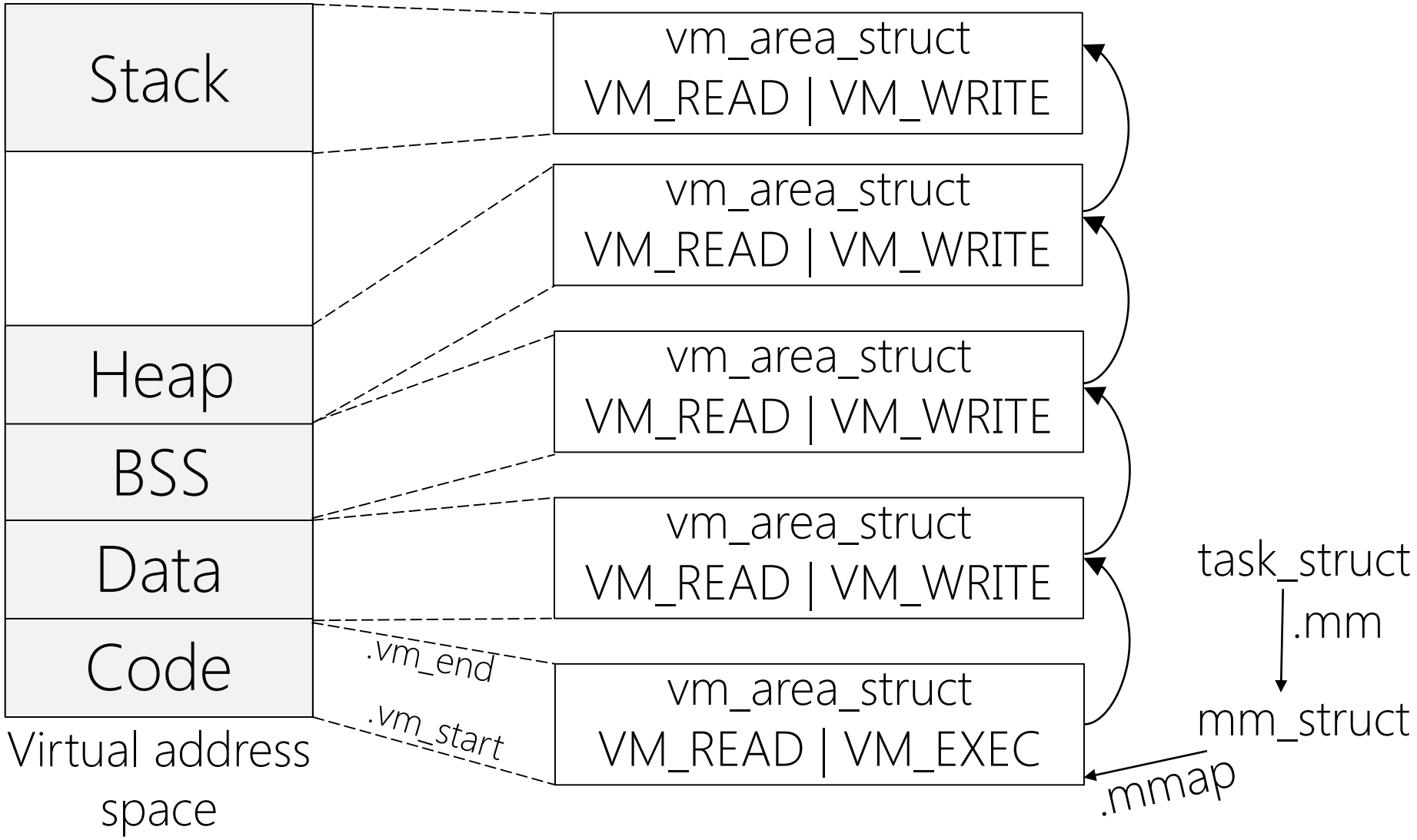


Q: How does the OS represent the logical regions of a process's address space?

Case study: Linux on x86

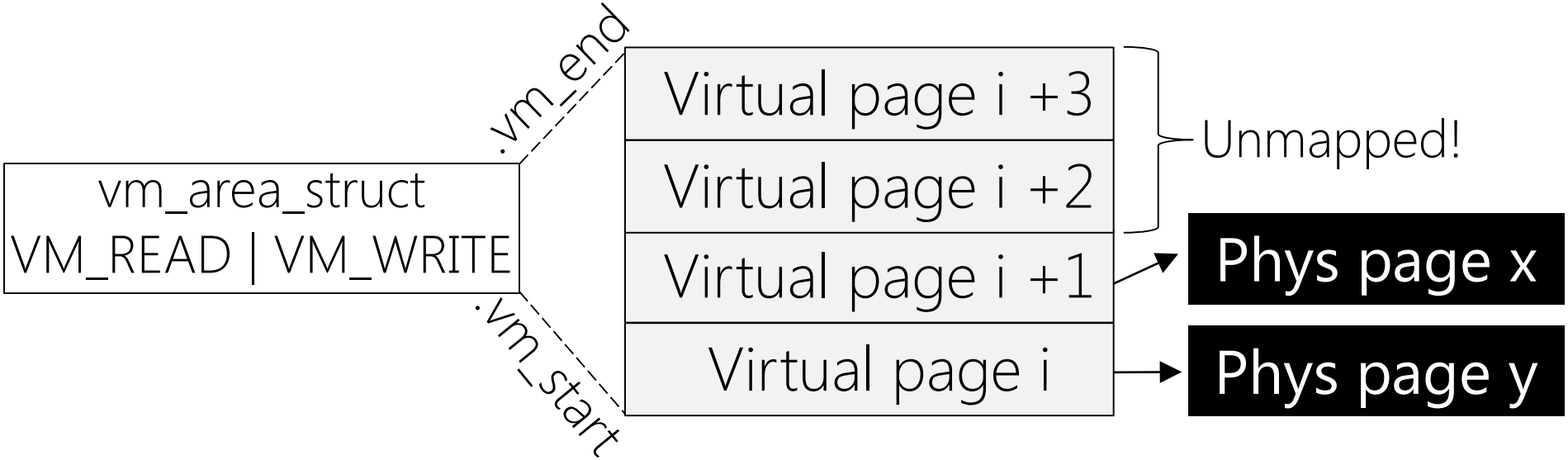
Linux Virtual Memory Areas (VMAs)

- A VMA represents a contiguous chunk of virtual memory that the OS has agreed to give to a process

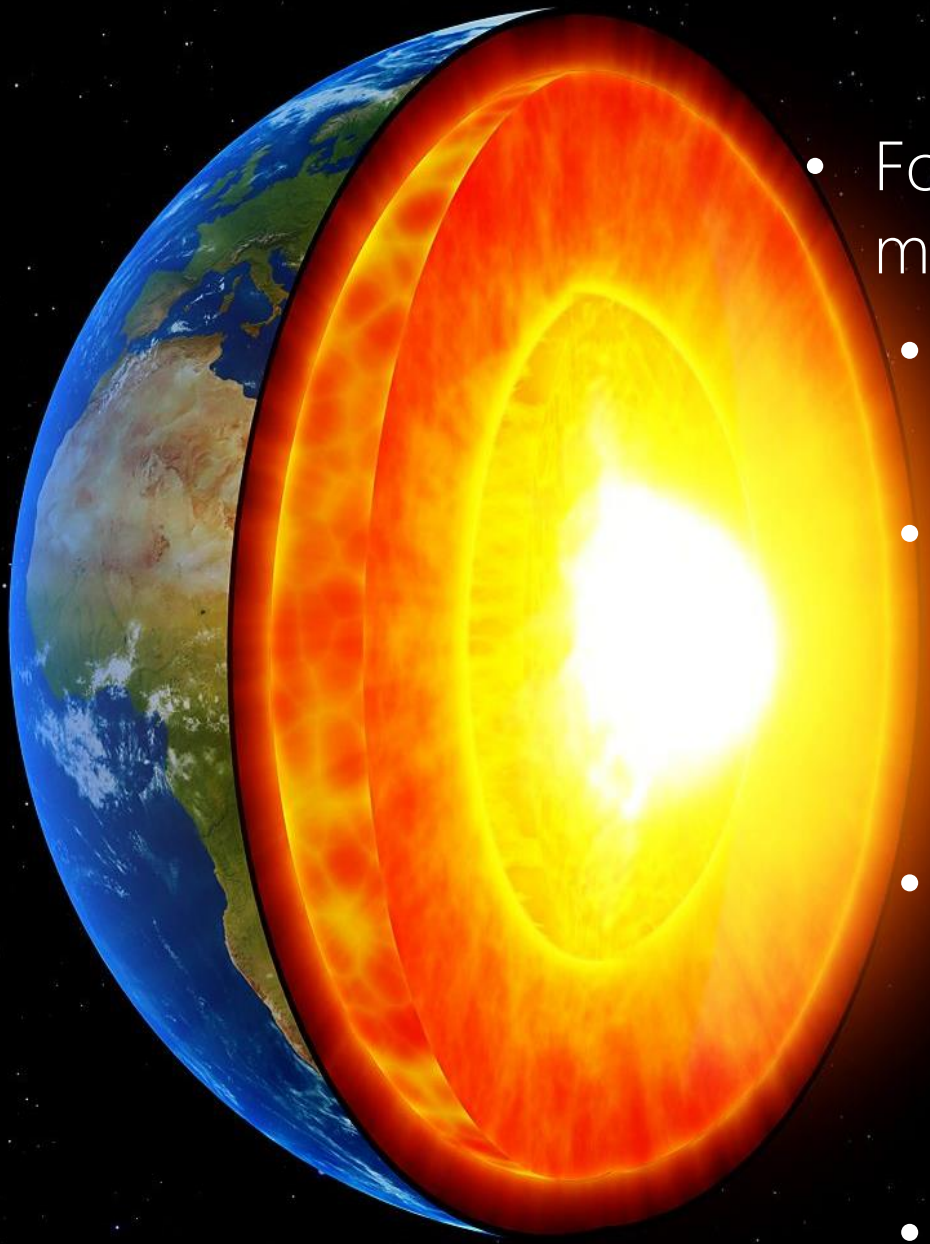


Linux Virtual Memory Areas (VMAs)

- A VMA represents a contiguous chunk of virtual memory that the OS has agreed to give to a process
  - VMAs represent the process' view of its address space . . .
  - . . . but the page table's "present?" bits indicate the actual situation with respect to backing physical memory!



- Accessing virtual page i+2 or i+3 causes a page fault
- In page fault handler, kernel sees that the faulting address resides in a valid VMA
- Handler extracts location of page in swap space by examining the PTE for the faulting address



The Core Map

- For each physical page, the core map records metadata like:
  - Is the physical page free or allocated?
  - If the physical page is allocated, which address space and virtual page are using the physical page?
  - Is the page locked in memory (e.g., because it is being written to disk, or written from disk)?
  - Other metadata

Uses For The Core Map

- The OS can mark kernel pages in the core map so that kernel pages are not assigned to user processes
- When the OS needs to map a swapped-out/new virtual page into physical memory, the OS consults the core map to see whether a free physical page is available
- If the core map tracks whether each physical page is dirty, a background kernel thread can asynchronously flush dirty pages to disk (the hope is to avoid a synchronous IO when a physical page must be evicted!)

BEWARE OF CONCURRENCY ISSUES

Demand Paging

- In most OSes, when a new program is loaded, the associated process gets:
  - A page table with all/almost all of the "valid" bits turned off
  - No/very few virtual pages in physical RAM
- As the process executes, its instructions touch virtual addresses which are not resident in physical RAM
  - Page faults induce the OS to bring the missing virtual pages into physical RAM on-demand
  - A process's working set is the set of all virtual pages that the process is currently accessing
  - If there is enough free RAM to hold a process's working set, then once the working set has been paged in, the process will not cause additional page faults . . .
  - . . . until the working set changes

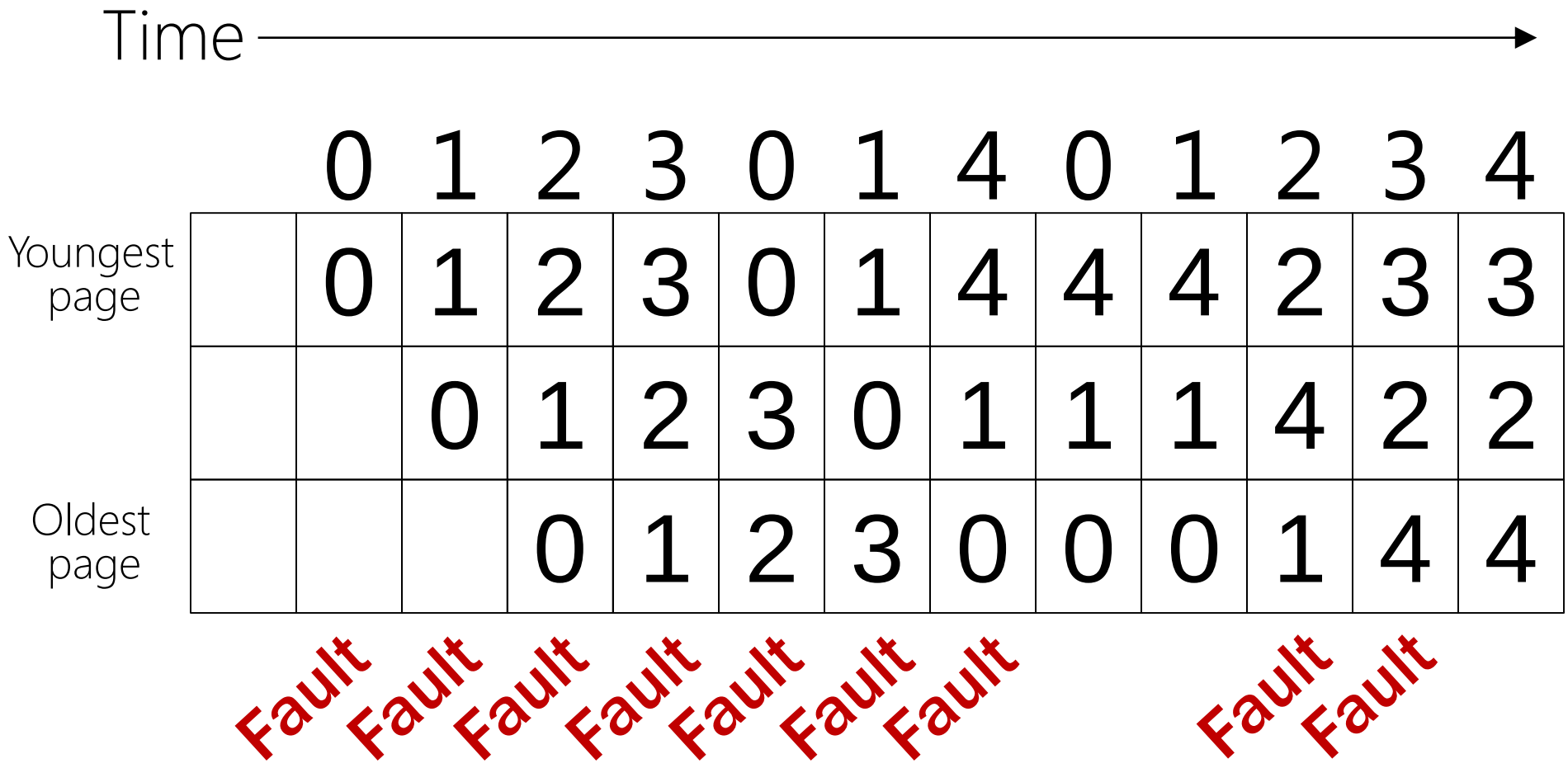
Page Replacement

- OS limits how much physical RAM can be allocated to one process
  - After that limit is reached, if the process wants to swap something in (or create a totally new page), the OS must first evict a preexisting virtual page
  - The OS uses a page replacement algorithm to select the victim page
- Belady's replacement algorithm: Evict the page that won't be accessed for the longest time
  - Good: the algorithm is provably optimal!
  - Bad: PERFECT FORECASTING IS IMPOSSIBLE UGGGGGH

Page Replacement: FIFO

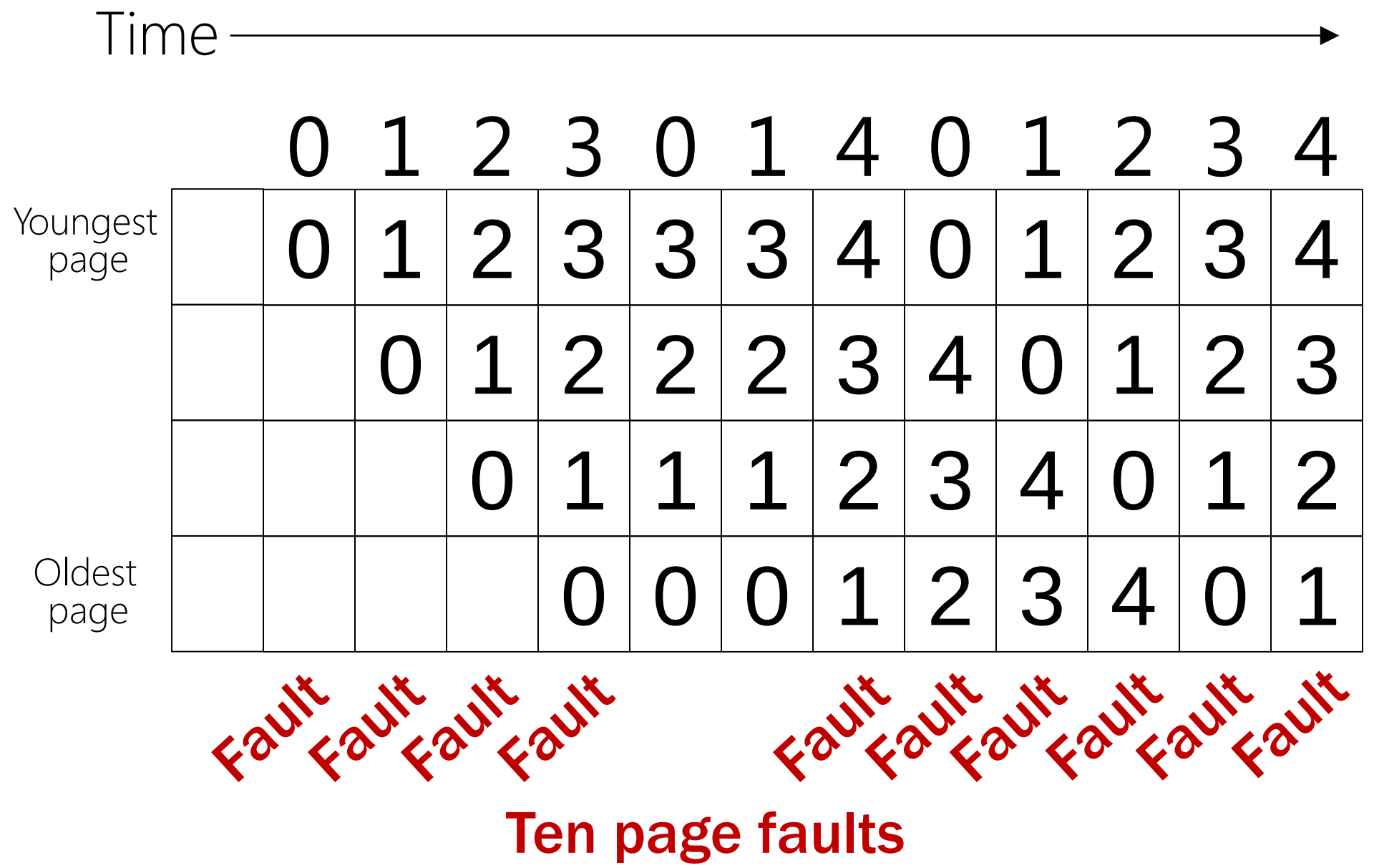
- OS maintains a FIFO queue of virtual pages
  - The queue has a maximum length of num_phys_pages
  - When the queue reaches maximum size and the OS needs to make room for a new virtual page:
    - the oldest virtual page is evicted from physical RAM
    - the new virtual page is placed in the newly-freed physical page
    - the new virtual page is added to the beginning of the queue
- The intuition is that the page that's been in RAM for the longest is no longer needed
  - This intuition may be wrong (e.g., a hot code page that is used for the entirety of a process's execution)
- FIFO also suffers from Belady's anomaly: the page fault rate might increase when FIFO is given *more* physical memory

FIFO: Three Physical Pages



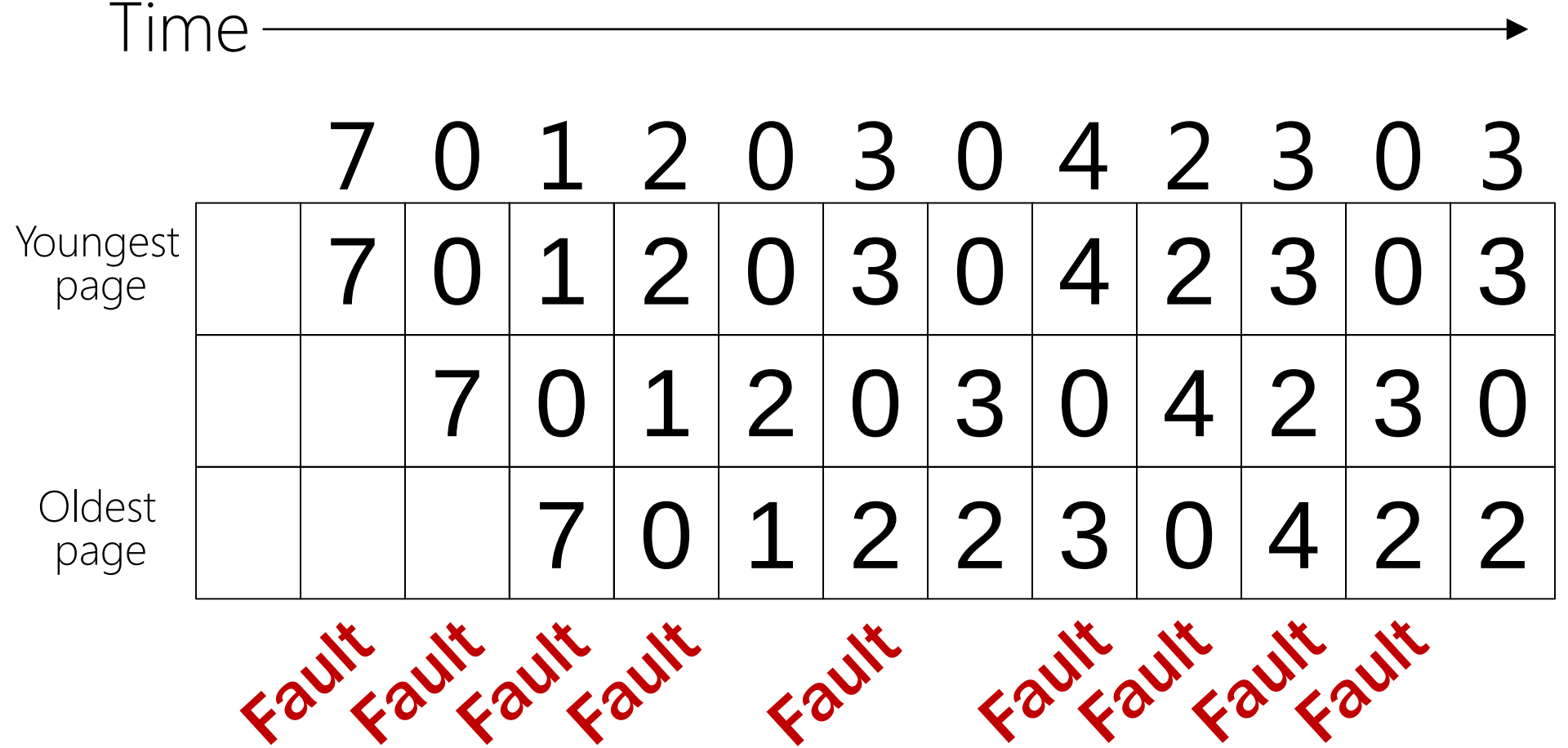
Nine page faults

FIFO: Four Physical Pages



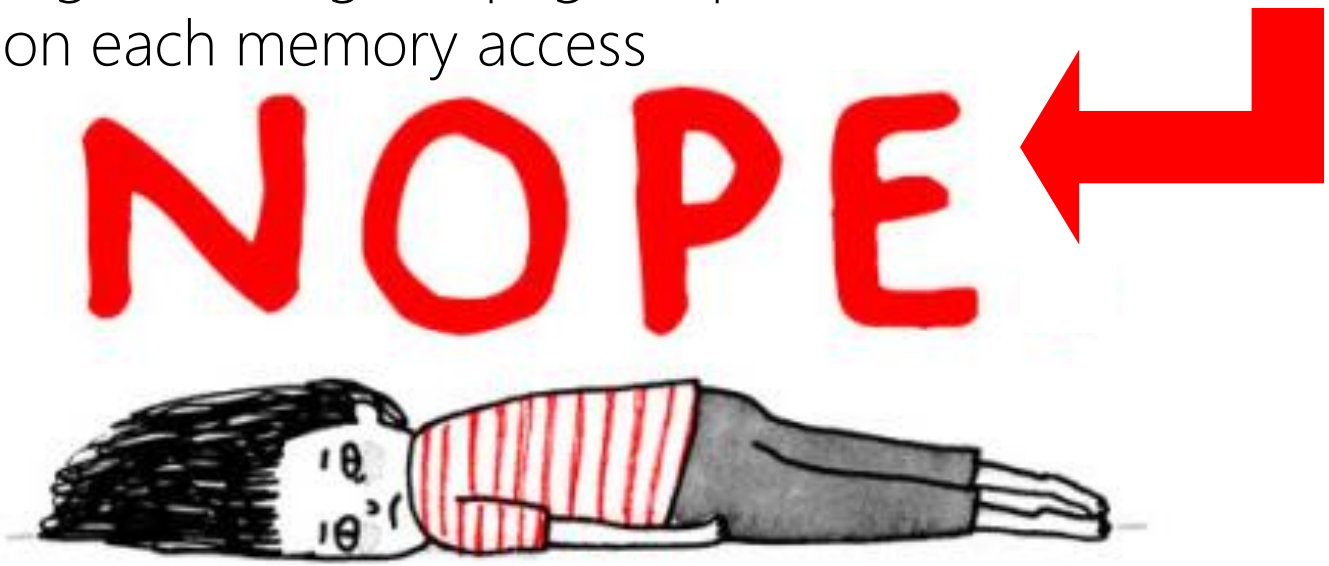
Page Replacement: Least-recently Used (LRU)

- Replace the page which hasn't been used for the longest amount of time: intuitively, if a page has been used recently, it will probably be used again soon



Page Replacement: Least-recently Used (LRU)

- LRU provides good performance, but it is difficult to implement efficiently
- Naïve implementation:
  - OS has a software-managed list of in-memory virtual pages
    - On each memory access, OS moves the accessed page to the front of the list: O(n)
    - At eviction time, OS evicts the page at the back of list: O(1)
  - Disadvantages: finding the page to promote is O(n); list must be updated on each memory access

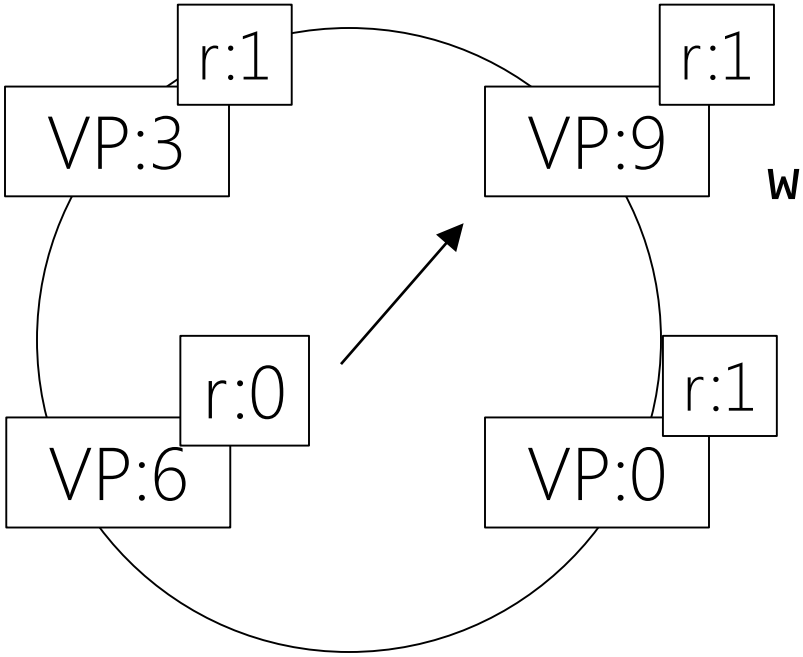


Page Replacement: Least-recently Used (LRU)

- An alternative implementation uses hardware counters (a single global counter, and per-physical-page counters)
  - Hardware increments global counter after each load or store
  - After each memory instruction, hardware also tags the relevant physical page with the global counter value
  - At eviction time, OS evicts the page with the lowest counter
  - Advantage: Per-memory-reference updates are fast if the hardware supports per-physical-page counters
  - Disadvantages: Finding the page-to-evict is O(n), and most hardware doesn't support per-physical-page counters

Page Replacement: Clock

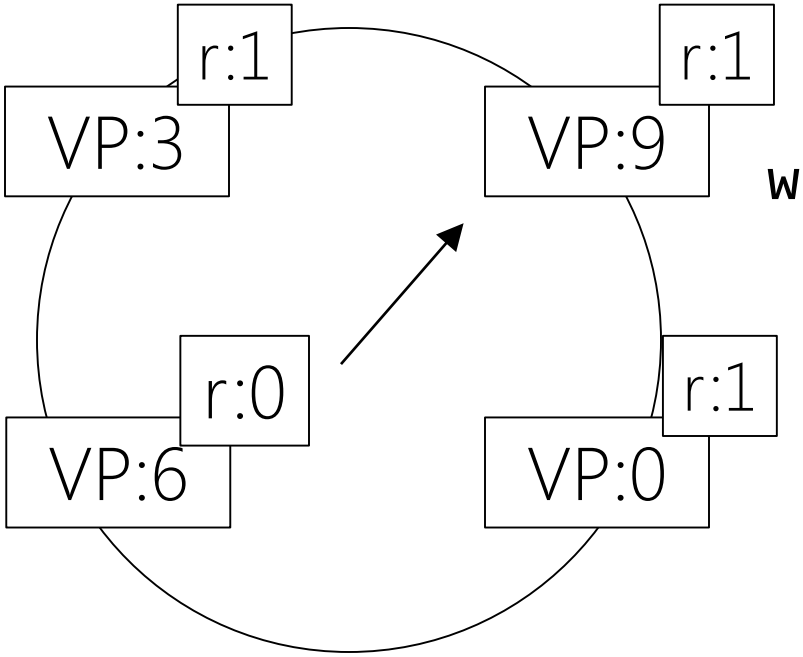
- The clock approach tries to approximate LRU
  - Assumes that:
    - The hardware will automatically set a "referenced" bit in the PTE when a load or store touches a page
    - Once set, the bit will only be cleared by software
  - x86 can provide such a bit, but not MIPS (why?)
- OS maintains a circular list of physical pages



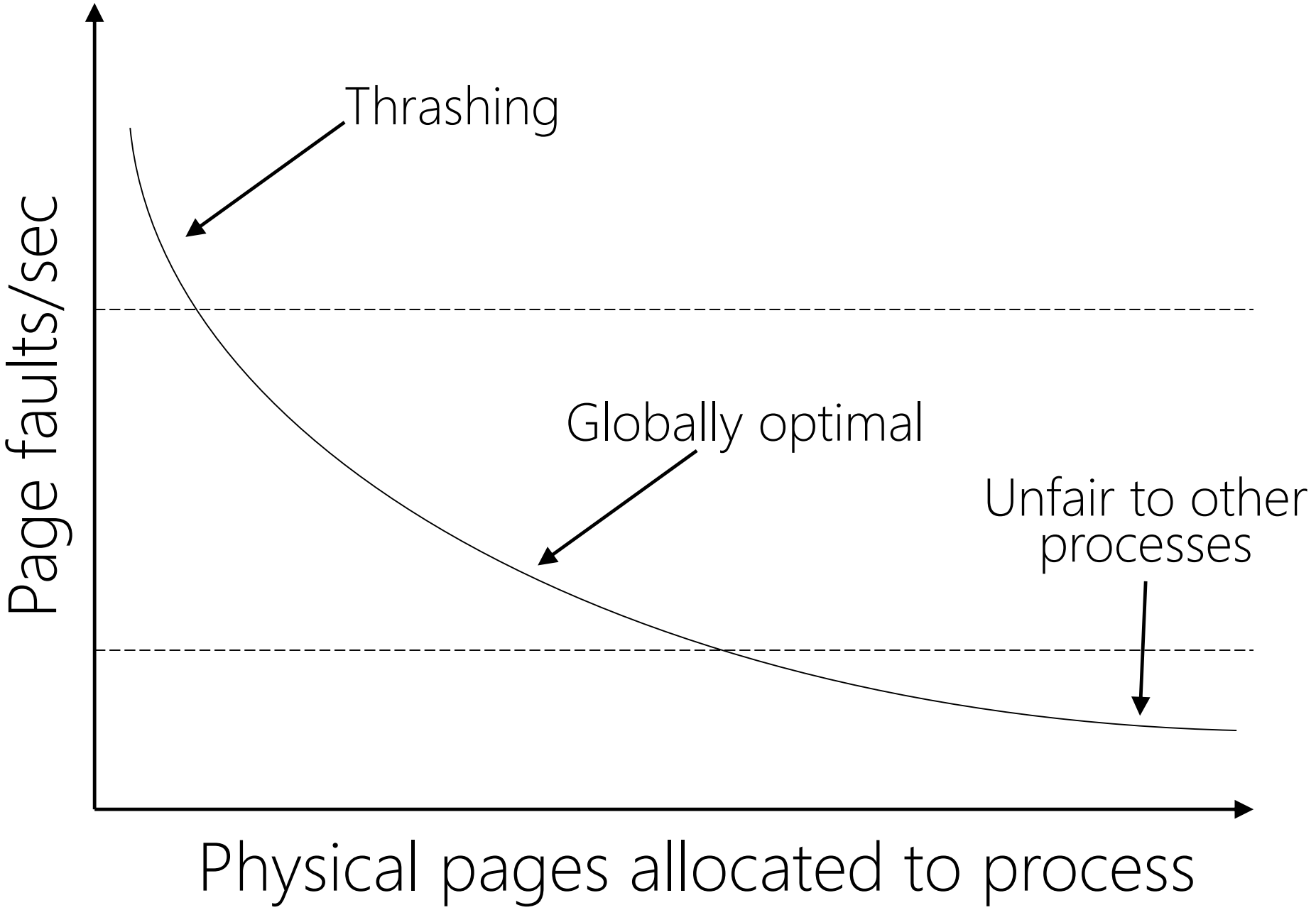

```
while True:
  if physPages[clock].r == 0:
    evict(physPages[clock])
    break
  physPages[clock].r = 0
  clock = (clock+1)%numPhysPages
```

Page Replacement: Clock

- If the clock hand is sweeping very slowly, there is plenty of physical RAM and few page faults (this is good)
- If the clock hand is sweeping quickly, there is too little physical RAM---page faults are frequent, and the machine is thrashing (i.e., spending a lot of time swapping instead of doing real work)




```
while True:
  if physPages[clock].r == 0:
    evict(physPages[clock])
    break
  physPages[clock].r = 0
  clock = (clock+1)%numPhysPages
```


Thrashing
Globally optimal
Unfair to other processes
Page faults/sec
Physical pages allocated to process