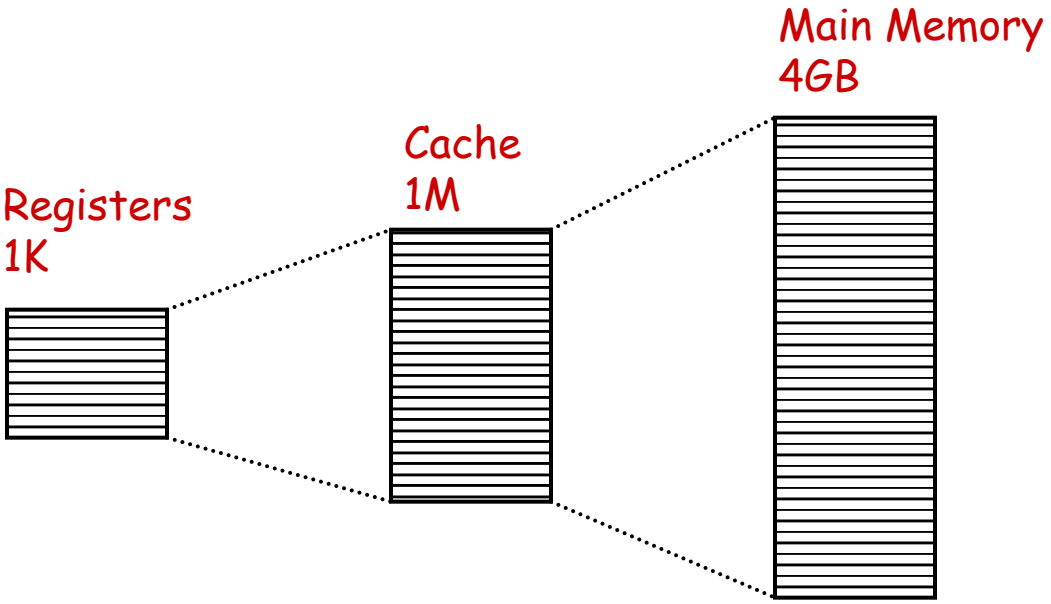
Virtual Memory

Chapter 10

The Memory Hierarchy

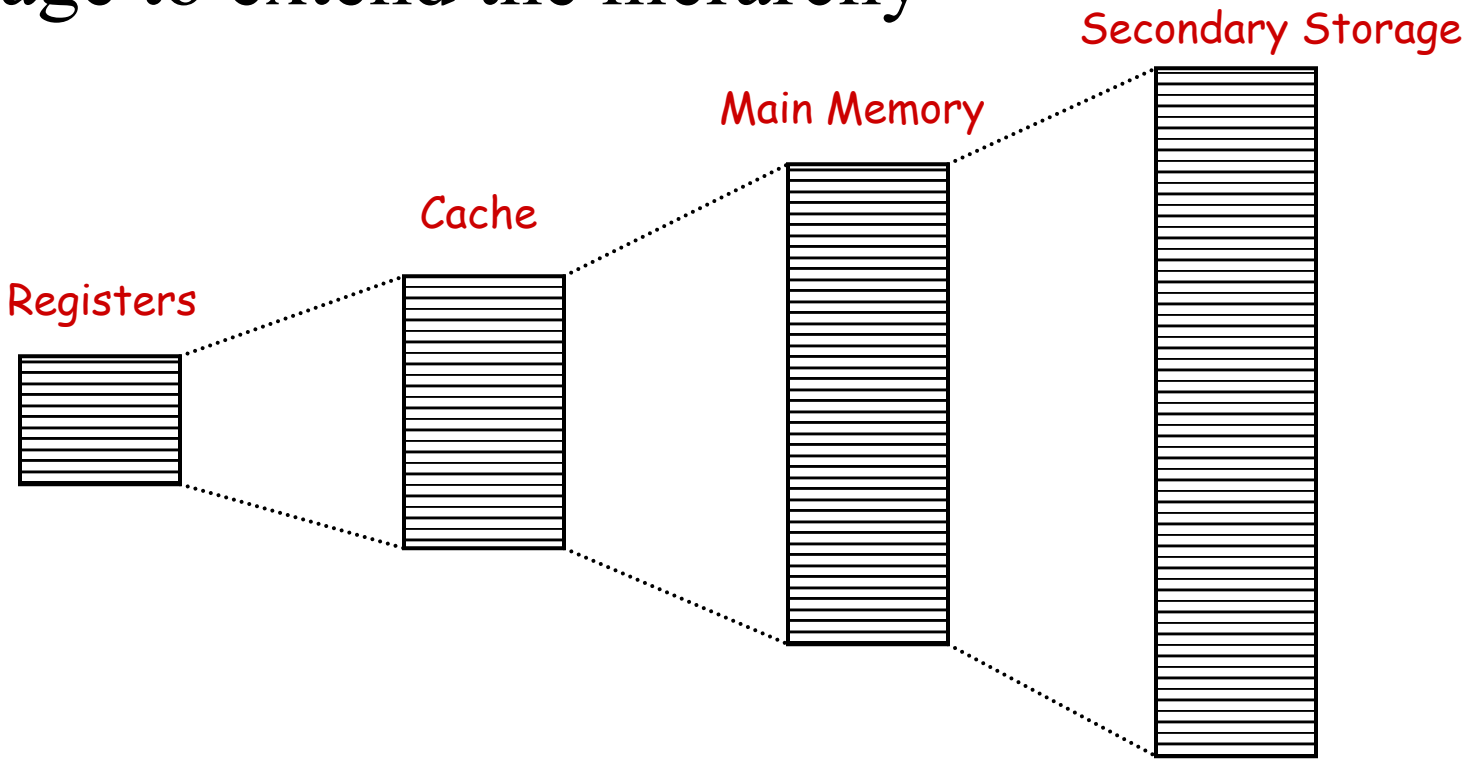
- Recall, different levels of storage, with different size & speed characteristics



- What if this isn't enough?

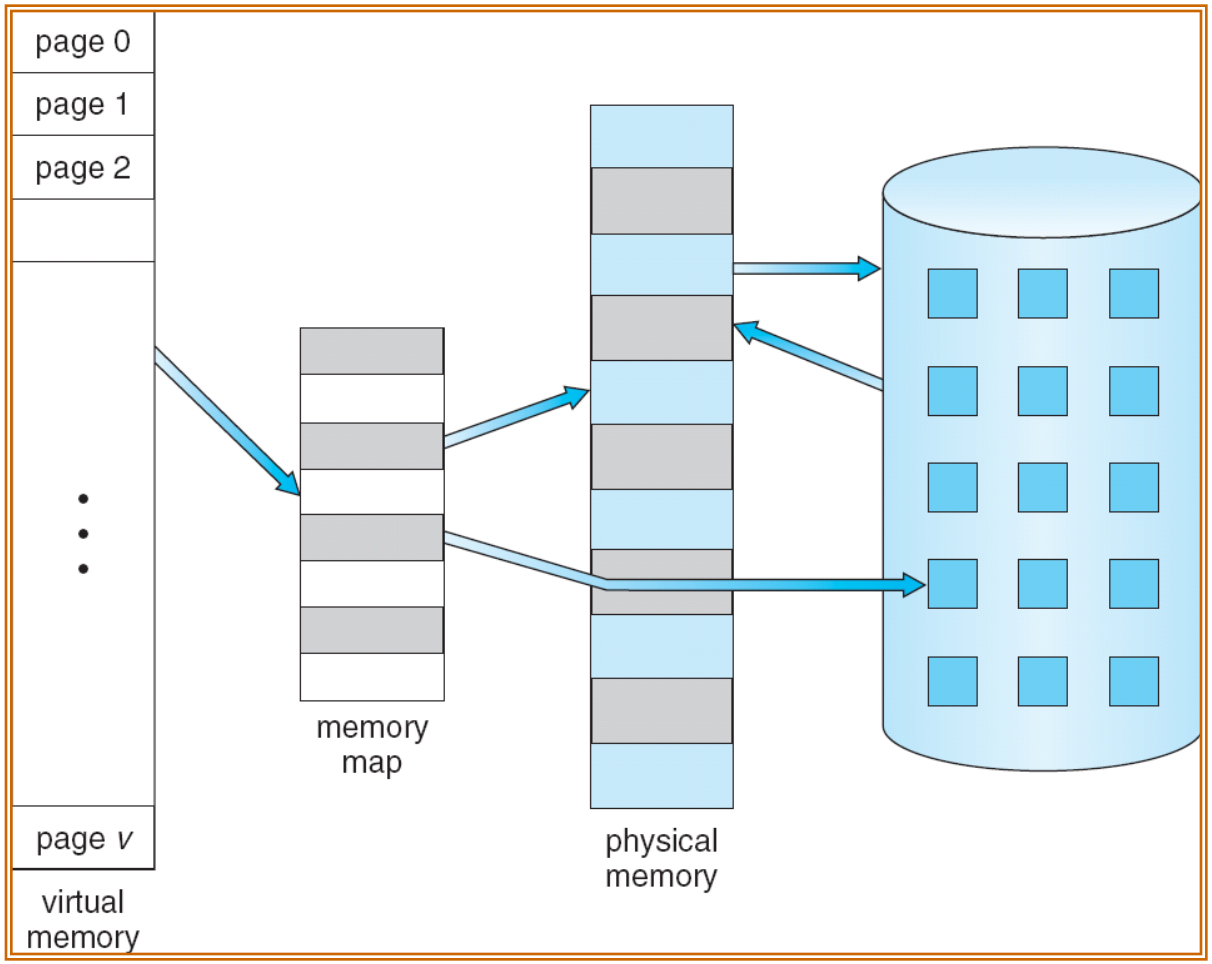
The Memory Hierarchy

- As with swapping, we can use secondary storage to extend the hierarchy



- But, now we can do it with finer granularity

Virtualizing Memory



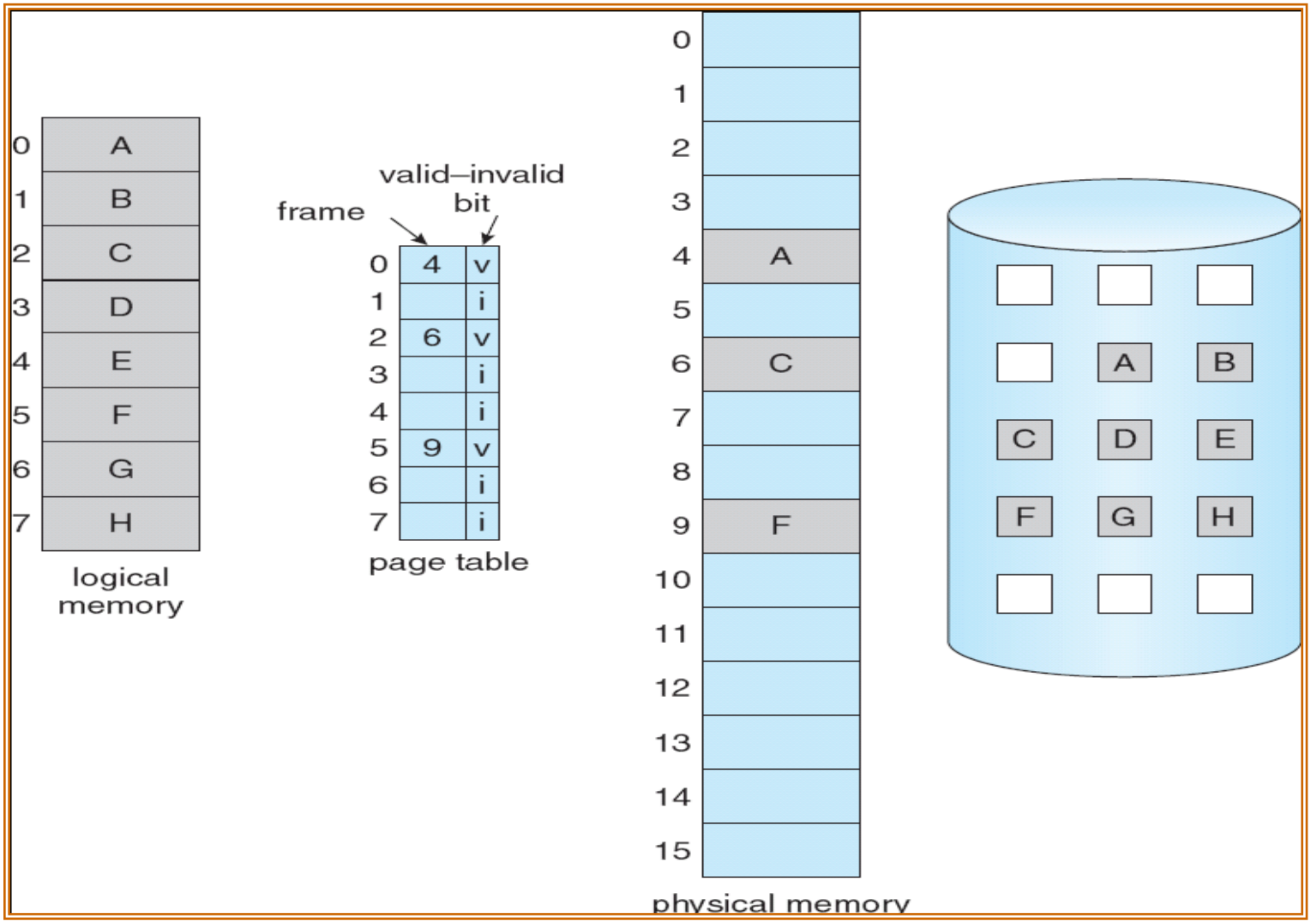
Demand Paging

- *Demand Paging*
  - Move seldom-used pages to backing store
  - *Memory resident* pages are the ones in main memory
  - Bring a page into memory when it's needed
    - How do we know when it's needed?
  - Less physical memory per process
  - *Page out* : copy a page from main memory to backing store
  - *Page in* : copy a page from backing store to main memory
- *Pure Demand Paging*
  - Don't even load pages into memory until they are first used
  - Less I/O (at process startup)
  - More concurrent processes, less physical memory per process

How Processes Demand their Pages

- If a page is on backing store, let's just mark it as invalid in the page table
- Then, when the process tries to use it
  - We get a trap to the kernel
  - That's called a *page fault*
- When process P page faults, the OS decides
  - Maybe the page isn't part of the logical address space for P
    Bad Process!
  - Or, maybe this page is part of P's address space, but it's on backing store
    Wait, let me go get that page for you right away!

Page Table under Demand Paging



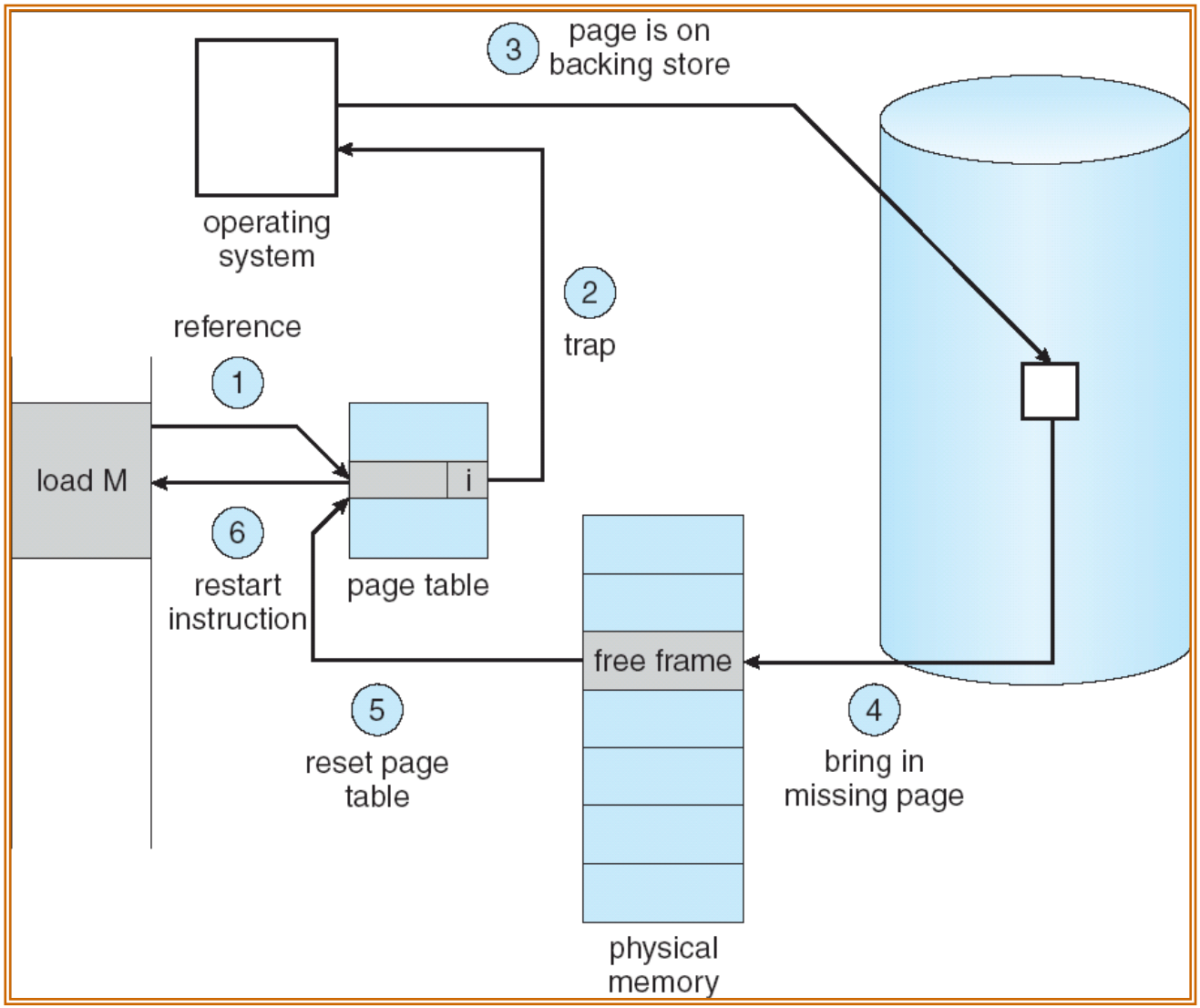
Handling a Page Fault

1. Some process P generates a page fault, trap to the OS
2. Operating system decides
   - Is this an invalid reference → maybe abort the process, signal it, etc.
3. Page is on backing store, find an empty frame
4. Mark process P as blocked for I/O
   Schedule I/O to load the page from backing store
   This I/O will take a while. What to do while we wait…

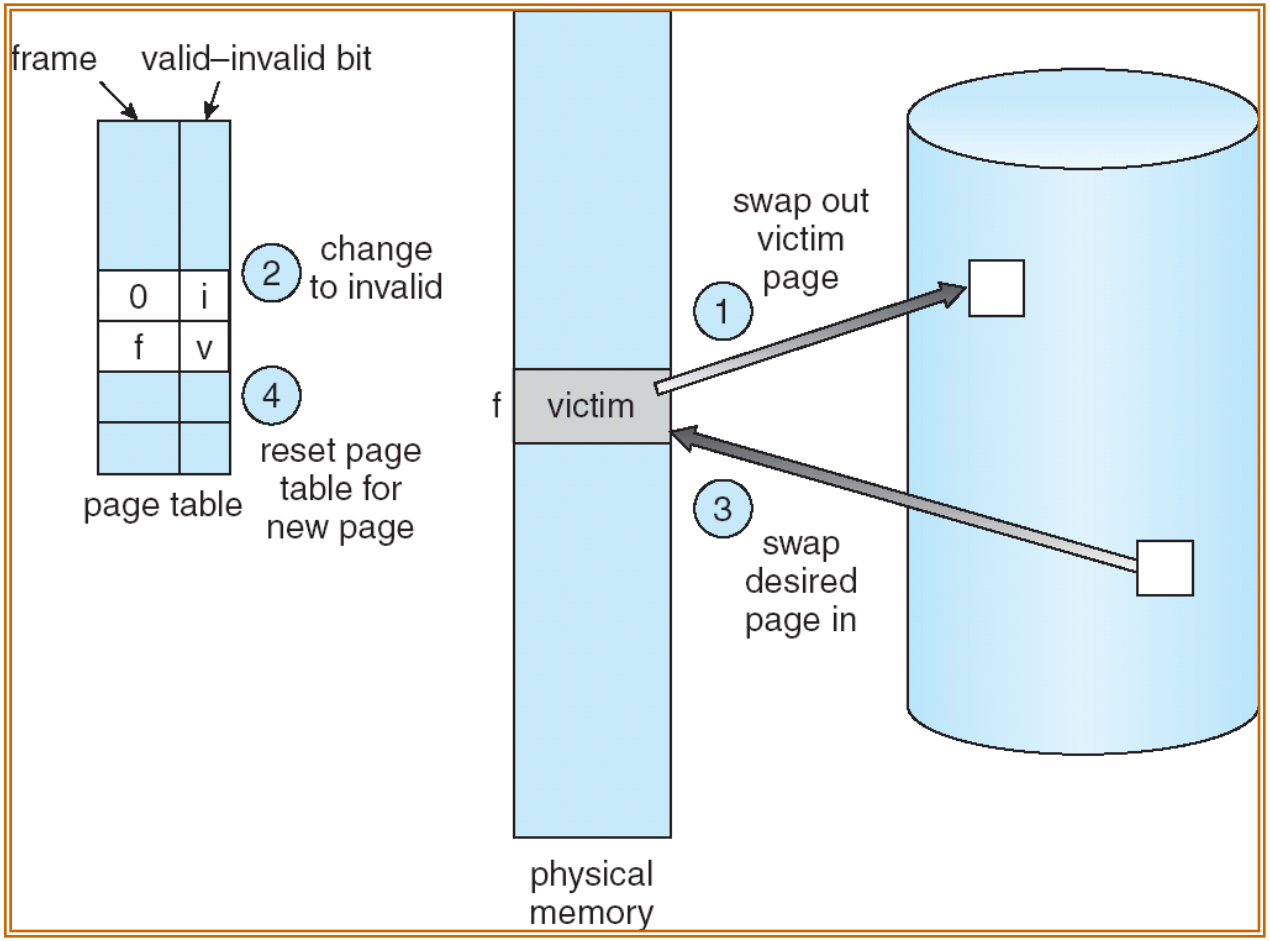
Imagine some elevator music here

5. Interrupt → I/O is done, fill in the page table entry
   Mark the page as valid
6. Mark the process as runnable, resuming the same instruction

Handling a Page Fault



Page Replacement with a Victim



Demand Paging and Performance

- Page fault rate: $0 \leq p \leq 1.0$
  - If $p = 0$, no page faults
  - If $p = 1.0$, every reference is a page fault
- We can compute *Effective Access Time* (EAT)

EAT = *(1-p) memory_access_time +*
p (page_fault_overhead +
page_out_time +
page_in_time +
fix_page_table_and_resume_time +
memory_access_time)

Demand Paging Performance

- Let's check this for plausible values
  - Memory access time: 200 ns
  - Average page fault overhead: 8 ms
    That's 8,000,000 ns
  - If $p = 0.001$,
    EAT = 0.999 x 200 ns + 0.001 x 8,000,000 ns
    EAT = about 8,200 ns
  - That's 41 times slower
  - So, we need a **much** lower page fault rate

Keeping the Page Fault Rate Low

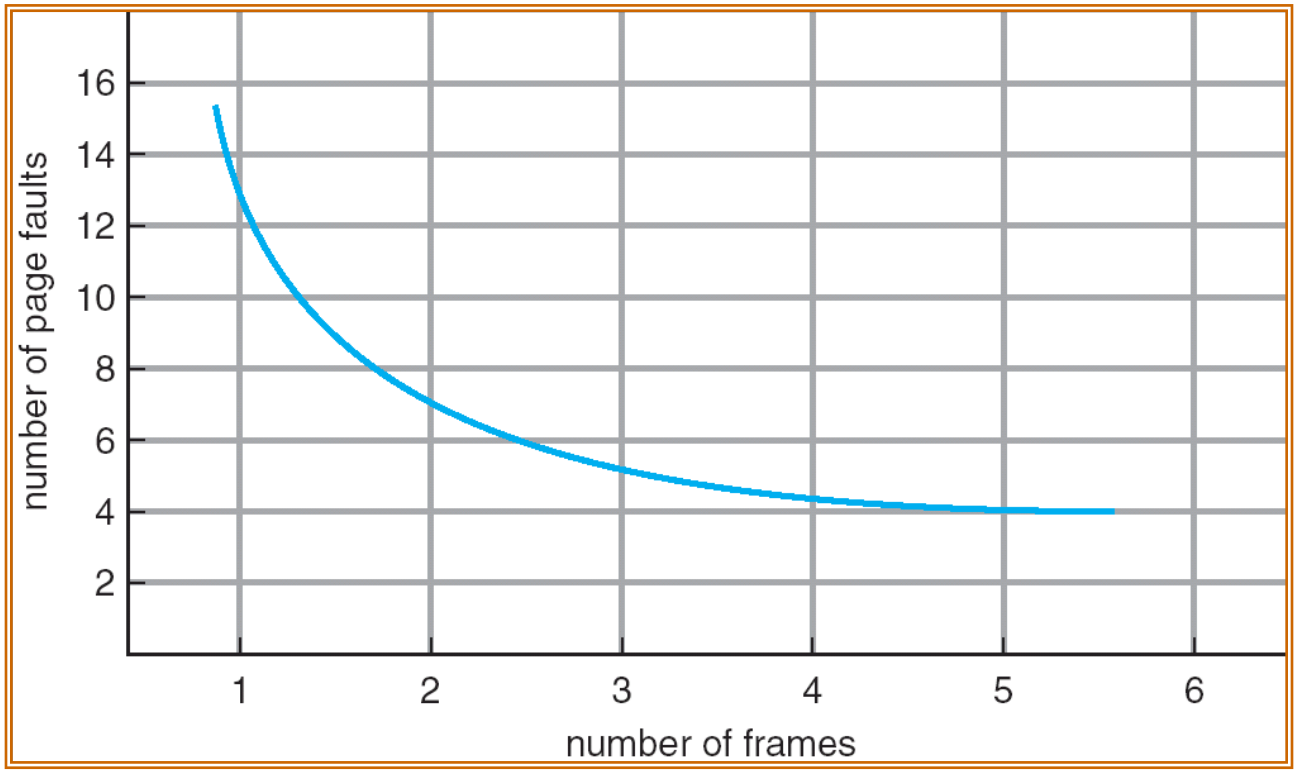
- How can the OS control the page fault rate?
  - Number of frames it allocates to each process
  - How it chooses a *victim*
- Choosing a good victim
  - Don't want one that will be needed milliseconds later
    - Want to choose an idle page
    - Need to choose quickly
  - Need an efficient way to keep up with what pages are actively used
  - Could use a *local* or a *global* page replacement policy

Page Replacement Algorithms

- A *page replacement algorithm* is just a rule for choosing a victim
- How do we know if we have a good algorithm?
  - We could measure the page fault rate
  - Generate a *reference string* of the sequence of pages used by a program
  - We just care about which pages are referenced, not individual bytes
  - Run the page replacement algorithm on that string, see how many page faults it produces

Page Fault Rate and Number Frames

- In general, we expect fewer page faults with more frames
- But, a good page replacement algorithm can let us get the most out of a limited number of frames



First-In-First-Out (FIFO) Page Replacement

- *FIFO* : Choose a victim that's been around the longest
- Let's try it out:
  - Pretend we have three memory frames available
  - Reference string: 1 2 3 4 1 2 5 1 2 3 4 5

  - (should get 9 page faults)

First-In-First-Out (FIFO) Page Replacement

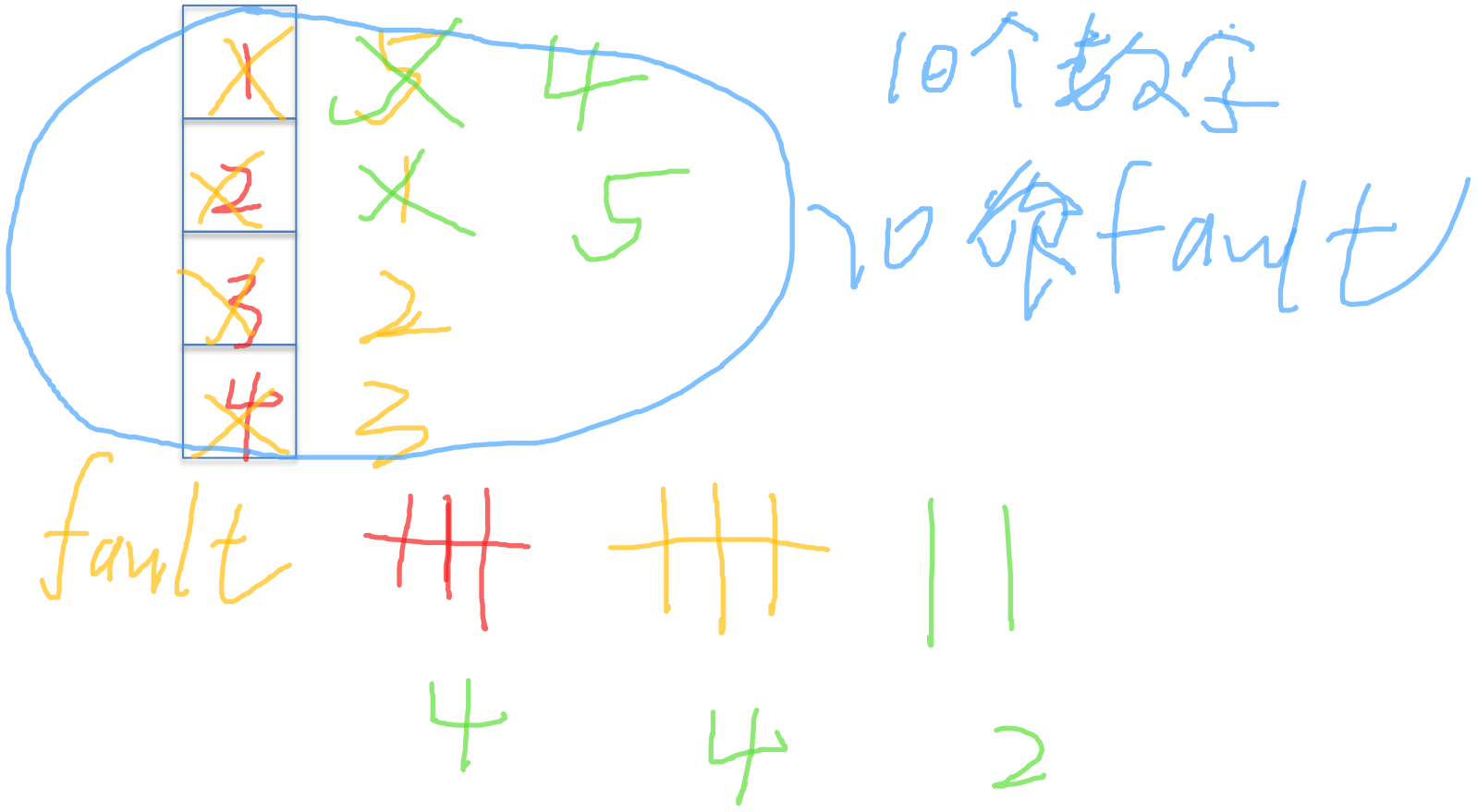
- Reference string: 1 2 3 4 1 2 5 1 2 3 4 5

Page cnt : 9 page fault

4 4 1 fault

One More Frame?

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



More Page faults with Additional Frames

- This is a surprise
  - We got 9 page faults with 3 frames
  - … and 10 page faults with 4 frames
- There's a name for this behavior: *Belady's Anomaly*
  - More frames can yield more page faults

Belady's Anomaly and Stack Algorithms

- A *stack algorithm*, is a page replacement algorithm that follows certain rules
  - It's immune to Belady's anomaly
  - The set of pages in memory when we have *f* frames
    is always a subset of
    The set of pages in memory when we have *f+1* frames
  - Said a different way, having more frames will let the algorithm keep additional pages in memory, but, it will never choose to throw out a page that would have remained in memory with fewer frames

FIFO, The Good and the Bad

- FIFO makes victim selection fast
- And, it only requires work when a page fault actually occurs

- But, it makes no effort to choose a good victim

- What would make a good victim?
  - A page that will never be used again.
  - Or, at least a page that won't be used for a long time.

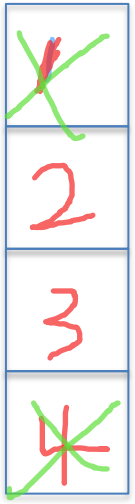
Optimal Page Replacement Algorithm

- That's the optimal algorithm (*OPT*), the victim is the page that won't be used for the longest time
- As the name suggests, it's optimal in number of page faults
- Let's try it out.
  - Four memory frames
  - Same reference string:
    1 2 3 4 1 2 5 1 2 3 4 5

Opt

choose it longest time

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



6 page faults

Page Replacement Exercises

- Pretend we have three frames
- Consider the following reference string:
  7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

- How many page faults do we get with FIFO?
- How many page faults with OPT?

FIFO

- Reference string:
  7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

OPT

- Reference string:
  7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

The Optimal Algorithm

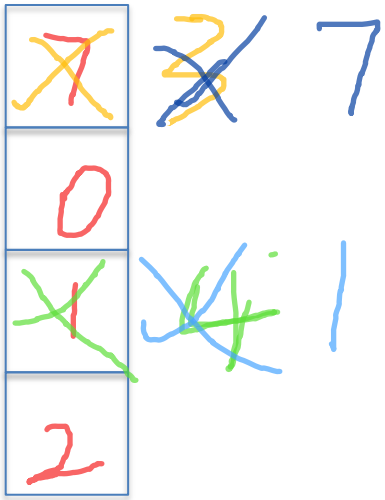
- So, we're done, right? Just implement OPT.
- Well, there's a problem
  - It requires knowledge of the future
  - This is a familiar problem (e.g., in CPU scheduling)
  - How have we handled it before?
  - Let's use past behavior to predict the future

Least-Recently Used

- Kind of a backward-looking approximation of OPT
- Choose a victim that hasn't been used for the longest time
- Let's try it out.
  - Four memory frames
  - Same reference string:
    7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

LRU

- Reference string:
  7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



8 page faults

Implementing LRU

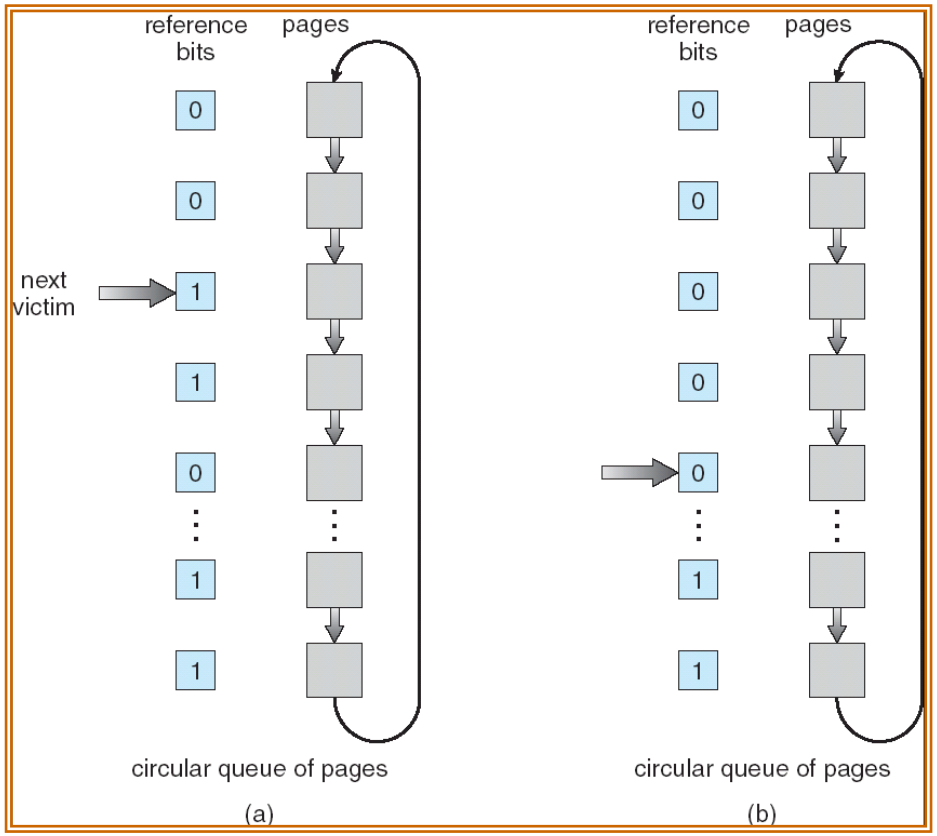
- How can we implement this?
- How about using a timestamp?
  - Increment the timestamp (just a counter) on every memory reference
  - Every page has a copy of the timestamp for when it was last accessed
  - Always replace the page with the oldest timestamp
- Is this realistic?
- OK, what now?
  - For practical applications, seems like we need to approximate LRU

LRU Approximations

- Recall, every page has a **reference bit**
  - The hardware sets it when the page is referenced
  - The OS will clear this bit, let processes run for a while
  - … then, see which pages are being used and which ones aren't
- The *Second Chance* algorithm:
  - Let them stay in memory (for now) if the reference bit is set.
  - Clear the bit and move on to the next (potential) victim.
  - Sometimes called the *clock algorithm*

The Clock Algorithm

- Keep a pointer to the next victim
  - If the reference bit is set, clear the bit and move on
  - If reference bit is clear, that's your victim
- Observe, frequently used pages will stay in memory
- It's computationally cheap, like FIFO



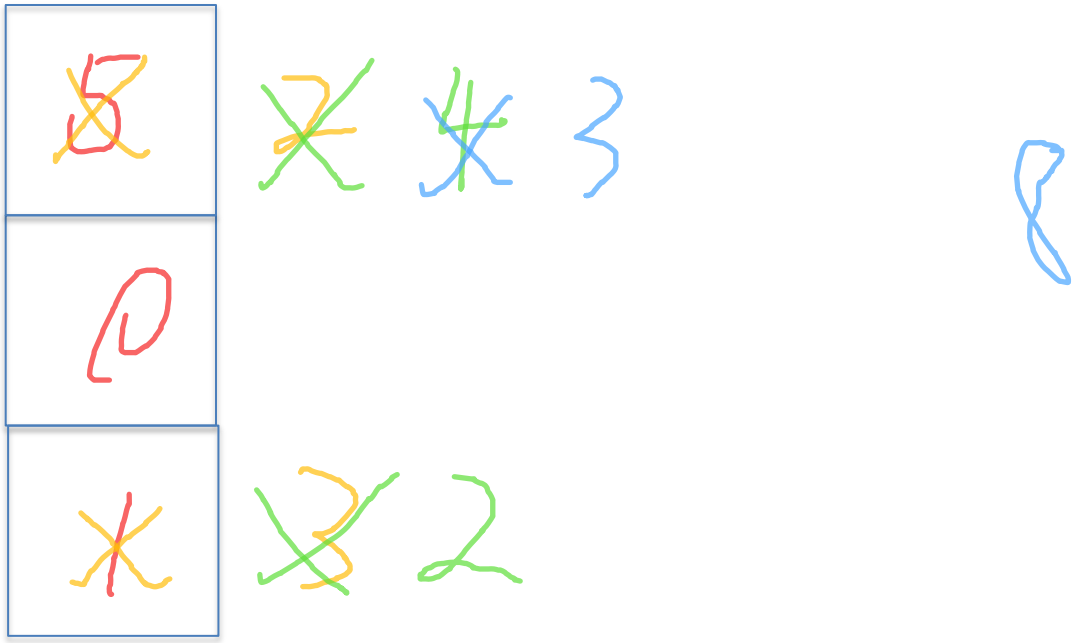
More Page Replacement Exercises

- Pretend we have 3 frames
- Reference string:
  5 0 1 2 0 3 0 4 0 2 0 3 0 3 2

- How many page faults under LRU?
- How many under second chance?

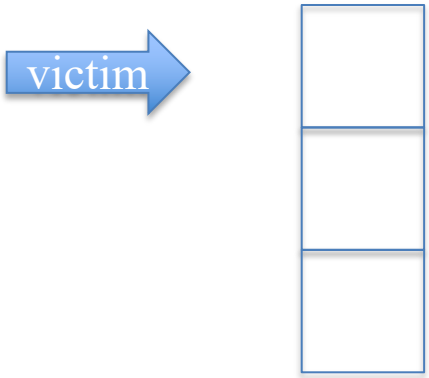
LRU

- Reference string:
  5 0 1 2 0 3 0 4 0 2 0 3 0 3 2



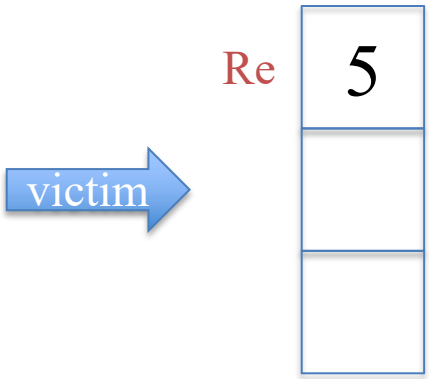
Second Chance

- Reference string:
  5 0 1 2 0 3 0 4 0 2 0 3 0 3 2



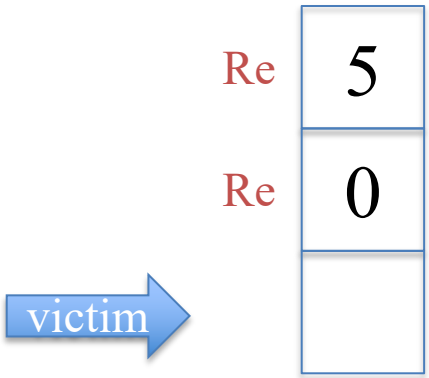
Second Chance

- Reference string:
  5 0 1 2 0 3 0 4 0 2 0 3 0 3 2



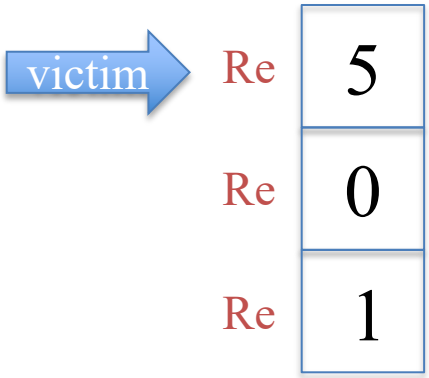
Second Chance

- Reference string:
  5 0 1 2 0 3 0 4 0 2 0 3 0 3 2



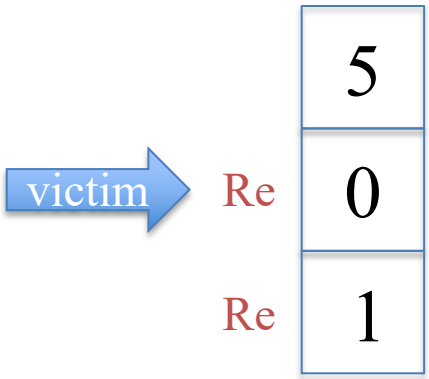
Second Chance

- Reference string:
  5 0 1 2 0 3 0 4 0 2 0 3 0 3 2



Second Chance

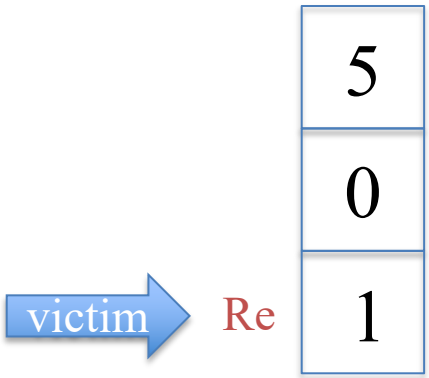
- Reference string:
  5 0 1 2 0 3 0 4 0 2 0 3 0 3 2



clear reference bit to 0

Second Chance

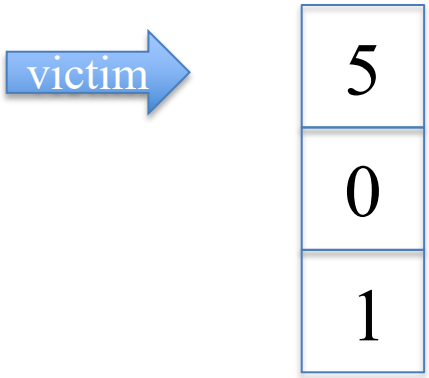
- Reference string:
  5 0 1 2 0 3 0 4 0 2 0 3 0 3 2



clear ref-bit to 0

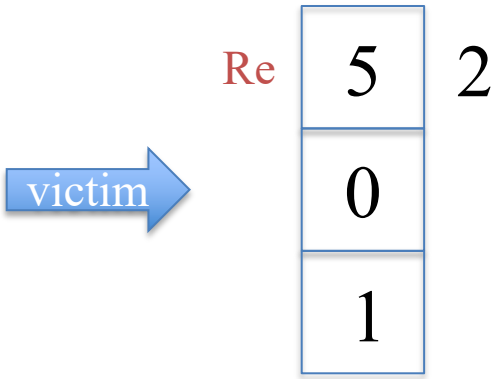
Second Chance

- Reference string:
  5 0 1 2 0 3 0 4 0 2 0 3 0 3 2



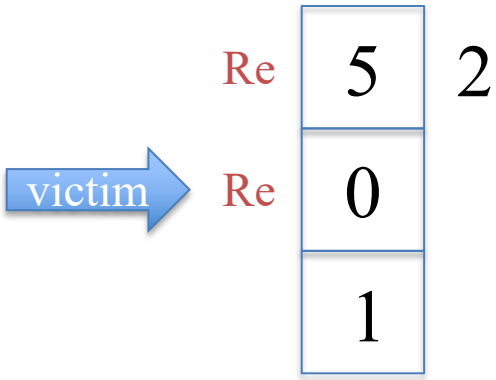
Second Chance

- Reference string:
  5 0 1 2 0 3 0 4 0 2 0 3 0 3 2



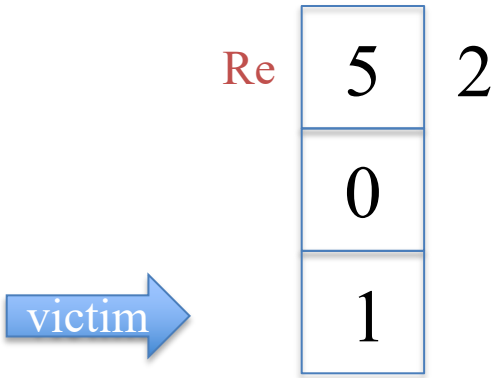
Second Chance

- Reference string:
  5 0 1 2 0 3 0 4 0 2 0 3 0 3 2



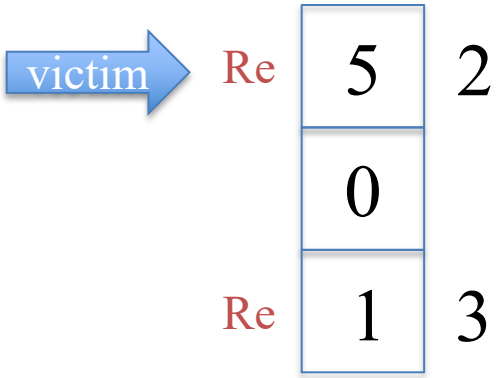
Second Chance

- Reference string:
  5 0 1 2 0 3 0 4 0 2 0 3 0 3 2



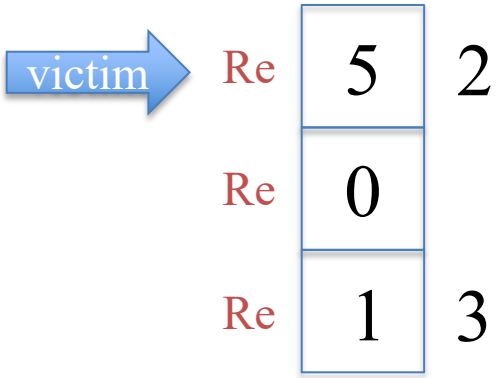
Second Chance

- Reference string:
  5 0 1 2 0 3 0 4 0 2 0 3 0 3 2



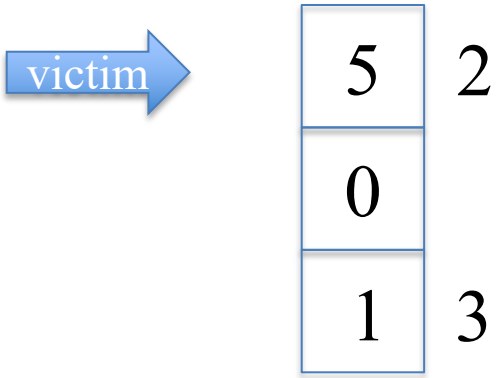
Second Chance

- Reference string:
  5 0 1 2 0 3 0 4 0 2 0 3 0 3 2



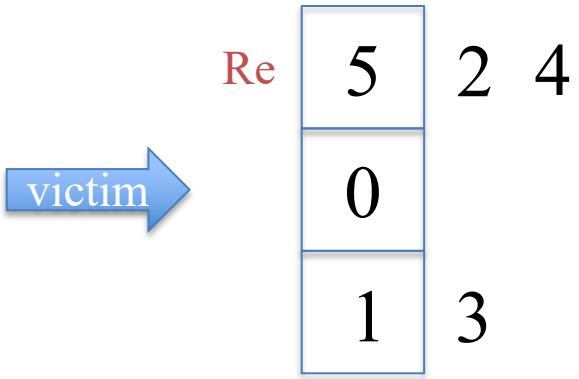
Second Chance

- Reference string:
  5 0 1 2 0 3 0 4 0 2 0 3 0 3 2



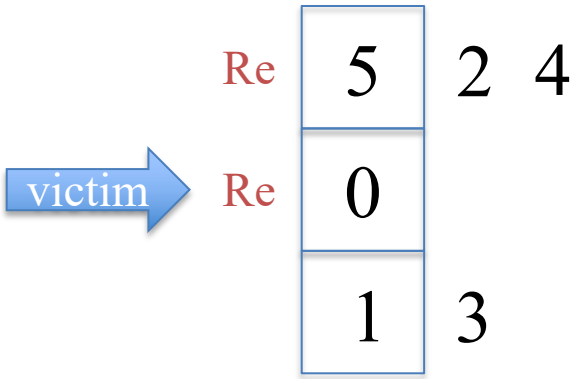
Second Chance

- Reference string:
  5 0 1 2 0 3 0 4 0 2 0 3 0 3 2



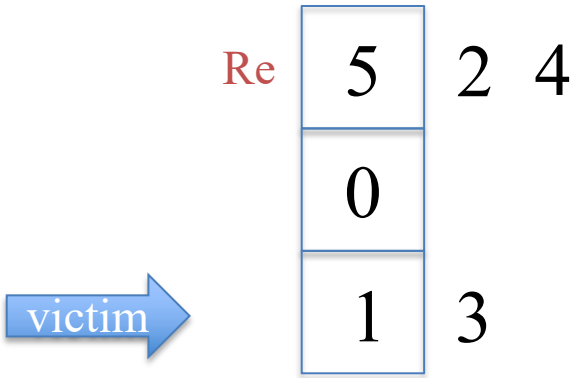
Second Chance

- Reference string:
  5 0 1 2 0 3 0 4 0 2 0 3 0 3 2



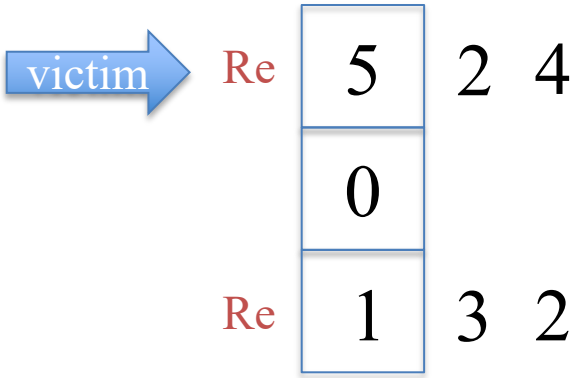
Second Chance

- Reference string:
  5 0 1 2 0 3 0 4 0 2 0 3 0 3 2



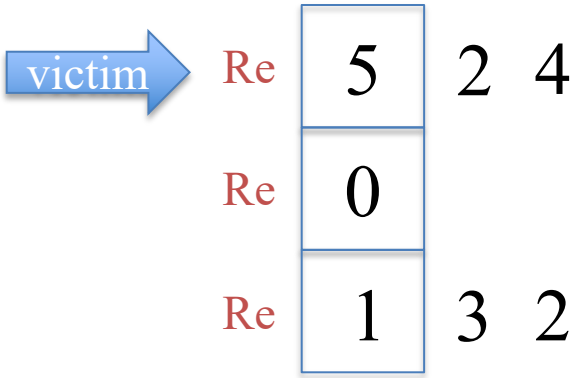
Second Chance

- Reference string:
  5 0 1 2 0 3 0 4 0 2 0 3 0 3 2



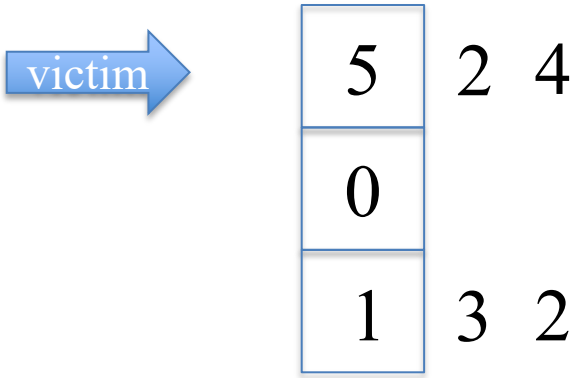
Second Chance

- Reference string:
  5 0 1 2 0 3 0 4 0 2 0 3 0 3 2



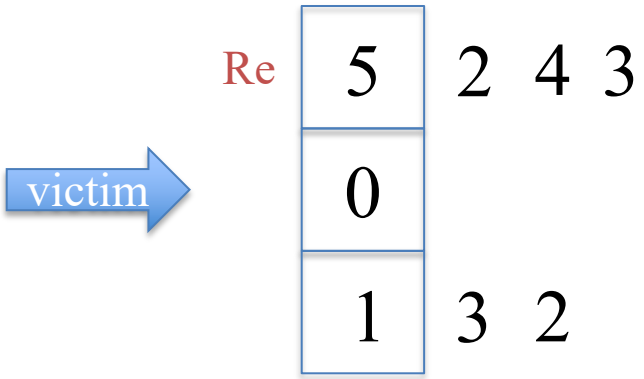
Second Chance

- Reference string:
  5 0 1 2 0 3 0 4 0 2 0 3 0 3 2



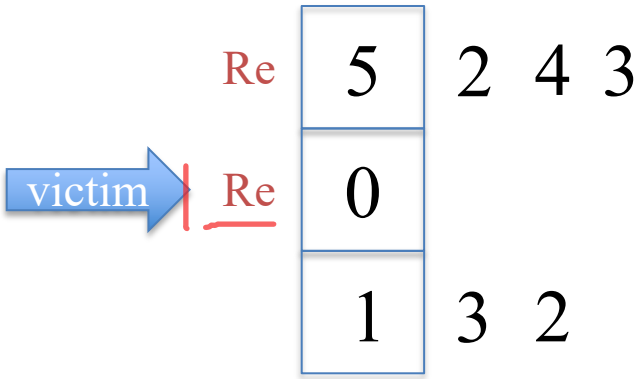
Second Chance

- Reference string:
  5 0 1 2 0 3 0 4 0 2 0 3 0 3 2



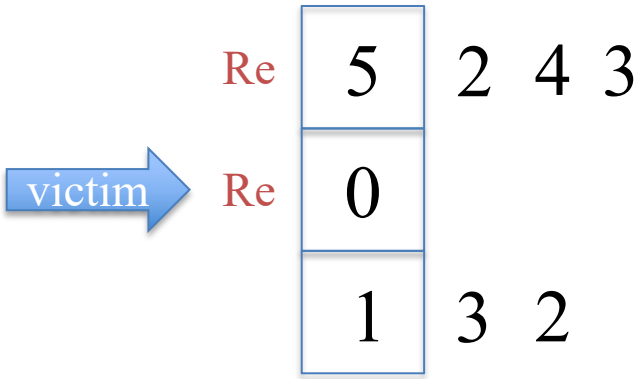
Second Chance

- Reference string:
  5 0 1 2 0 3 0 4 0 2 0 3 0 3 2



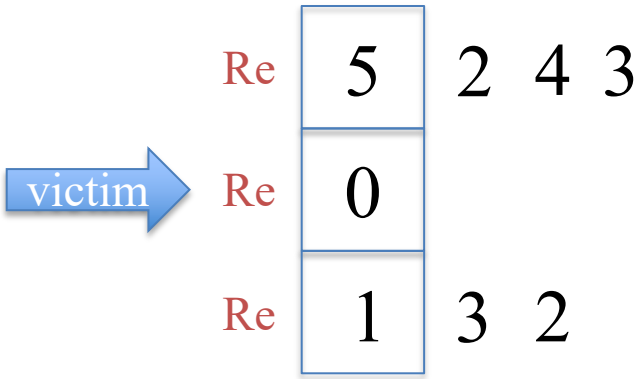
Second Chance

- Reference string:
  5 0 1 2 0 3 0 4 0 2 0 3 0 3 2



Second Chance

- Reference string:
  5 0 1 2 0 3 0 4 0 2 0 3 0 3 2



Alternatives to Second Chance

- Counting Algorithms
  - The reference bit is like one bit of usage history
  - The idea, keep a longer history to choose a better victim
  - Periodically check reference bits
    - Maintain a per-page counter, increment if bit is set
    - Clear the bit, so you can check again later
- Least Frequently Used (LFU)
  - Count the number of references to each page
  - Replace the page with the smallest count
  - Potential problems?

Additional Reference Bits

- A better idea: record a history of per/page reference bit values
  - Maintain an n-bit unsigned integer for each page
  - Shift the counter to the right, and write the reference bit into the high-order position
  - Clear the reference bit and see if the page gets used again.

00111111	Used for a while, but not recently
11000000	Idle for a while, but used recently
11000110	Used recently and in the past
11111111	Used all the time

- So, we just choose the page with the smallest numeric value
- Clever, but still kind of expensive

The Enhanced Second-Chance Algorithm

- The Idea, use both the reference and dirty bits
  - Recall, the dirty bit is set (by hardware) when a page is written to
  - (is it OK to end the previous bullet with a preposition? How would you fix it?)
  - Use the dirty bit to tell if an in-memory page matches a copy on disk
  - The reference bit tells which pages are being used
  - The dirty bit tells us which pages are more expensive to replace

The Enhanced Second-Chance Algorithm

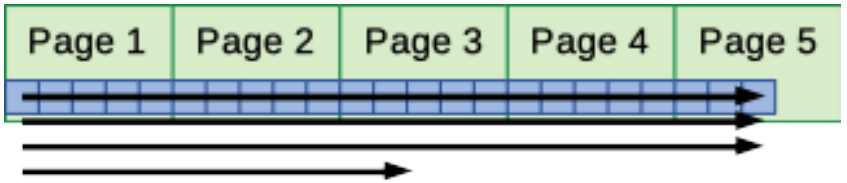
- So, we get four flavors of pages:

Re	M	
0	0	Neither modified nor recently used
0	1	Need to write it out, but not recently used
1	0	Recently used, but the copy on backing store is still good
1	1	Recently used, and in-memory copy doesn't match backing store

- Which ones make the best victims?
- Also, this will give us something to do with an idle backing store device

LRU Disasters

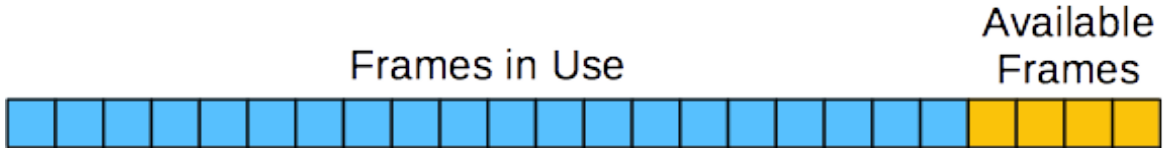
- LRU can exhibit pathological behavior when:
  - You access pages in the same order, over and over.
  - And you have almost enough frames to hold them all.



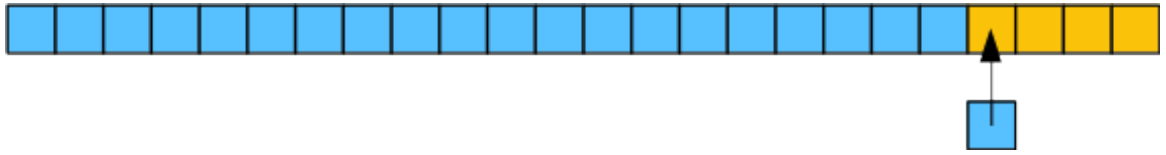
- Some systems switch between alternative (local) page replacement algorithms
  - … trying to reduce the page fault rate.
  - … including random, if nothing else is working.

Page Buffering

- It pays to keep some free frames available.



- When a page fault occurs, you can use one of these immediately to bring in the faulted page.

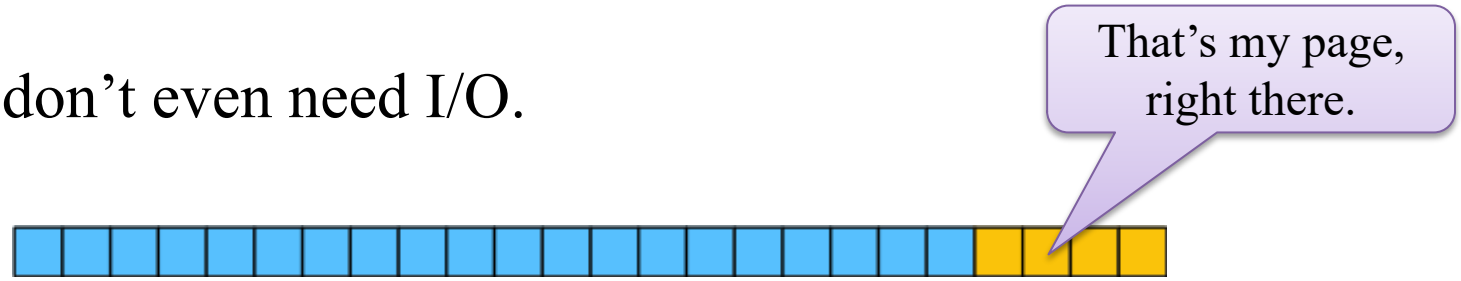


- Then, afterward, write out an idle page to maintain the pool of free frames

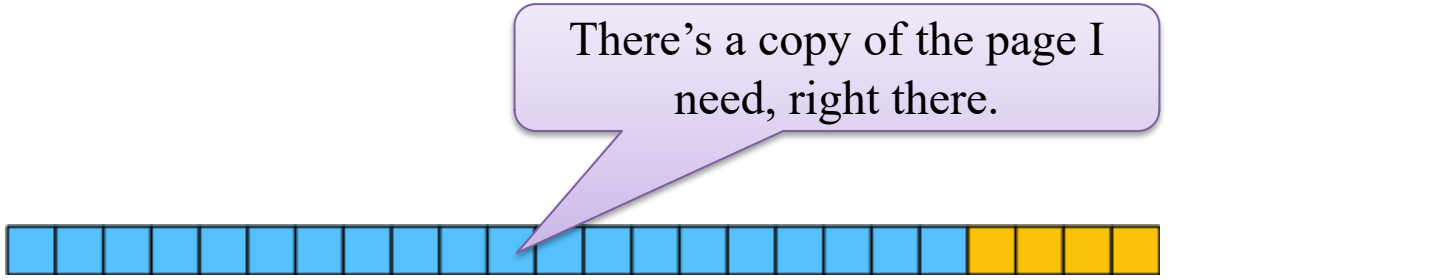
- It's like paying in advance for half the page fault.

Minor Page Faults

- Spare frames may still contain a page of a process.
- If that process page faults, it may be very fast to service the fault.
  - We don't even need I/O.



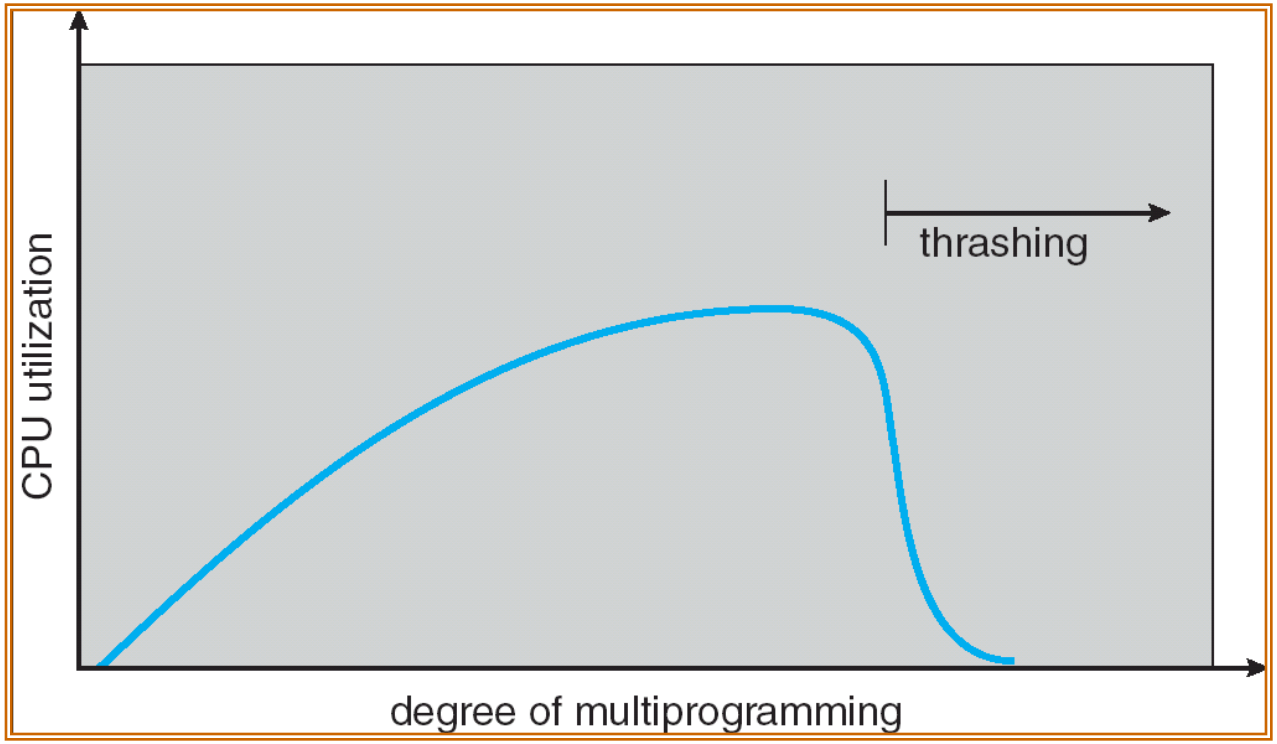
- This is called a *minor page fault*.
- The same kind of thing can happen when faulting on a shared page.



Running Short on Frames

- In the early days of computing
  - Resource utilization was a primary concern
  - CPU underutilized → start more CPU-bound jobs
- If a process doesn't have enough frames, it starts to page fault, a lot.
  - This leads to **high I/O utilization**
  - … and **low CPU utilization**
  - So, add some more CPU-bound jobs to increase CPU utilization?
  - Bad idea it turns more CPU-bound jobs into I/O-bound jobs
- This is *thrashing*
  - The system (or just one process) isn't getting much work done
  - … because it's spending all its time servicing page faults.

Thrashing Behavior



Dealing with Thrashing

- Can we tell when a system is thrashing?
  - Is there a difference between a lot of I/O-bound jobs and thrashing?
  - Yes, look at the cause of the I/O
  - If it's mostly page faults, then you're thrashing
  - So, we can monitor the page fault rate
- Local vs. global page replacement
  - Global replacement can let thrashing spread from one process to others
  - So, just use local?
  - Maybe. But then we have to decide how many frames each process gets.

Frames per Process

- *Proportional Allocation* : give each process a number of frames proportional to its total memory size.

Minimum Frames per Process

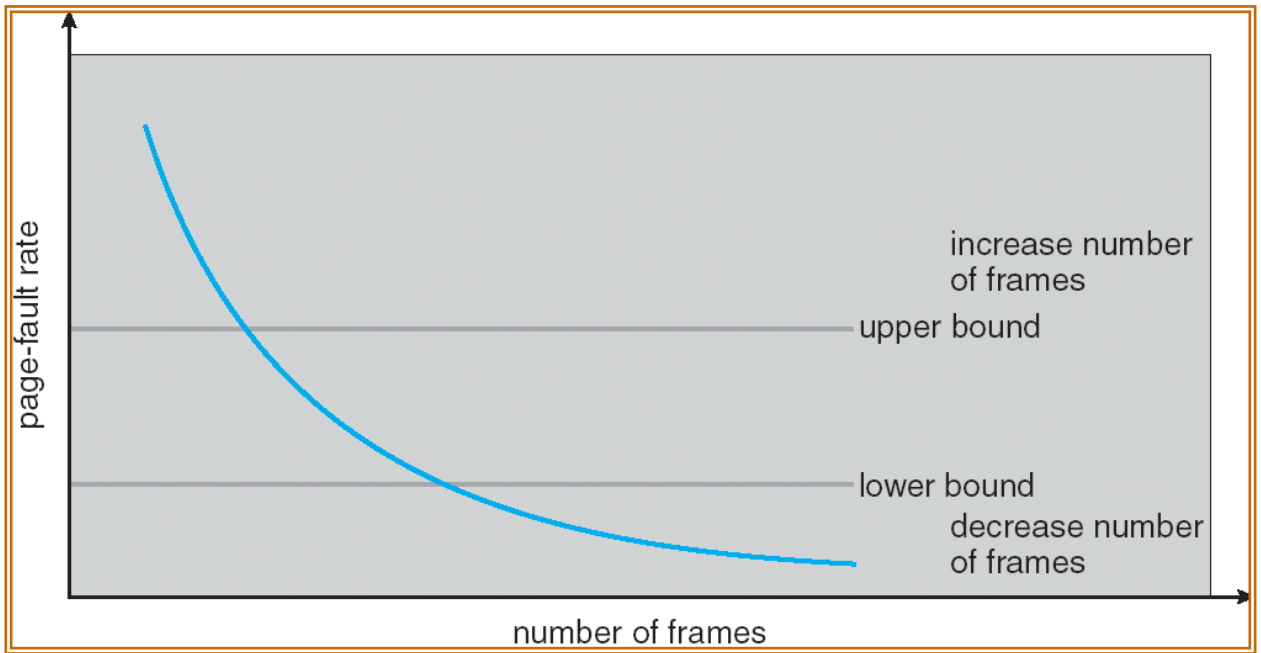
- There is a minimum number of frames each process **must** have
  - How many page faults can one instruction cause?
  - Maybe the instruction isn't memory resident.
  - Also, maybe also some parts of the page table aren't.
  - And, maybe the destination isn't … and maybe it spans two pages.
  - But … there's a maximum limit on this.
  - Depends on the CPU.
  - OS **must** give at least this many frames.
- Of course, a process may need a lot more frames to run efficiently

The Working Set

- A process' *working set* is the set of pages it's actively using right now
- We can measure a process' working set
  - Identify all pages used in the most recent 1,000,000 memory references
  - If we can't keep the working set in memory, the process won't run efficiently
  - We could measure the working set … but we probably wouldn't want to
  - Programs that exhibit better locality of reference will have smaller working sets
- As a program runs, its working set will change
  - Our frame allocation policy should probably adapt to this

Adapting to Page Fault Frequency

- We can adapt to changes in the working set … even without computing the working set
  - Too many page faults → I need more frames
  - Too few → maybe I could get by with fewer frames



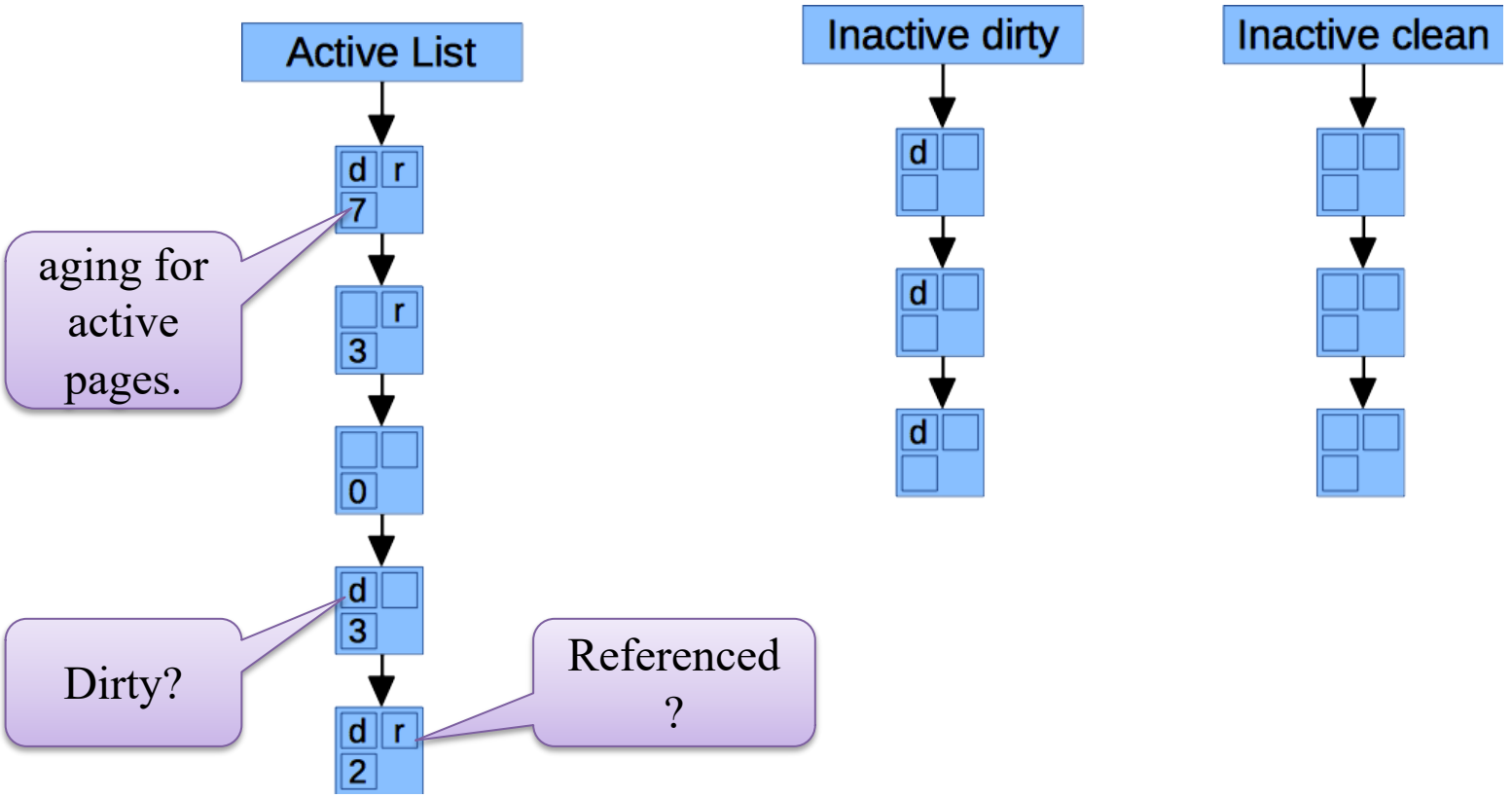
Locality and the Programmer

- The OS is concerned with performance of the system as a whole
- Me, I'm just concerned with my program
  - What do I remember? Effective access time is affected by two things:
    - If I get a TLB miss, my access time will 2 or 3 times slower
    - If I get a page fault, my access time may be a million times slower
  - Fortunately, coding to promote locality helps with both of these.
- Some coding techniques and data structures exhibit better locality
  - Copy an array?
  - Copy a linked list?
  - Repeated hash table queries?
  - Quicksort?
  - Copy a matrix row-by-row?
  - Copy a matrix column-by-column?
  - Matrix multiplication? (example)

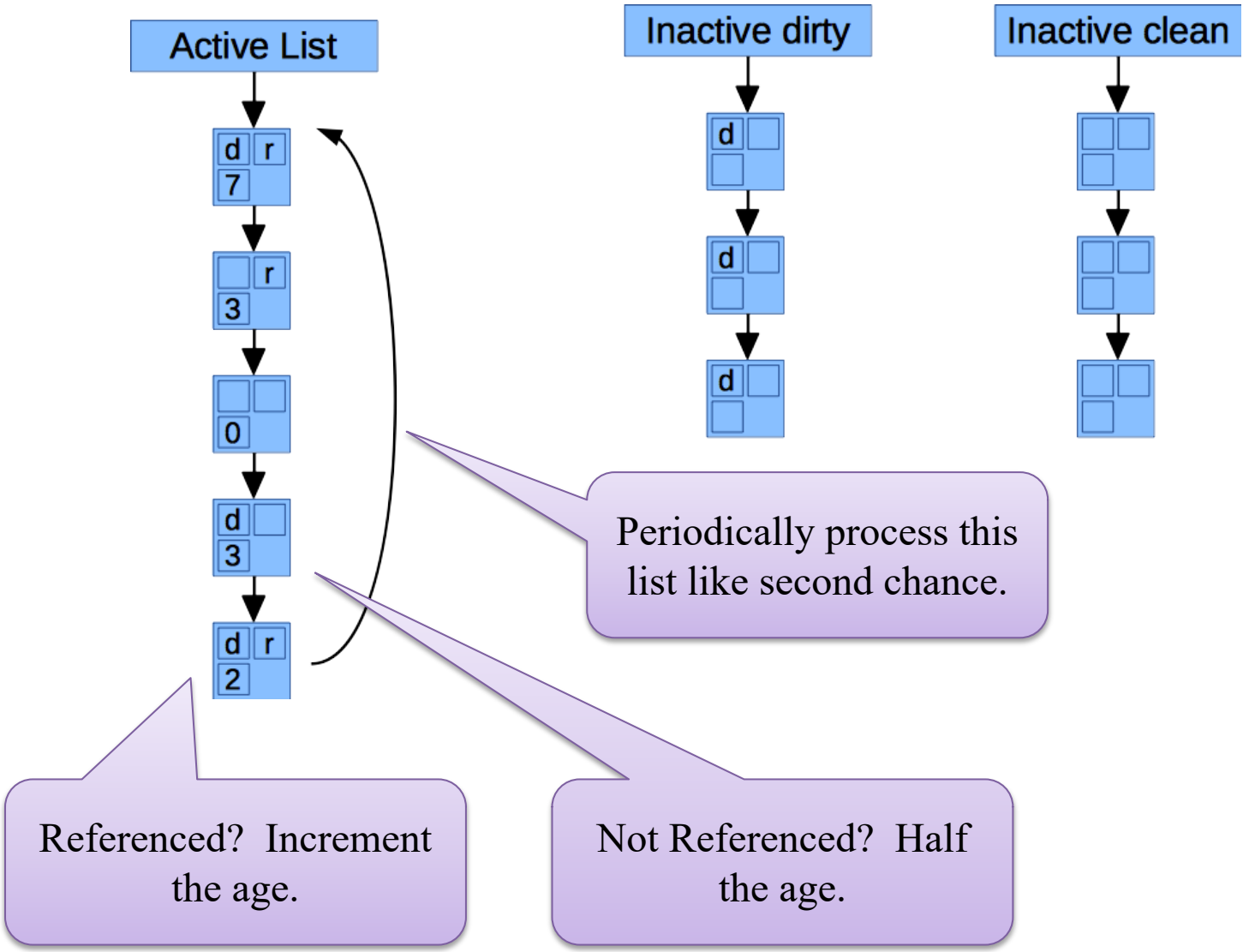
Page Replacement in Linux 2.4

- Want to see a real example (although maybe a little dated)
- Kernel maintains several lists of pages
  - *Active list* : pages that aren't (currently) candidates for replacement
  - Inactive lists : pages that would make good victims
    - *Clean list* : pages that are identical to a copy on backing store.
    - *Dirty list* : pages that aren't

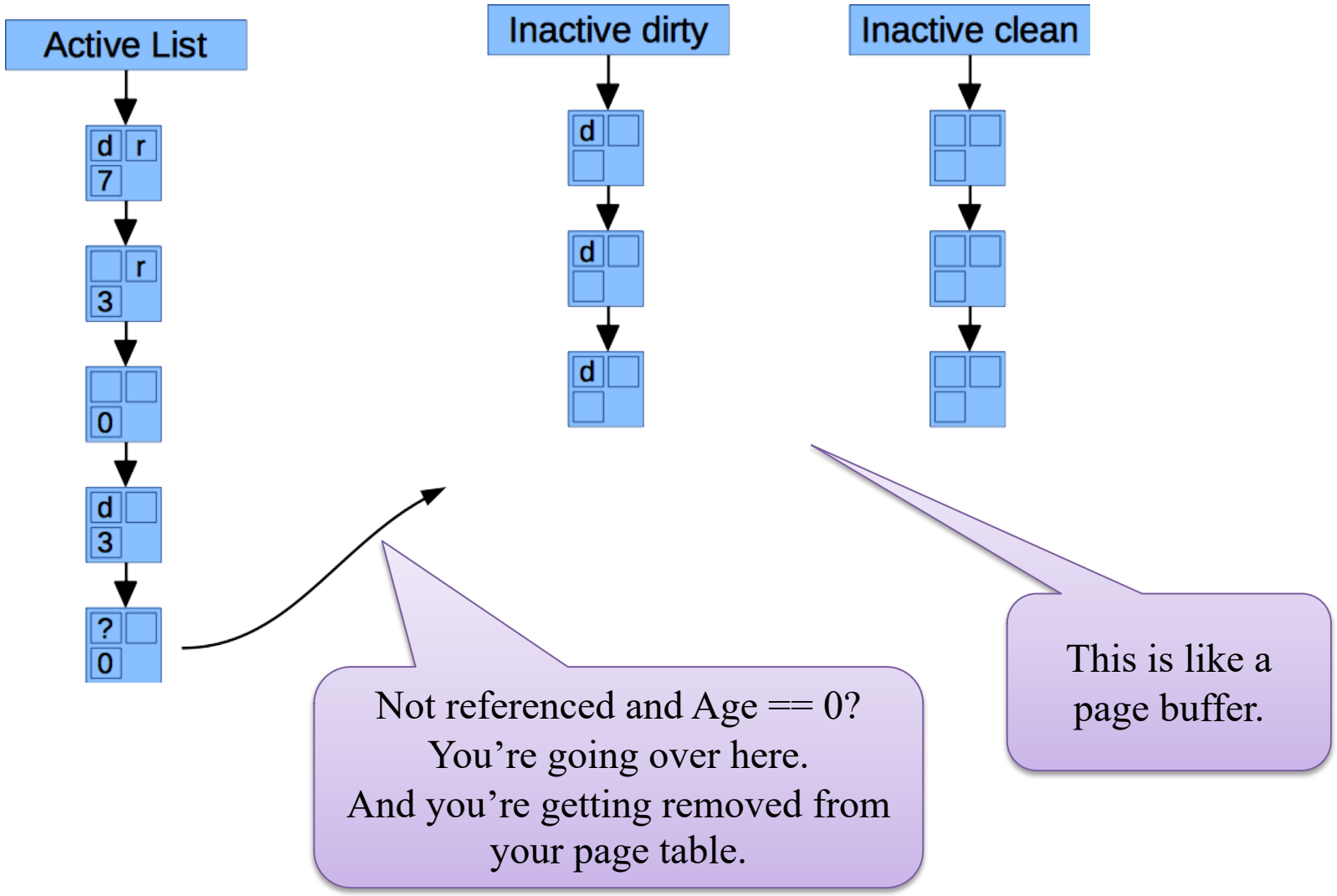
Page Replacement in Linux 2.4



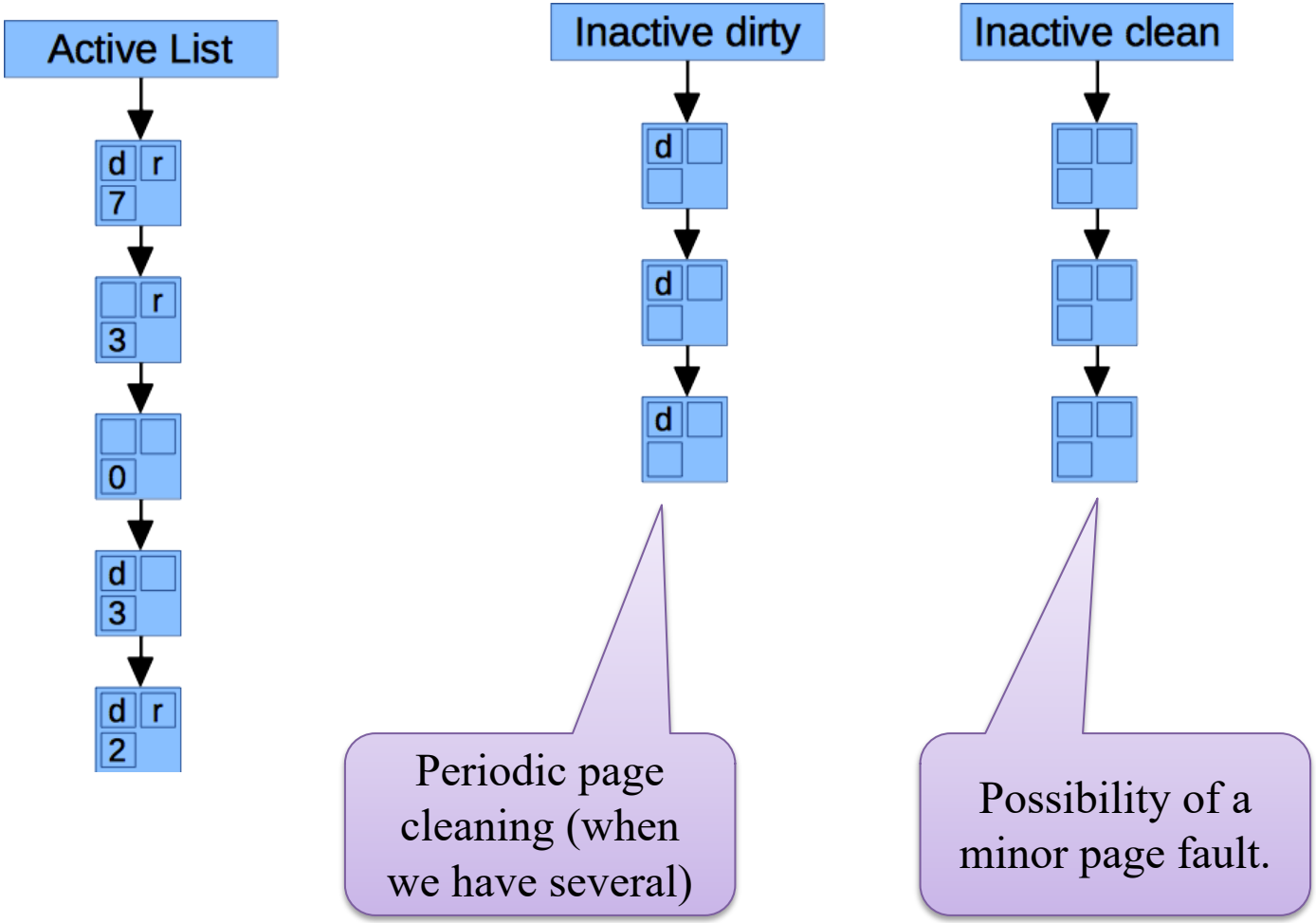
Page Replacement in Linux 2.4



Page Replacement in Linux 2.4



Page Replacement in Linux 2.4



What We've Learned

- Demand paging
  - Get more out of limited physical memory
  - Using existing paged memory hardware
- Page replacement algorithms
  - OPT, Approximation for OPT, Approximation for that, etc.
- Thrashing and Locality of Reference