CSC369: Operating Systems

Fall 2014

Andrew Petersen

No Afternoon Office Hours

- No afternoon office hours today!

- I'll be in the office immediately after class
- I'm also available by appointment tomorrow

SOTI

- Thanks for dropping by the industry booth last week!
- I heard a story that two of you helped fix their wireless.
- The career centre wants you to know that while the company has high experience requirements for many positions, they will be looking for junior developers.

Paging

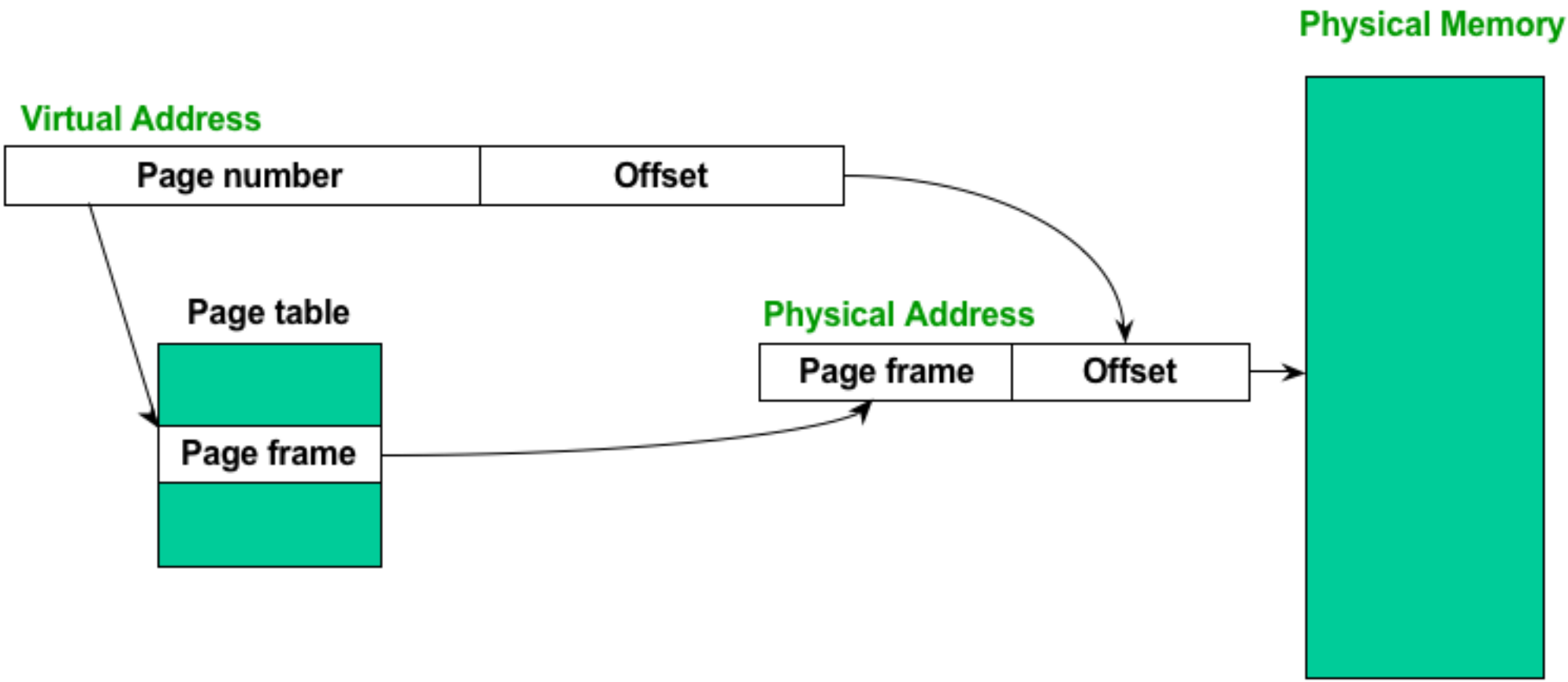
- Partition memory into equal, fixed-size chunks
  - These are called page frames or simply frames
- Divide processes' memory into chunks of the same size
  - These are called pages
- Any page can be assigned to any free page frame
  - External fragmentation is eliminated
  - Internal fragmentation is at most a part of a page
- Possible page frame sizes are restricted to powers of 2 to simplify translation

Support for Paging

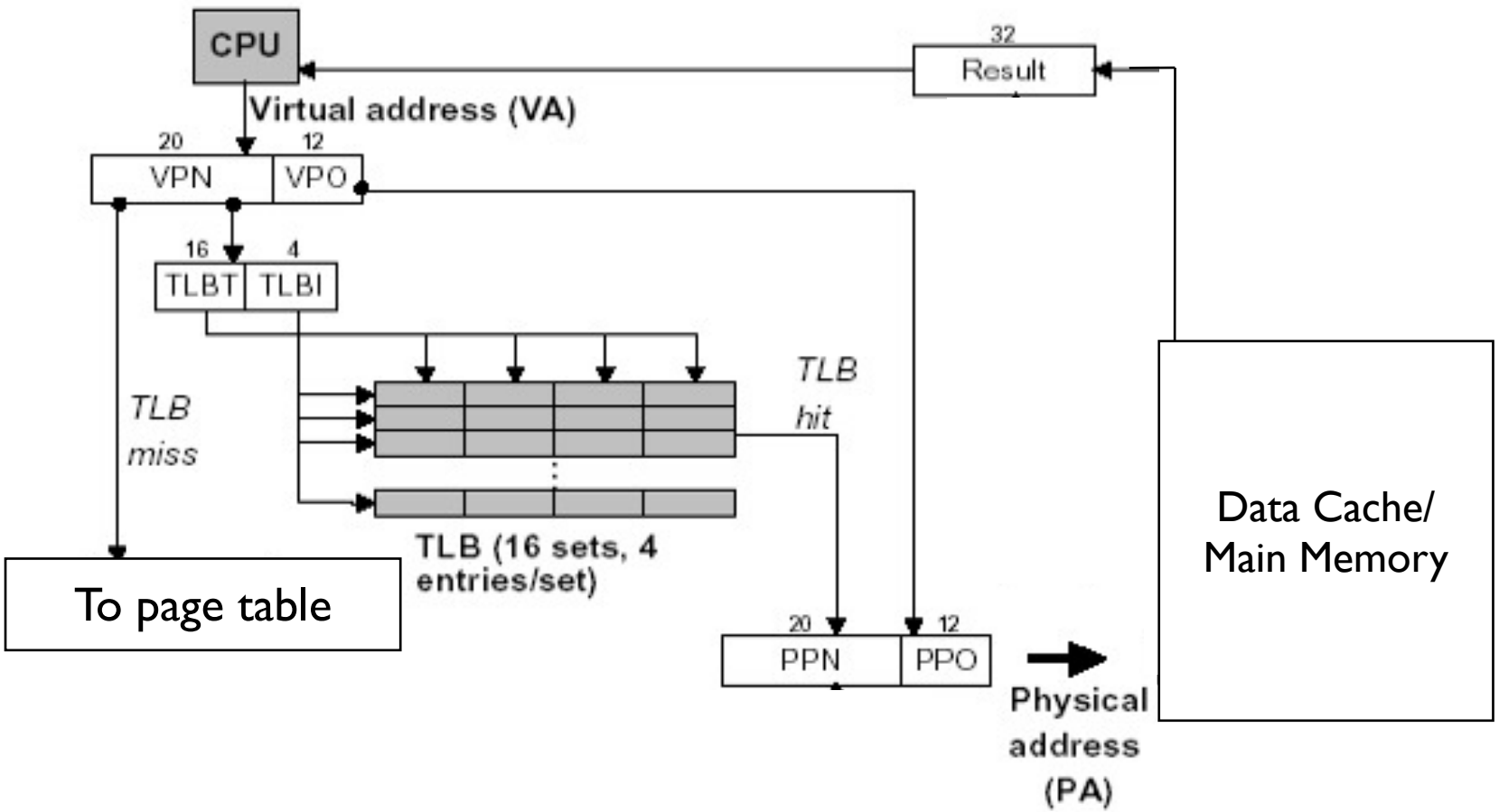
- Need more than base & limit register

- The OS maintains a page table for each process
  - Page table records which physical frame holds each page
  - virtual addresses are now page number + page offset
    - page number = vaddr / page_size
    - page offset = vaddr % page_size
  - On each memory reference, processor translates the page number to its frame number and adds the offset to generate a physical address

Paged Address Translation



Example: Pentium



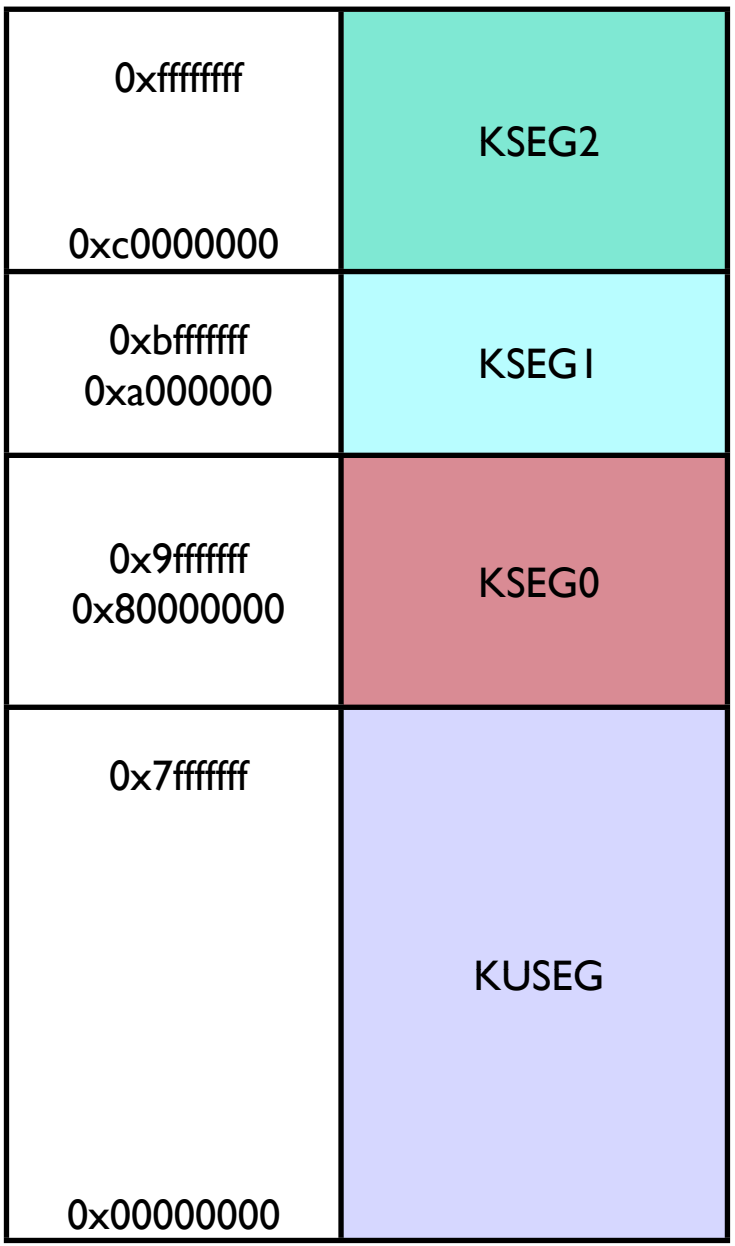
Where are Page Tables Stored?

- Simplest version
  - a linear array of entries, 1 entry per page
  - Stored in memory, attached to process
  - Virtual page number (VPN) is array index

```
struct addrspace {
    paddr_t pgtbl;
    …
}
```

```
struct addrspace *
as_create(void) {
    struct addrspace *as =
     kmalloc(sizeof(struct addrspace));
    int nentries = (unsigned)(-1) >> 13;
    int npages = DIVROUNDUP(nentries*
                 sizeof(pte_t), PAGE_SIZE);
    as->pgtbl = getppages(npages);
    ...
}
```

MIPS R2000 Virtual Memory Space



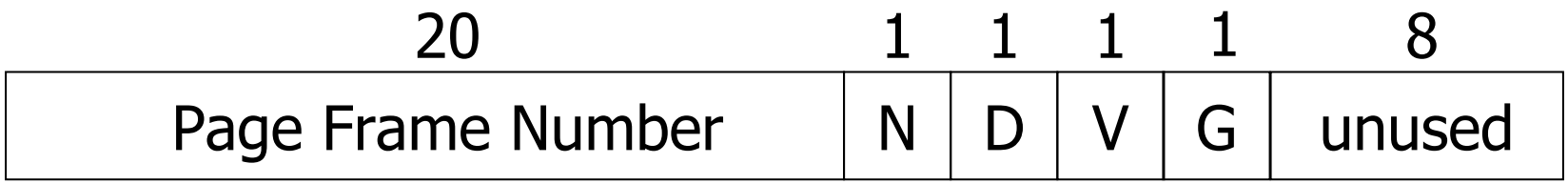
- MMU defines 4 distinct regions with different properties:
  - KUSEG for user-addresses. Translated using paging, cacheable
  - KSEG0 for direct-mapped, cacheable kernel addresses
    - Translation to/from physical simply subtracts/adds 0x80000000 to V.A.
  - KSEG1 like KSEG0 but no caching
  - KSEG2 for kernel addresses that are translated using paging

Addressing Page Tables

Where do we store page tables (which address space)?

- Physical memory
  - Easy to address, no translation required (or very simple translation, like KSEG0 in MIPS R2000)
  - allocated page tables consume memory for lifetime of VAS
- Virtual memory (OS virtual address space, KSEG2)
  - Cold (unused) page table pages can be paged out to disk
  - But, addressing page tables requires translation
  - Do not page the outer page table (called wiring)
- If we're going to page the page tables, might as well page the entire OS address space, too
  - Need to wire special code and data (fault, interrupt handlers)

Example: MIPS Page Table Entries



- N == not cached
- D == dirty (meaning "writable", not set by hardware)
- V == valid
- G == global (can be used by all processes)

- Maximum 2^{20} physical pages, each 4 kB
- Maximum 4GB of physical RAM

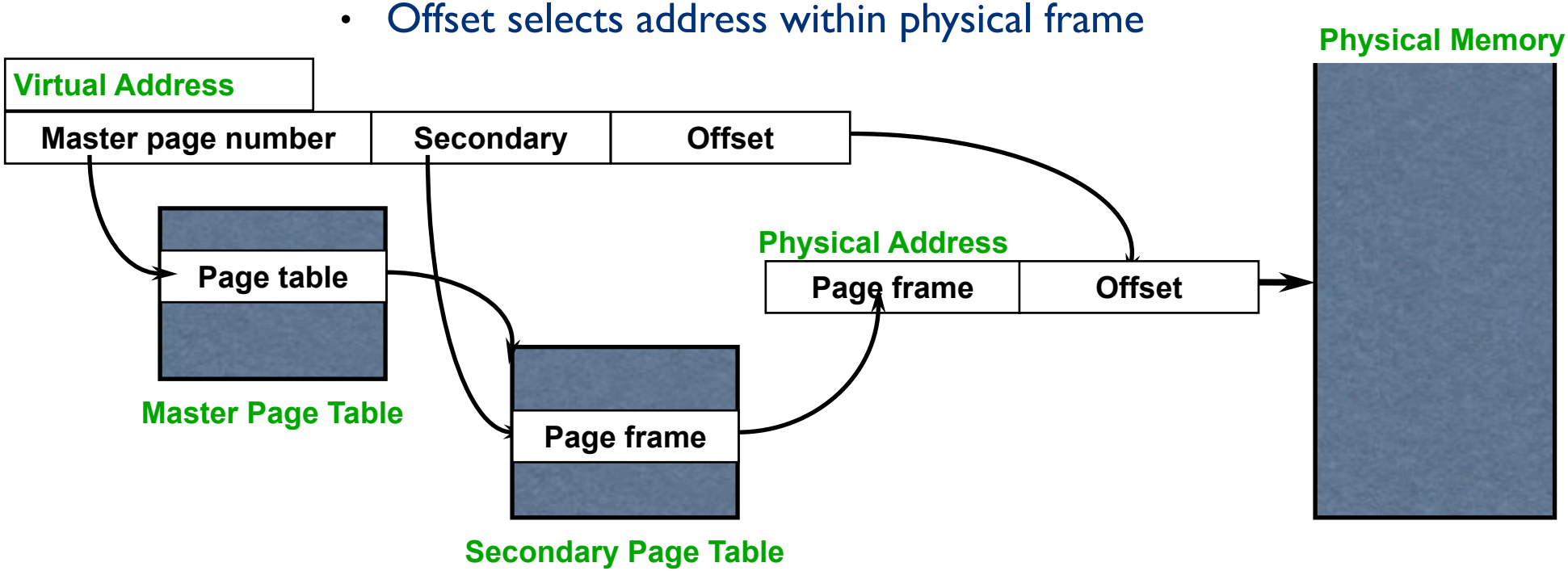
Paging Limitations - Space

- The memory required for a page table can be very large
- Need one PTE per page
  - 32 bit virtual address space w/ 4K pages = 2^{20} PTEs
  - 4 bytes/PTE = 4MB/page table
  - 25 processes = 100MB just for page tables!
    - And modern processors have 64-bit address spaces -> 16 petabytes for the page table!
  - Observation: We only need to map the portion of the address space actually being used (a tiny fraction of entire addr space)

Two-Level Page Tables

Virtual addresses (VAs) have three parts:

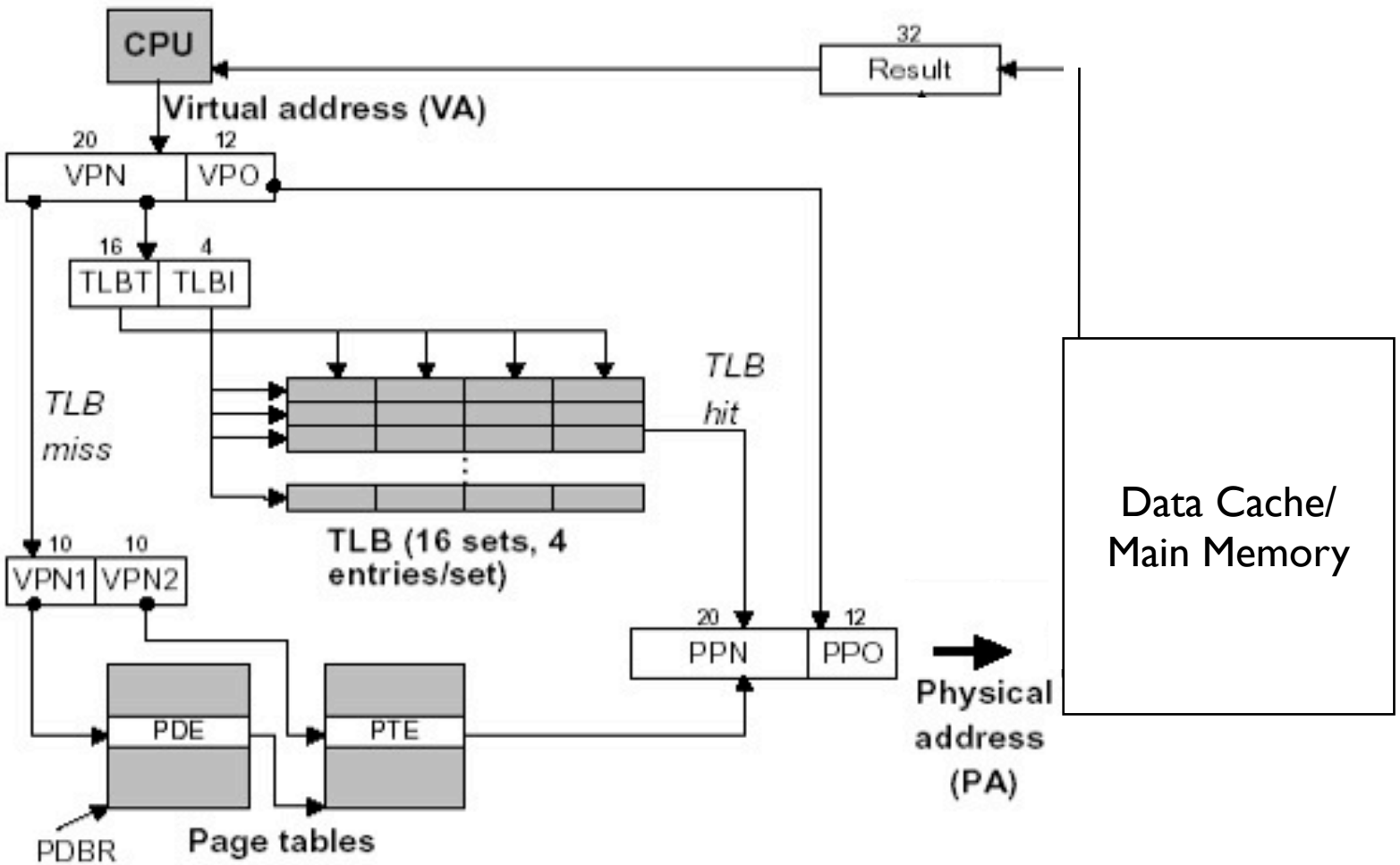
- Master page number, secondary page number, and offset
- Master page table maps VAs to secondary page table
- Secondary page table maps page number to physical frame
- Offset selects address within physical frame



2-Level Paging Example

- 32-bit virtual address space
  - 4K pages, 4 bytes/PTE
  - How many bits in offset?
    - 4K = 12 bits, leaves 20 bits
  - Want master/secondary page tables in 1 page frame each:
    - 4K/4 bytes = 1K entries. How many bits?
    - Master (1K) = 10, offset = 12, inner = 32 – 10 – 12 = 10 bits

Pentium Address Translation



64-bit Address Spaces

- Suppose we just extended the hierarchical page tables with more levels
  - 4K pages → 52 bits for page numbers
  - Maximum 1024 entries per level → 6 levels
    - Too much overhead
  - 16K pages → 48 bits for page numbers
  - Maximum 4096 entries per level -> 4 levels
    - Better, but still a lot

Efficient Translations

- Our original page table scheme already doubled the cost of doing memory lookups
  - One lookup into the page table, another to fetch the data
- Two-level page tables triple the cost!
  - Two lookups into the page tables, a third to fetch the data
  - And this assumes the page table is in memory
- TLB's hide the cost for frequently-used pages

Inverted Page Tables

- Keep one table with an entry for each physical page frame
- Entries record which virtual page # is stored in that frame
  - Need to record process id as well
- Less space, but lookups are slower
  - References use virtual addresses, table is indexed by physical addresses
  - Use hashing to reduce the search time

Advanced Material

Sharing

- Private virtual address spaces protect applications from each other
  - Usually exactly what we want
- But this makes it difficult to share data (have to copy)
  - Parents and children in a forking Web server or proxy will want to share an in-memory cache without copying
- We can use shared memory to allow processes to share data using direct memory references
  - Both processes see updates to the shared memory segment
    - Process B can immediately read an update by process A
  - How are we going to coordinate access to shared data?

Sharing

- How can we implement sharing using page tables?
  - Have PTEs in both tables map to the same physical frame
  - Each PTE can have different protection values
  - Must update both PTEs when page becomes invalid
- Can map shared memory at same or different virtual addresses in each process' address space
  - Different: Flexible (no address space conflicts), but pointers inside the shared memory segment are invalid (Why?)
  - Same: Less flexible, but shared pointers are valid (Why?)
- What happens if a pointer inside the shared segment references an address outside the segment?

Copy on Write

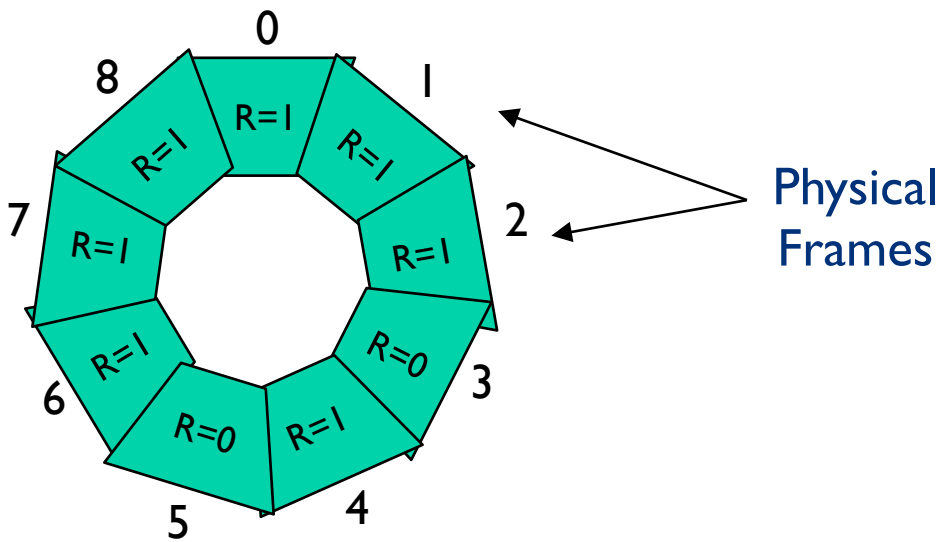
- OSes spend a lot of time copying data
  - System call arguments between user/kernel space
  - Entire address spaces to implement fork()
- Use Copy on Write (CoW) to defer large copies as long as possible, hoping to avoid them altogether
  - Instead of copying pages, create shared mappings of parent pages in child virtual address space
  - Shared pages are protected as read-only in child
    - Reads happen as usual
    - Writes generate a protection fault, trap to OS, copy page, change page mapping in client page table, restart write instruction
  - How does this help fork()?

Mapped Files

- Mapped files enable processes to do file I/O using loads and stores
  - Instead of "open, read into buffer, operate on buffer, …"
- Bind a file to a virtual memory region (mmap() in Unix)
  - PTEs map virtual addresses to physical frames holding file data
  - Virtual address base + N refers to offset N in file
- Initially, all pages mapped to file are invalid
  - OS reads a page from file when invalid page is accessed
  - OS writes a page to file when evicted, or region unmapped
  - If page is not dirty (has not been written to), no write needed
    - Another use of the dirty bit in PTE

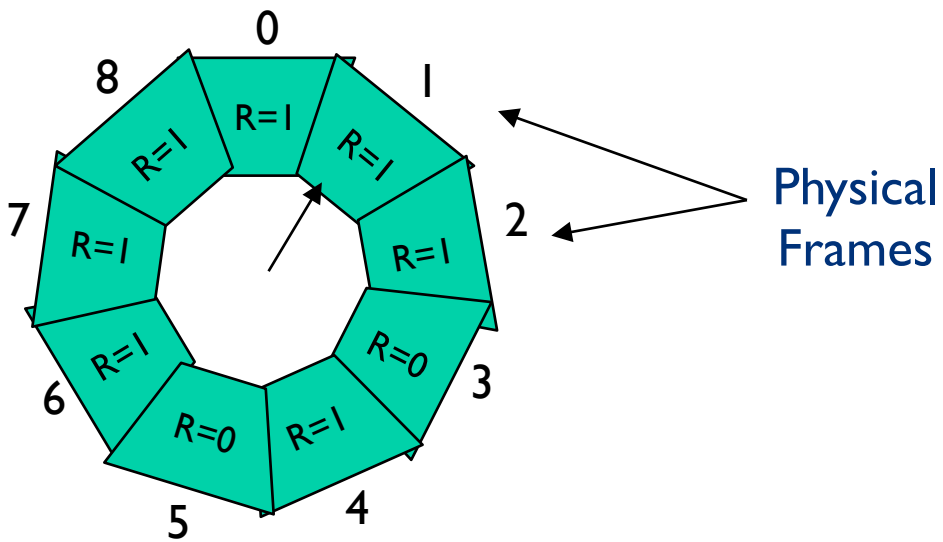
Clock Slides (for A2)

An Approximation: The Clock Algorithm



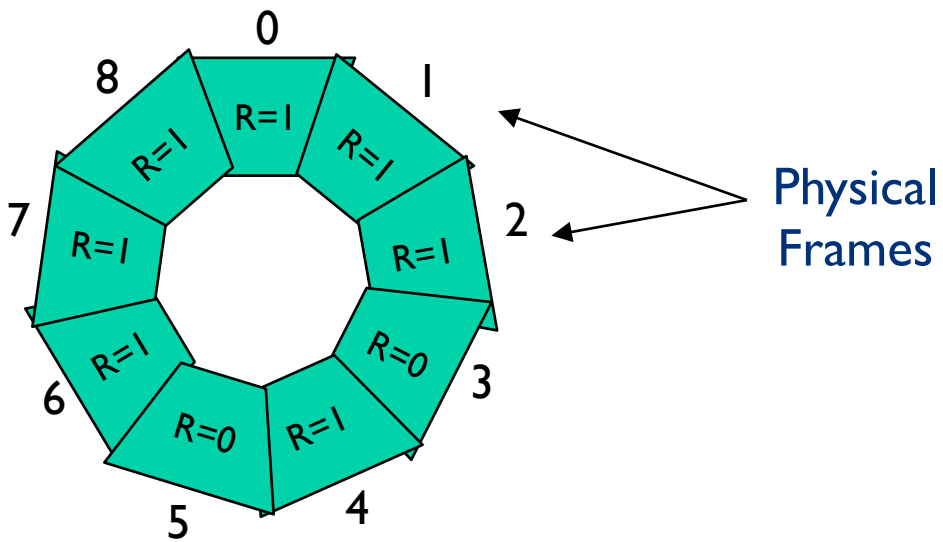
- 1st page fault:
  - Advance hand to frame 4, use frame 3
- 2nd page fault (assume none of these pages are referenced)
  - Advance hand to frame 6, use frame 5

An Approximation: The Clock Algorithm



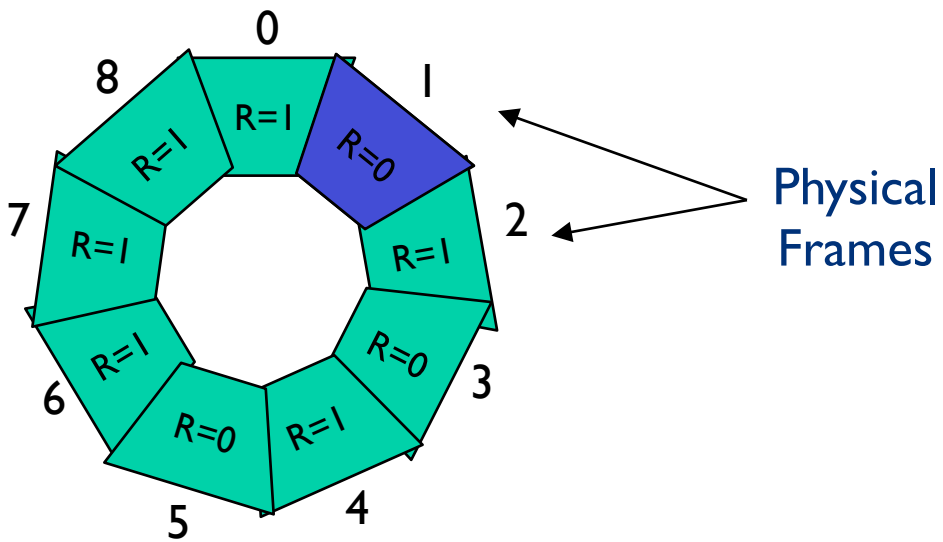
- 1st page fault:
  - Advance hand to frame 4, use frame 3
- 2nd page fault (assume none of these pages are referenced)
  - Advance hand to frame 6, use frame 5

An Approximation: The Clock Algorithm



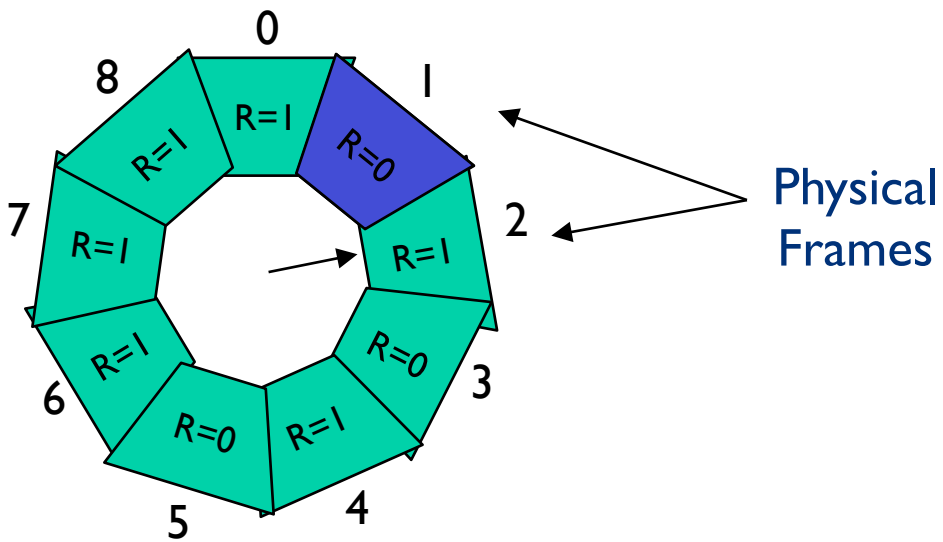
- 1st page fault:
  - Advance hand to frame 4, use frame 3
- 2nd page fault (assume none of these pages are referenced)
  - Advance hand to frame 6, use frame 5

An Approximation: The Clock Algorithm



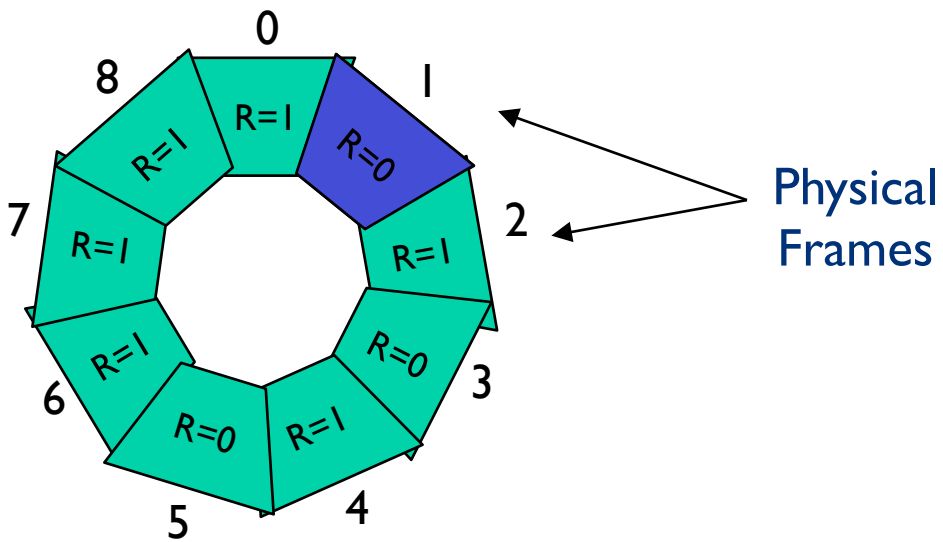
- 1st page fault:
  - Advance hand to frame 4, use frame 3
- 2nd page fault (assume none of these pages are referenced)
  - Advance hand to frame 6, use frame 5

An Approximation: The Clock Algorithm



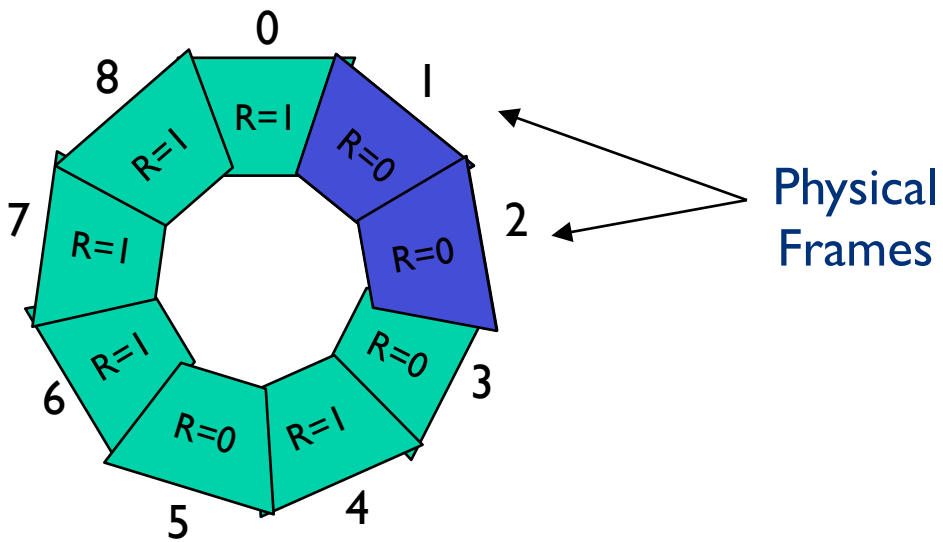
- 1st page fault:
  - Advance hand to frame 4, use frame 3
- 2nd page fault (assume none of these pages are referenced)
  - Advance hand to frame 6, use frame 5

An Approximation: The Clock Algorithm



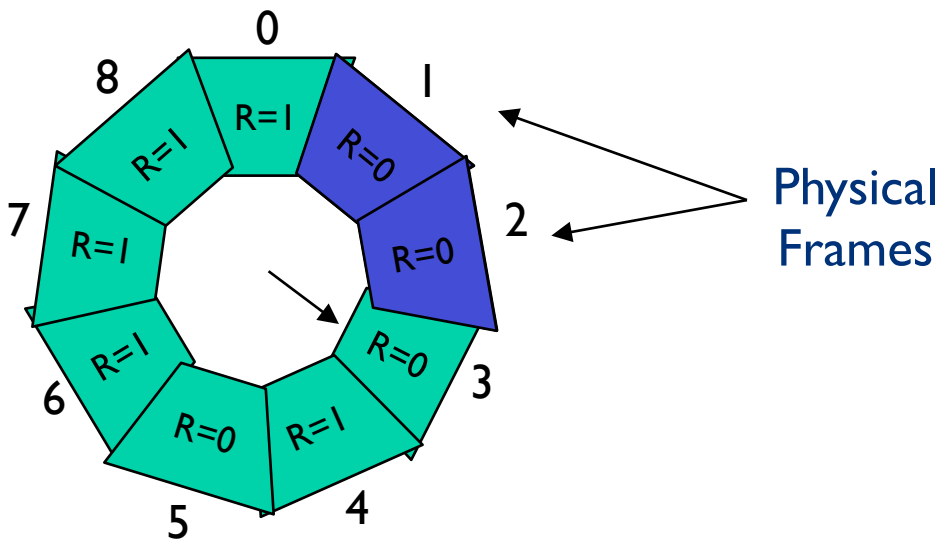
- 1st page fault:
  - Advance hand to frame 4, use frame 3
- 2nd page fault (assume none of these pages are referenced)
  - Advance hand to frame 6, use frame 5

An Approximation: The Clock Algorithm



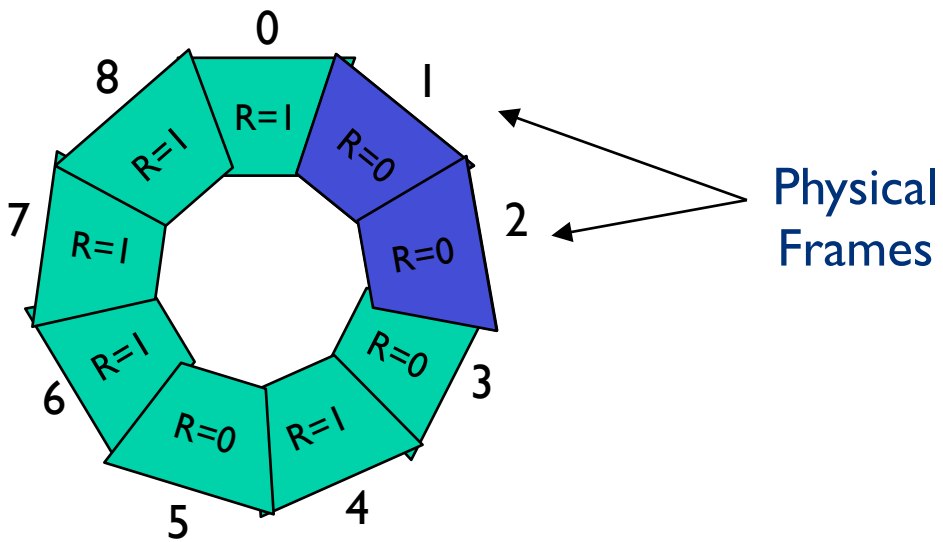
- 1st page fault:
  - Advance hand to frame 4, use frame 3
- 2nd page fault (assume none of these pages are referenced)
  - Advance hand to frame 6, use frame 5

An Approximation: The Clock Algorithm



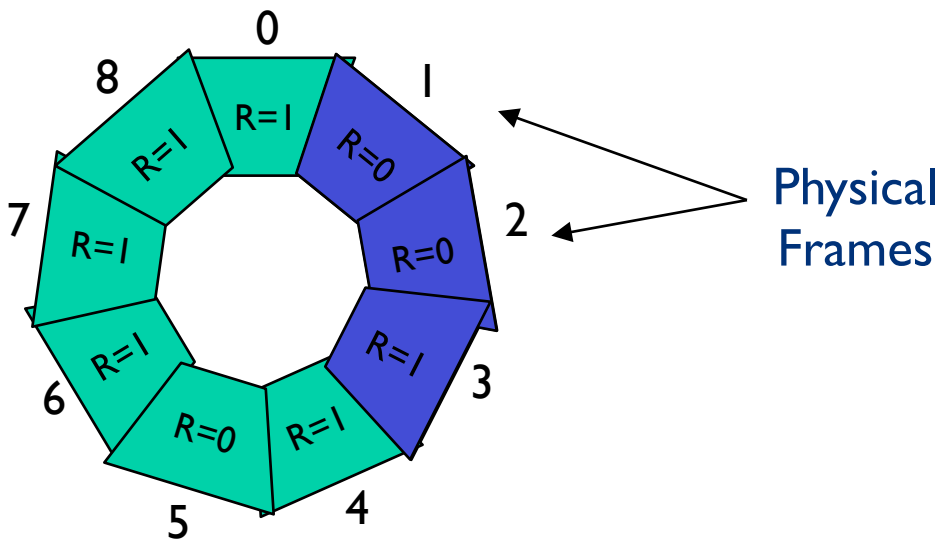
- 1st page fault:
  - Advance hand to frame 4, use frame 3
- 2nd page fault (assume none of these pages are referenced)
  - Advance hand to frame 6, use frame 5

An Approximation: The Clock Algorithm



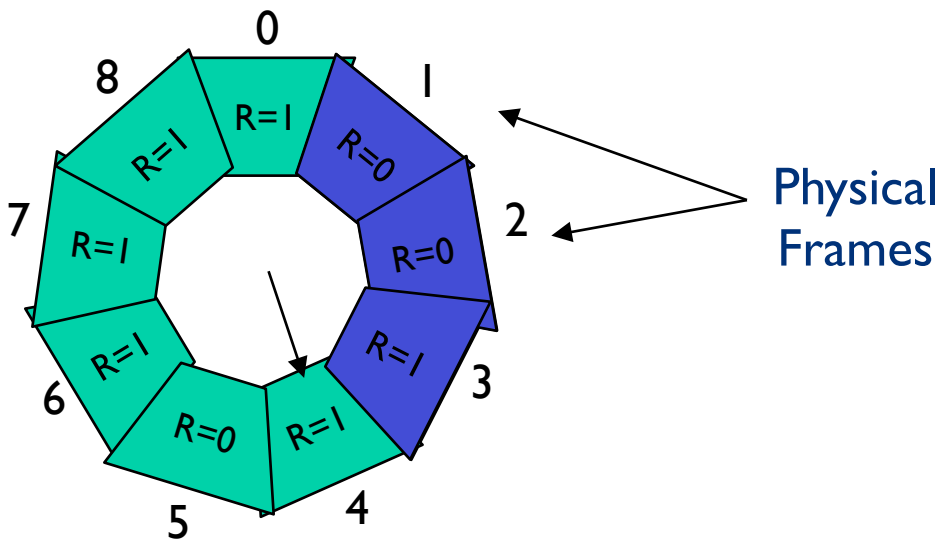
- 1st page fault:
  - Advance hand to frame 4, use frame 3
- 2nd page fault (assume none of these pages are referenced)
  - Advance hand to frame 6, use frame 5

An Approximation: The Clock Algorithm



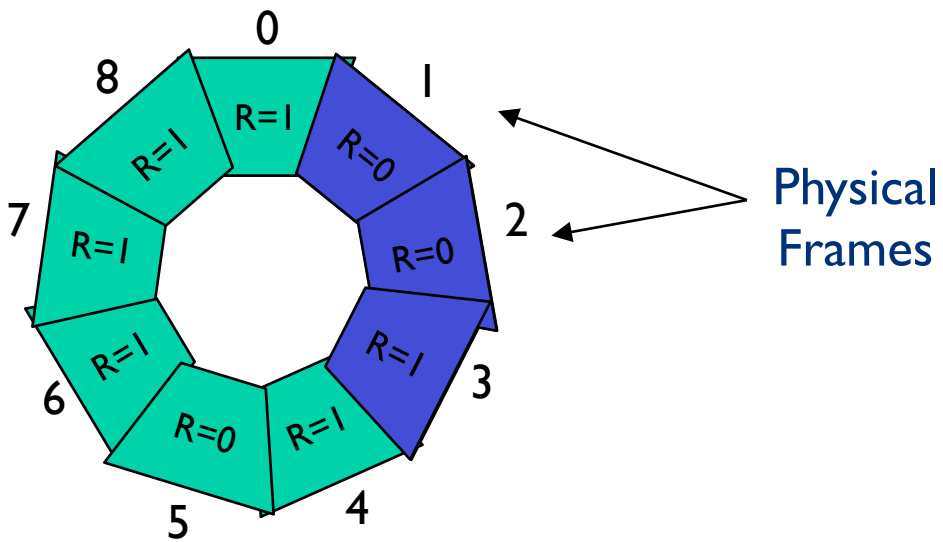
- 1st page fault:
  - Advance hand to frame 4, use frame 3
- 2nd page fault (assume none of these pages are referenced)
  - Advance hand to frame 6, use frame 5

An Approximation: The Clock Algorithm



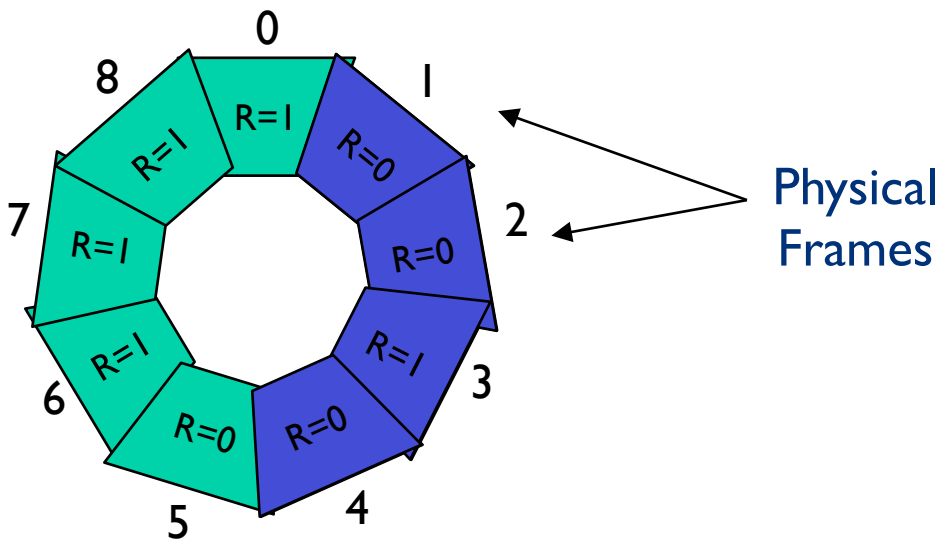
- 1st page fault:
  - Advance hand to frame 4, use frame 3
- 2nd page fault (assume none of these pages are referenced)
  - Advance hand to frame 6, use frame 5

An Approximation: The Clock Algorithm



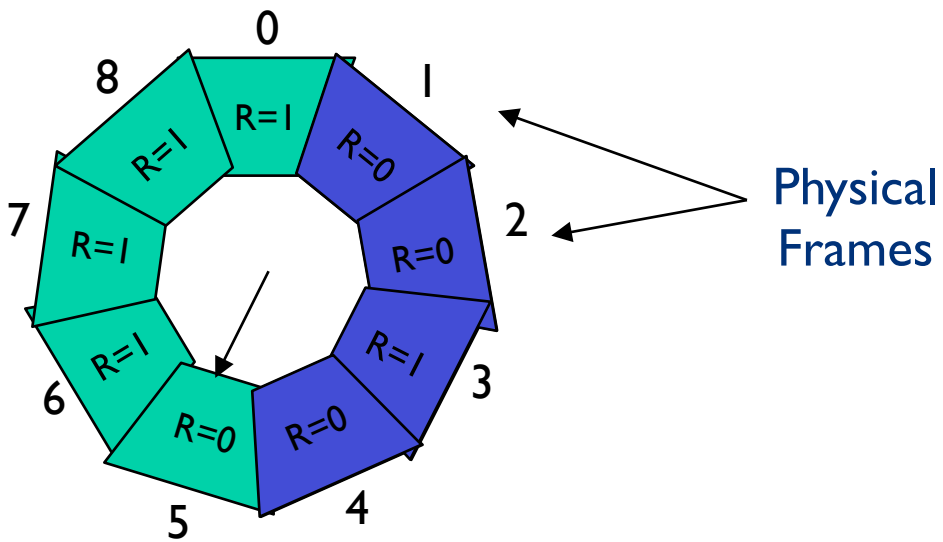
- 1st page fault:
  - Advance hand to frame 4, use frame 3
- 2nd page fault (assume none of these pages are referenced)
  - Advance hand to frame 6, use frame 5

An Approximation: The Clock Algorithm



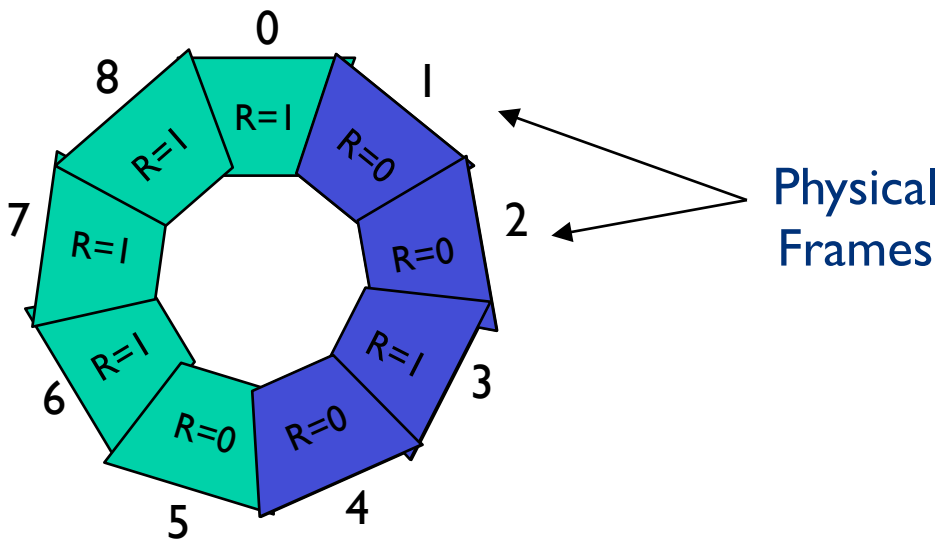
- 1st page fault:
  - Advance hand to frame 4, use frame 3
- 2nd page fault (assume none of these pages are referenced)
  - Advance hand to frame 6, use frame 5

An Approximation: The Clock Algorithm



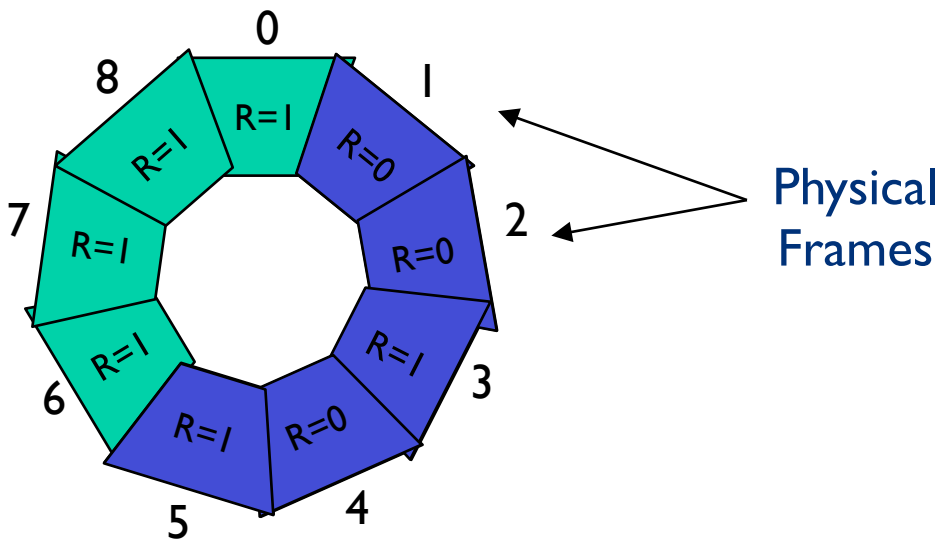
- 1st page fault:
  - Advance hand to frame 4, use frame 3
- 2nd page fault (assume none of these pages are referenced)
  - Advance hand to frame 6, use frame 5

An Approximation: The Clock Algorithm



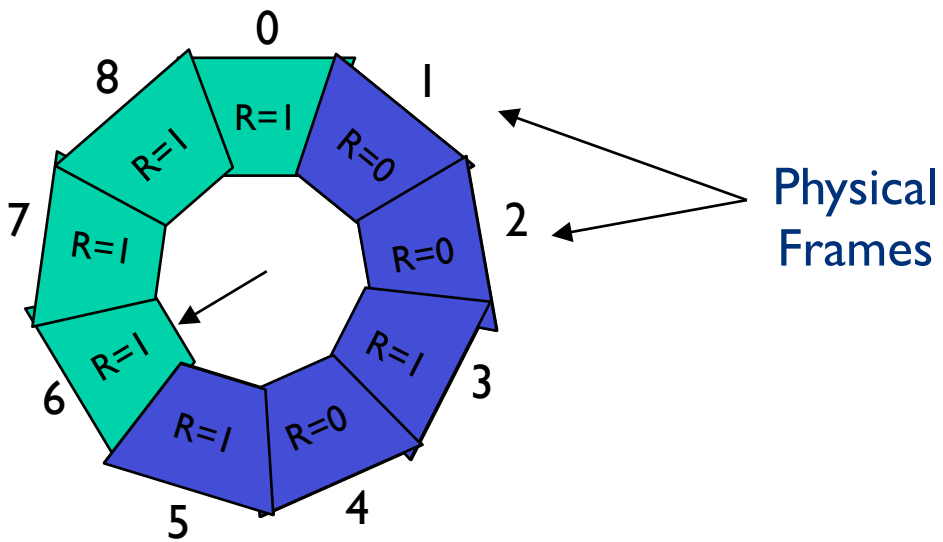
- 1st page fault:
  - Advance hand to frame 4, use frame 3
- 2nd page fault (assume none of these pages are referenced)
  - Advance hand to frame 6, use frame 5

An Approximation: The Clock Algorithm



- 1st page fault:
  - Advance hand to frame 4, use frame 3
- 2nd page fault (assume none of these pages are referenced)
  - Advance hand to frame 6, use frame 5

An Approximation: The Clock Algorithm



- 1st page fault:
  - Advance hand to frame 4, use frame 3
- 2nd page fault (assume none of these pages are referenced)
  - Advance hand to frame 6, use frame 5

Clock Details

- A clock hand sweeps through the entries in order
- When an eviction is required, inspect the reference bit of the current page
  - If ref bit is 0, replace the page
  - If ref bit is 1, clear ref bit and move to next page
  - Pages that are used often enough to keep reference bits set will not be replaced
- This scheme is sometimes called "Not Recently Used"