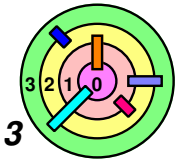
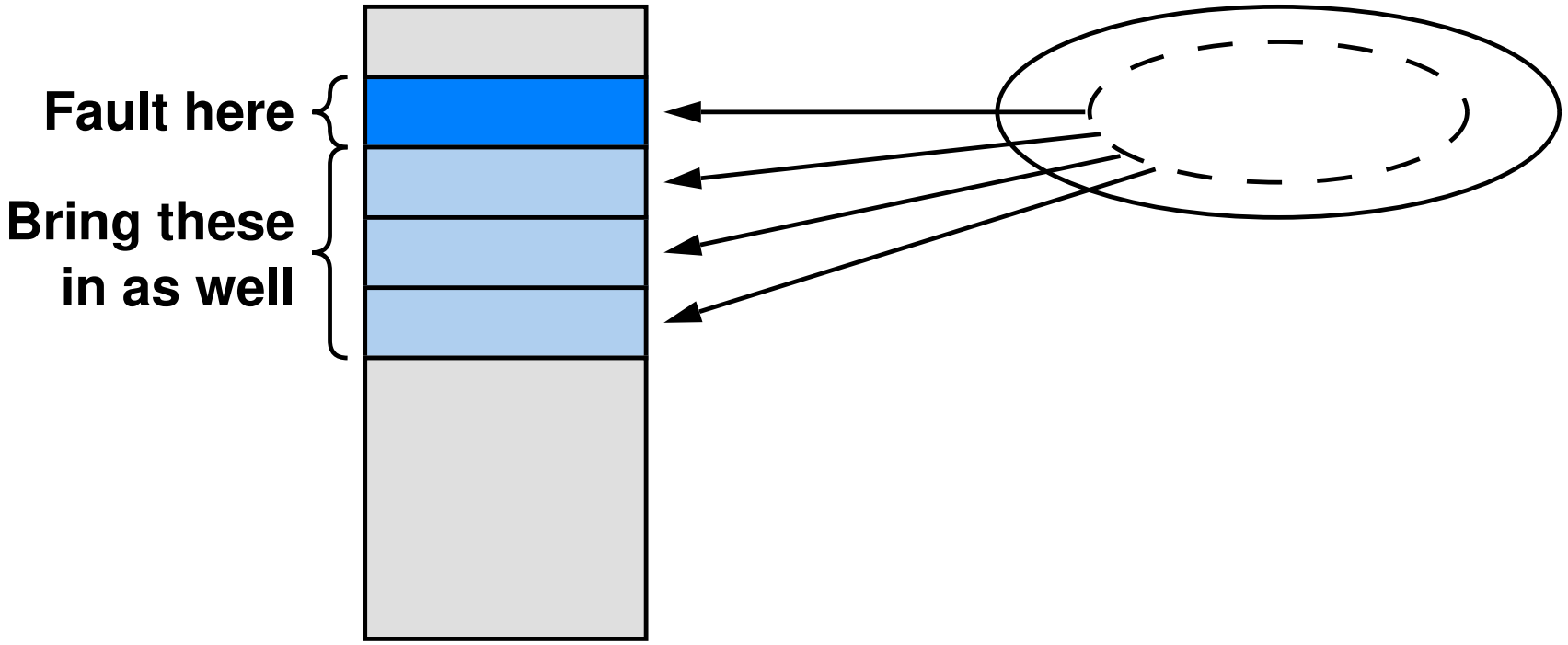
Performance

- *Page Fault* (accessing a page with *V=0*)
  1) Trap occurs (due to a page fault)
  2) Find free physical page
  3) Write page out if no free physical page
  4) Fetch page
  5) Return from trap

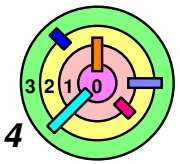
- A page fault can result in disk operations and slow down the application
  - do not want to wait for the disk!
  - need to reduce this latency
    - *prefetching*
    - *pageout daemon*



Improving the Fetch Policy

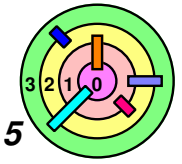


- This is *prefetching*
  - accesses to pages is often sequential
  - gamble that this is worthwhile (since it takes up more memory)
- This improves step (4) on previous page
  - but it uses up physical memory faster and makes (3) more likely to occur

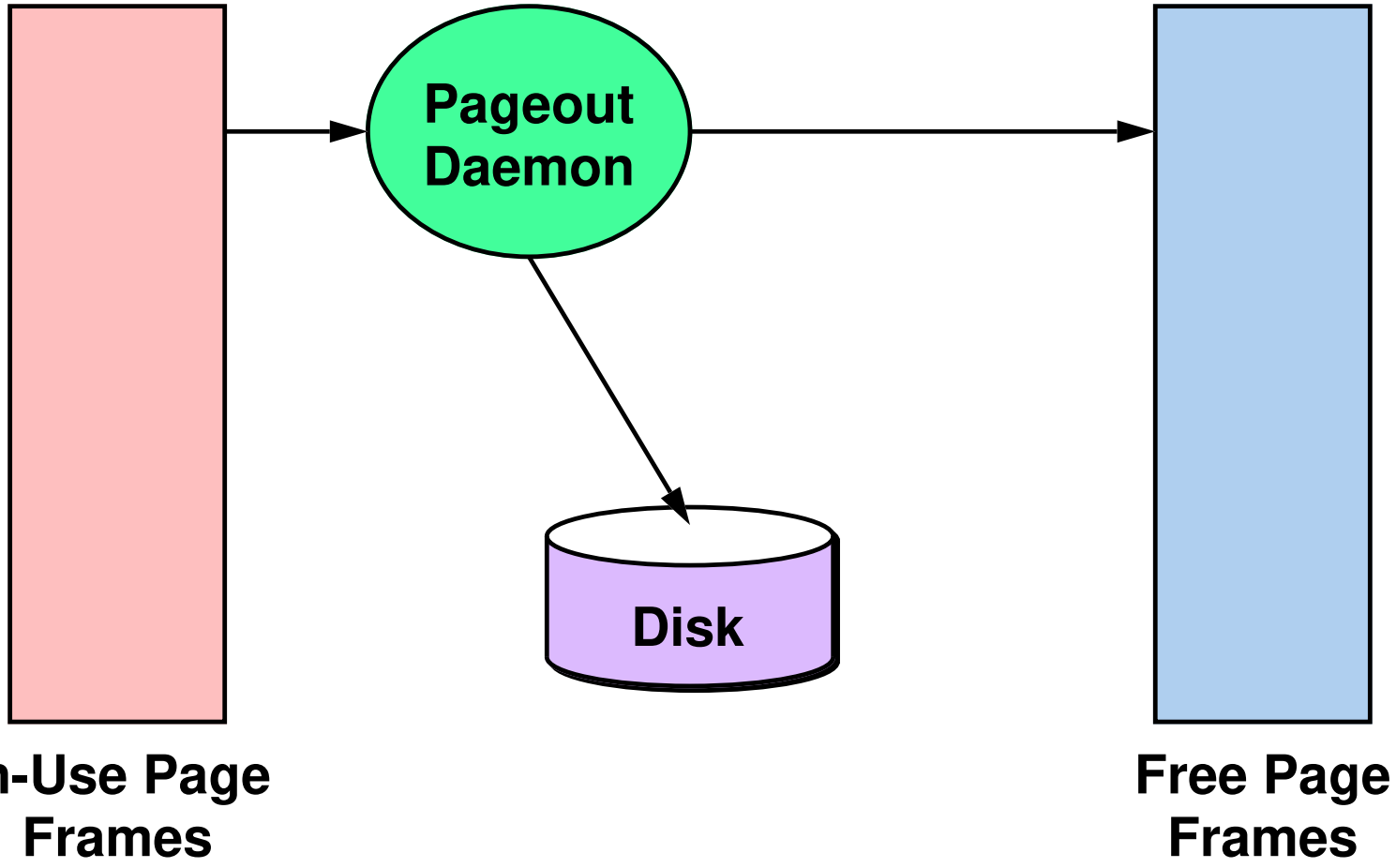


Improving the Replacement Policy

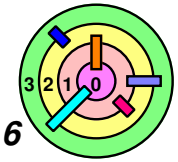
- ***When*** **is replacement done?**
  - **doing it "on demand" causes excessive delays**
    - **so, "on-demand" (or Lazy Evaluation) is *not always* a good policy**
  - **should be performed as a separate, *concurrent* activity**
    - **use a thread *(i.e., a pageout deamon)* to *continuously* and *aggressively* look for free pages**



The "Pageout Daemon"

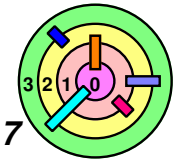


- ***Page frames*** **are used to keep track of physical pages**
- **Can use *multiple* pageout daemons**

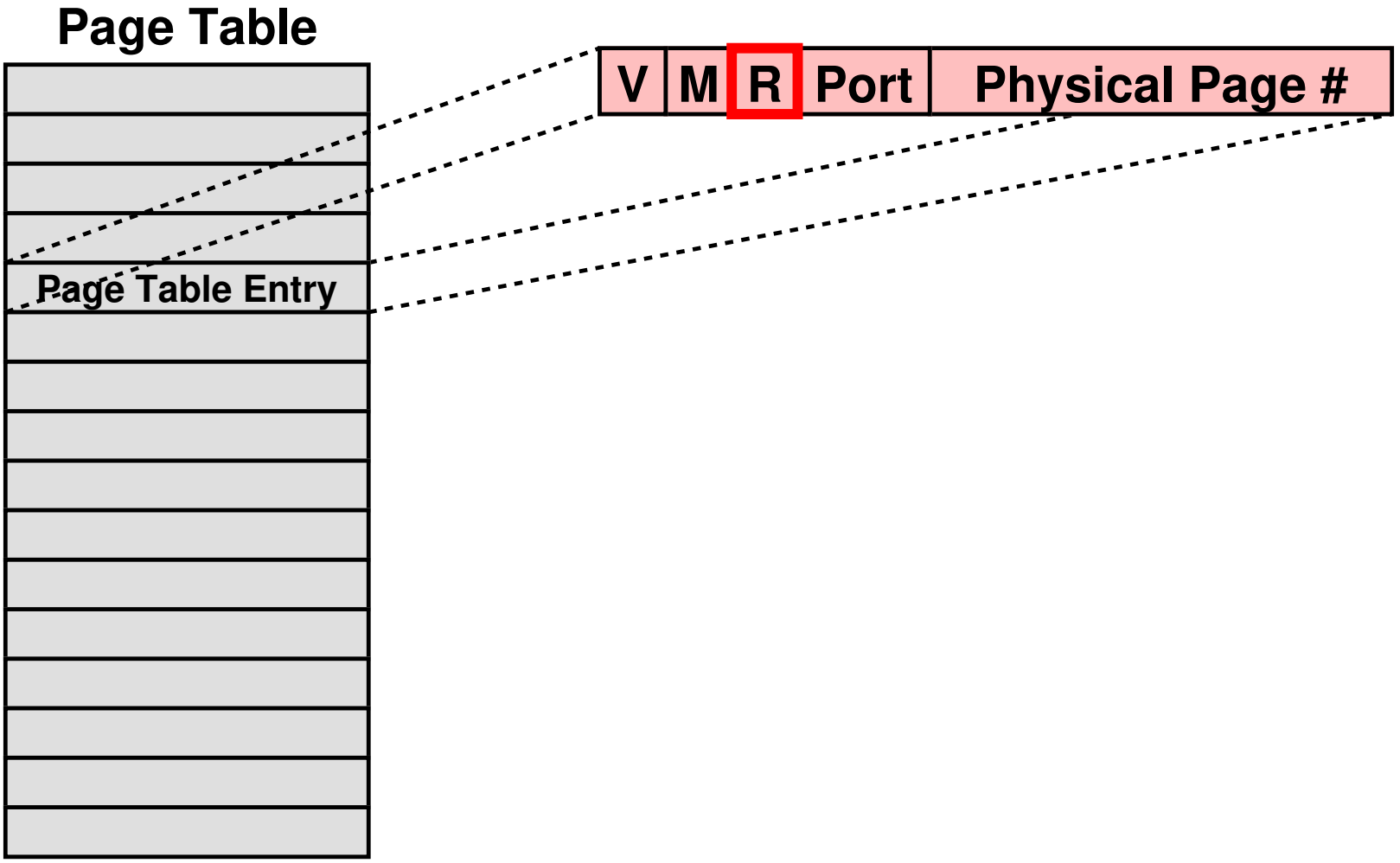


Choosing the Page to Remove - Replacement Policy

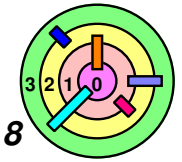
- *Which* pages are replaced?
  - FIFO policy is not good
  - want to replace those pages least likely to be referenced soon
- If your DVD rack is full and you just bought a new DVD
  - which DVD would you remove from the rack to make room for the new DVD?
- Idealized policies:
  - FIFO (First-In-First-Out)
  - LFU (Least-Frequently-Used)
  - LRU (Least-Recently-Used)



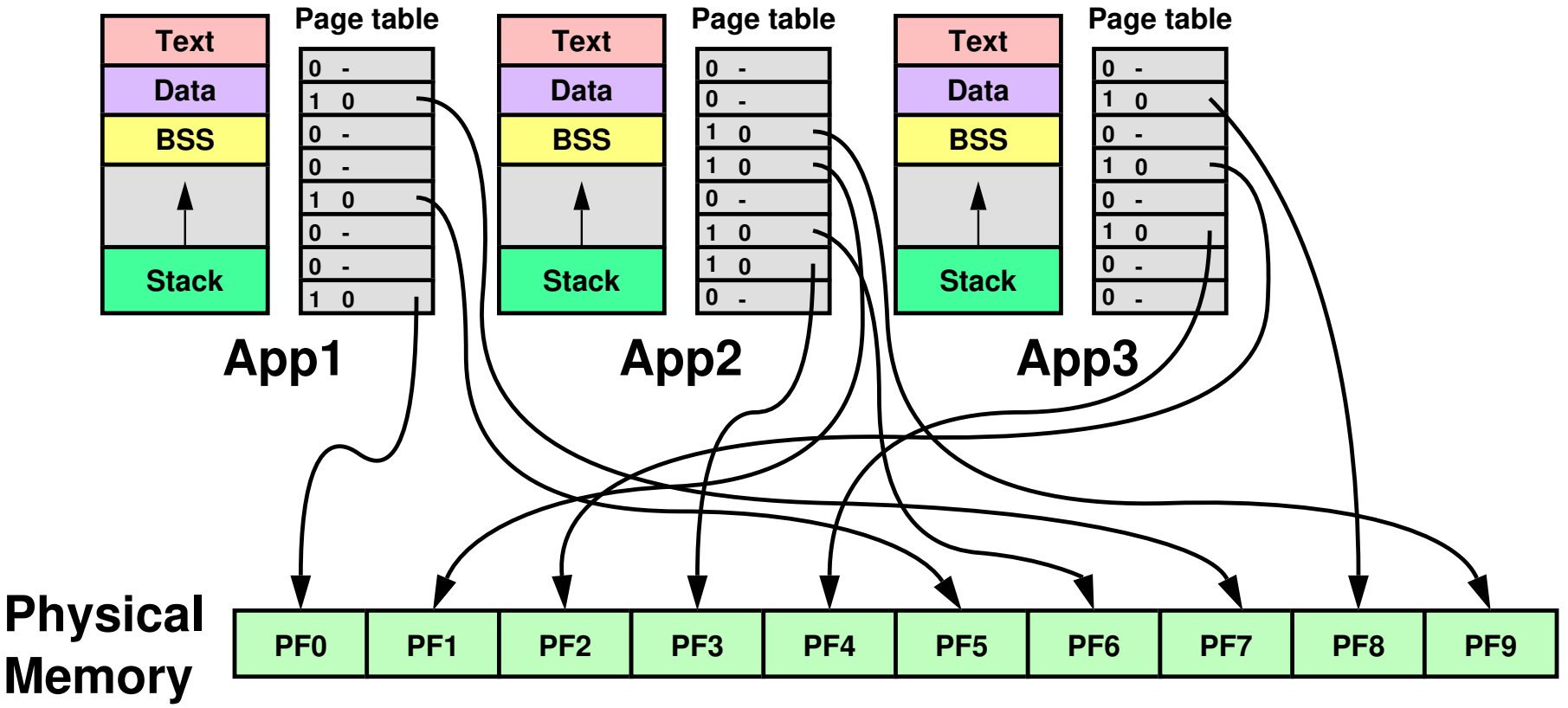
Implementing LRU



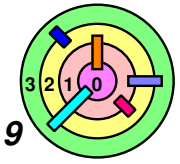
To approximate LRU (a very coarse approximation), the *reference* bit in the page table entry is used



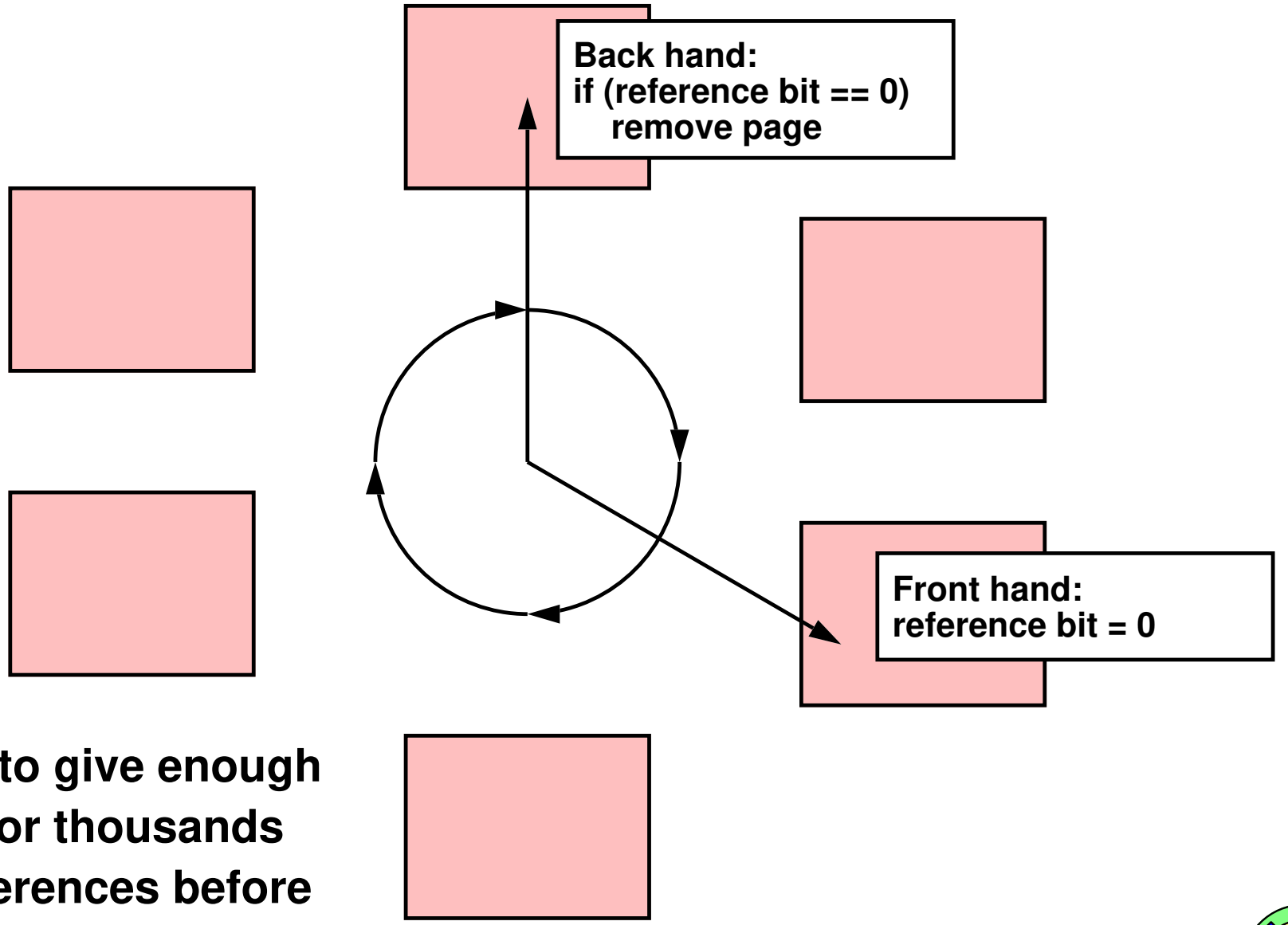
Using The Reference Bits



- Why would some pages referenced more often than other?
  - code?
  - stack?
  - depends on the application

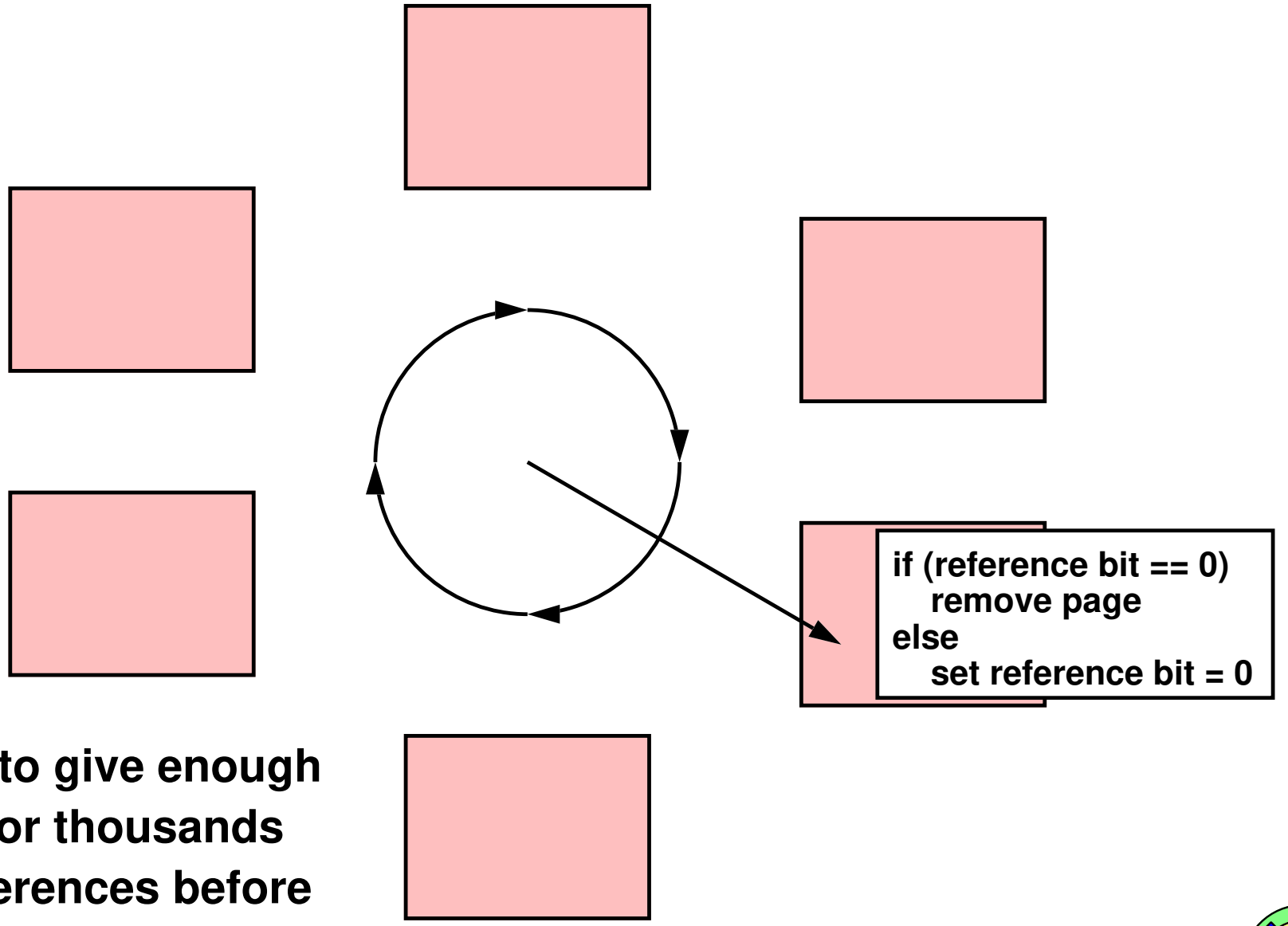


Clock Algorithm - Two-handed



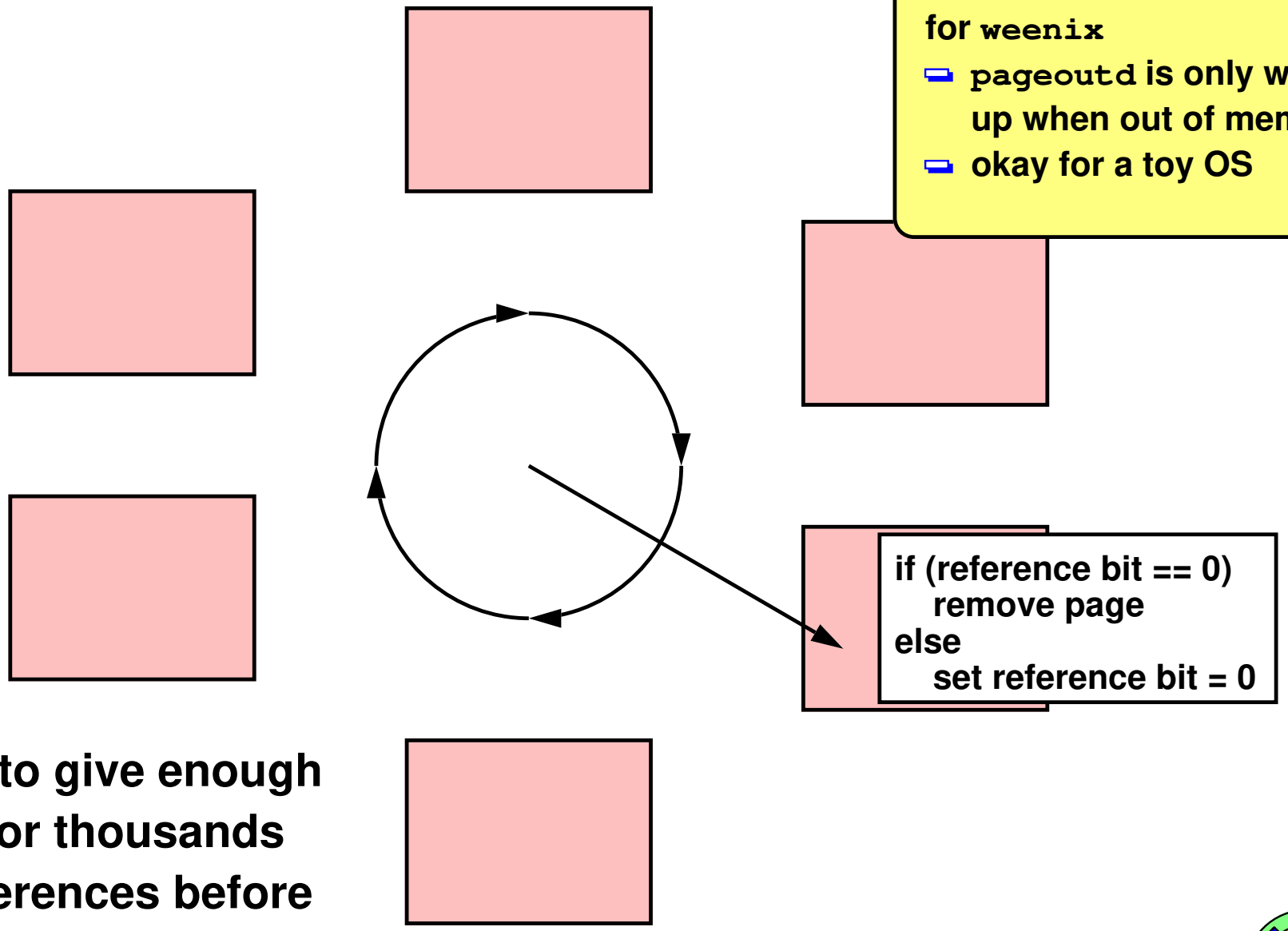
- Need to give enough time for thousands of references before checking

Clock Algorithm - One-handed



- Need to give enough time for thousands of references before checking

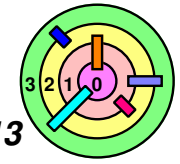
Clock Algorithm - One-handed



- Need to give enough time for thousands of references before checking

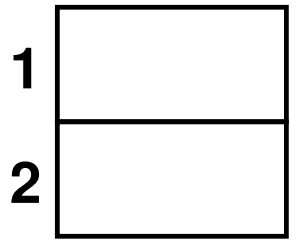
Global vs. Local Allocation

- What if a process uses up all the page frames?
- Global allocation
  - all processes *compete* for page frames from a single pool
  - problem:
    - memory-hungry processes will get all the memory
    - possibility of *thrashing*
- Local allocation
  - each process has its own *private* pool of page frames
  - Windows does this
    - processes do not have to compete for the same pool of page frames
    - goal is to *minimize the possibility of thrashing*

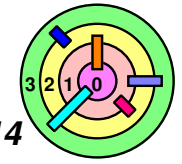


Thrashing

- Consider a system that has exactly two page frames:
  - process A has a page in frame 1
  - process B has a page in frame 2

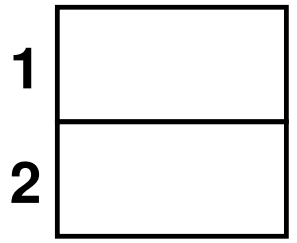


- Process A references another page, causing a page fault
  - the page in frame 2 is removed from B and given to A
- Process B faults immediately; the page in frame 1 is given to B
- Process A resumes execution and faults again; the page in frame 1 is given back to A
- ...
  - neither processes makes progress

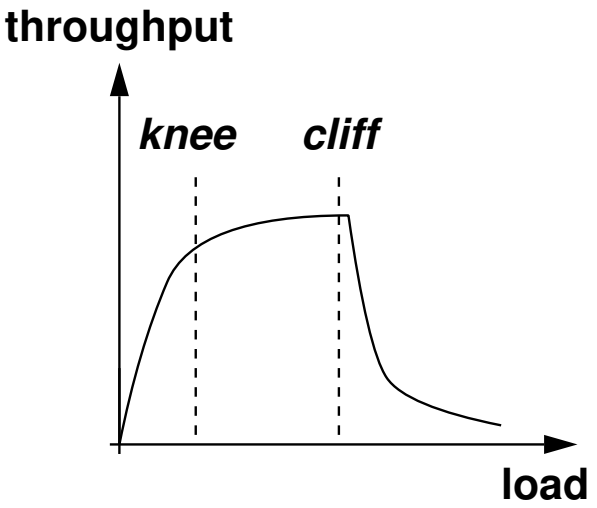


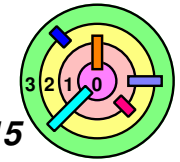
Thrashing

- Consider a system that has exactly two page frames:
  - process A has a page in frame 1
  - process B has a page in frame 2



- Process A references another page, causing a page fault
  - the page in frame 2 is removed from B and given to A
- Process B faults immediately; the page in frame 1 is given to B
- Process A resumes execution and faults again; the page in frame 1 is given back to A
- ...
  - neither processes makes progress
- Although this is a contrived example, it highlights the basic problem
  - need 3 physical page frames, but only 2 are available





The Working-Set Principle

- To deal with thrashing, the idea of *Working-Set* can be used
  - although it may be difficult to implement exactly
- The set of pages being used by a program (the working set) is relatively small and changes slowly with time
  - *WS(P,T)* is the set of *pages* used by process *P* over time period *T*
- Over time period *T*, *P* should be given *|WS(P,T)|* page frames
  - if space isn't available, then *P* should not run and should be *swapped out*
- If the sum of the working-set of all processes is less than the total amount of available physical memory
  - then thrashing cannot occur
  - using *Local Allocation* is a way to *reduce the chance* of *thrashing*