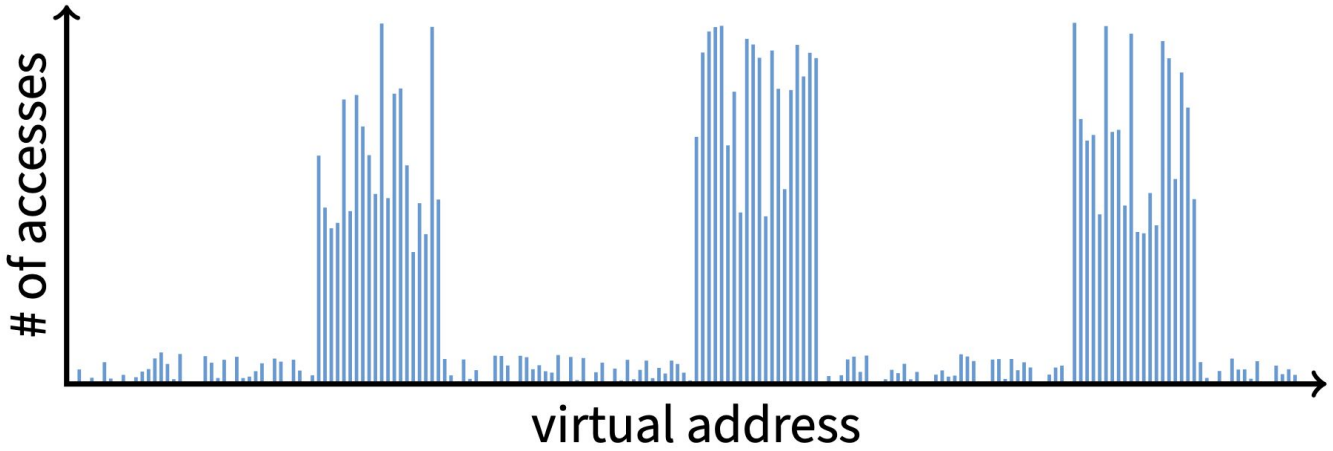
Virtual Memory Beyond Physical Memory

W4118 Operating Systems I

columbia-os.github.io

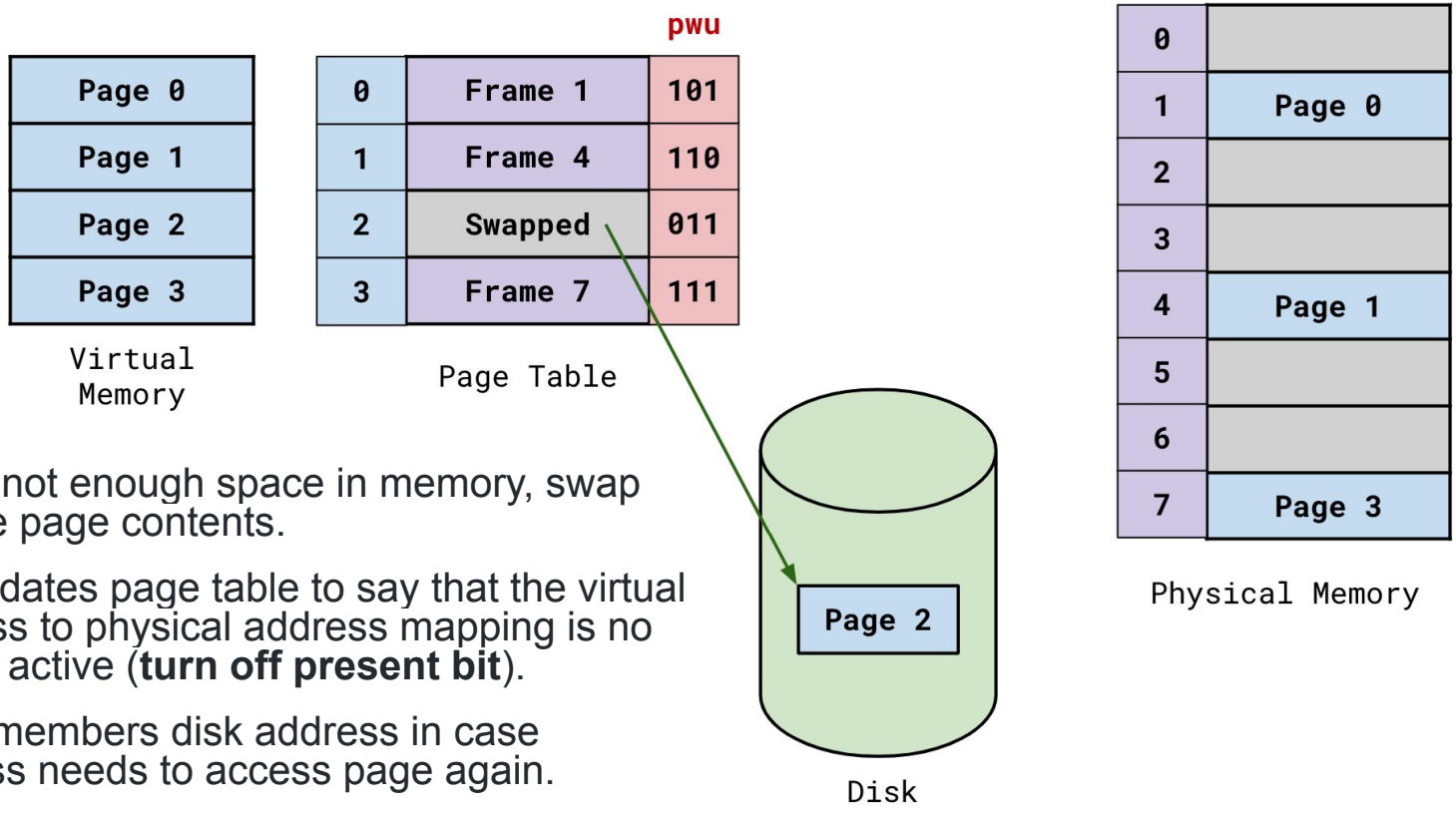
Credits to Jae and David Mazières

Working Set Model



- Disk much slower than memory but not everything might fit in memory
- 80/20 rule: 20% of memory gets 80% of memory accesses
    - Keep the hot 20% in memory, keep the cold 80% in disk

Pages are not always in memory



- When not enough space in memory, swap out the page contents.
- OS updates page table to say that the virtual address to physical address mapping is no longer active (**turn off present bit**).
- OS remembers disk address in case process needs to access page again.

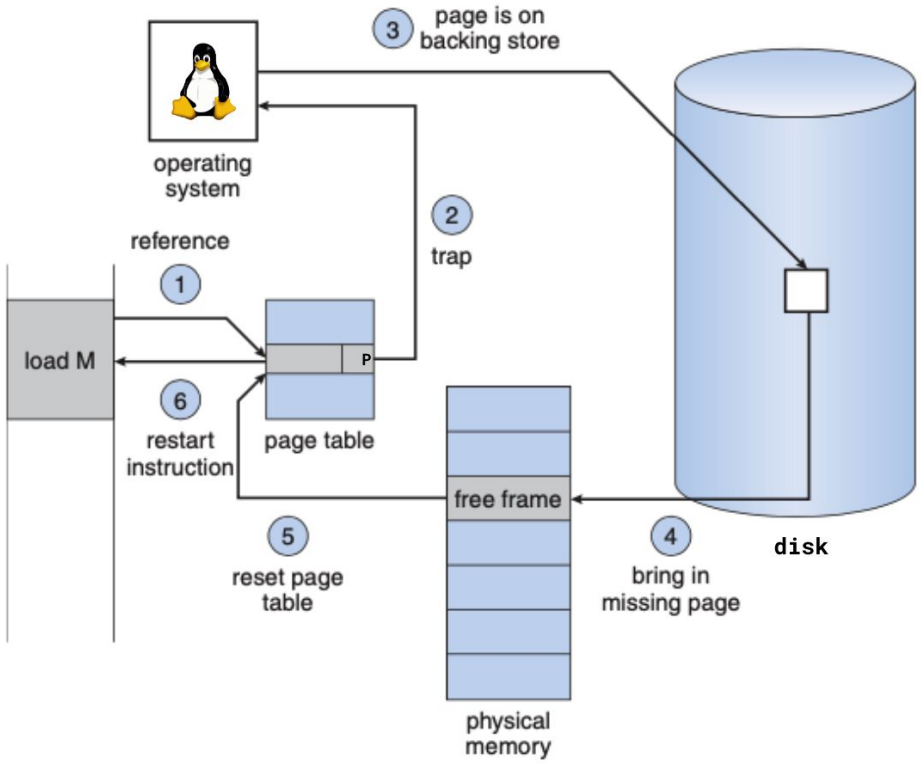
Page Faults

If the page is not present?

⇒ Need the OS to intervene

⇒ raise exception to trap to OS

Page Fault



Page Faults

If the page is not present?

⇒ Need the OS to intervene

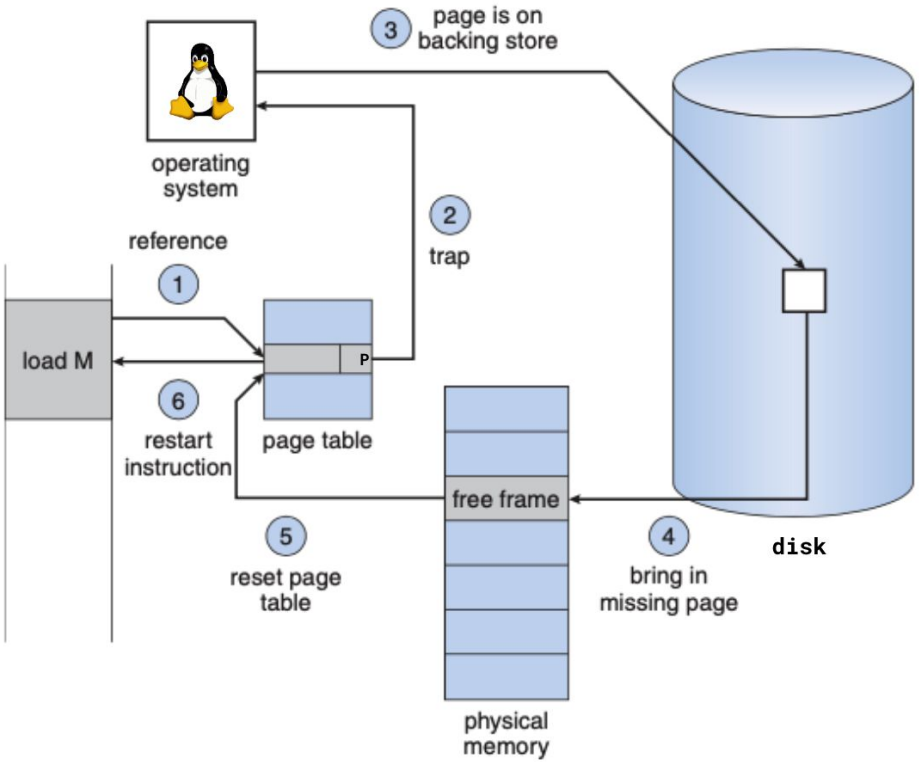
⇒ raise exception to trap to OS

Page Fault

1. `load` instruction by MMU

   check TLB (not shown)

   TLB miss, consult page tables



Page Faults

If the page is not present?

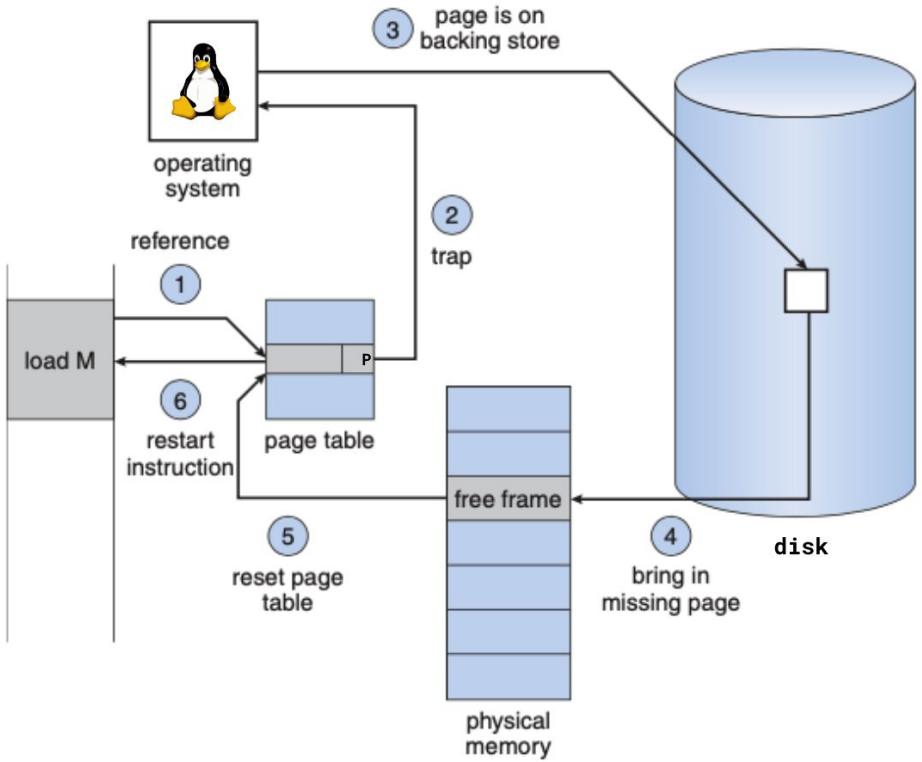
⇒ Need the OS to intervene

⇒ raise exception to trap to OS

Page Fault

2. **present** bit off

   trap to OS via page fault



Page Faults

If the page is not present?

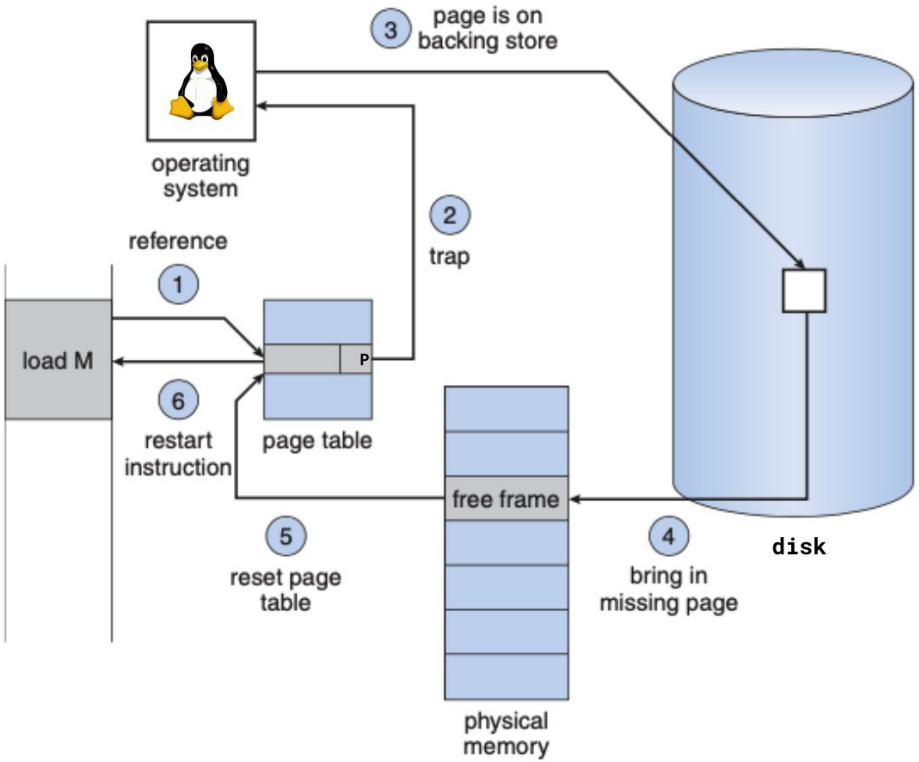
⇒ Need the OS to intervene

⇒ raise exception to trap to OS

Page Fault

3. issue I/O request and put process to sleep

this is why `kmalloc()` could block: might need to fetch referenced pages from disk or swap some pages out to disk to make space



Page Faults

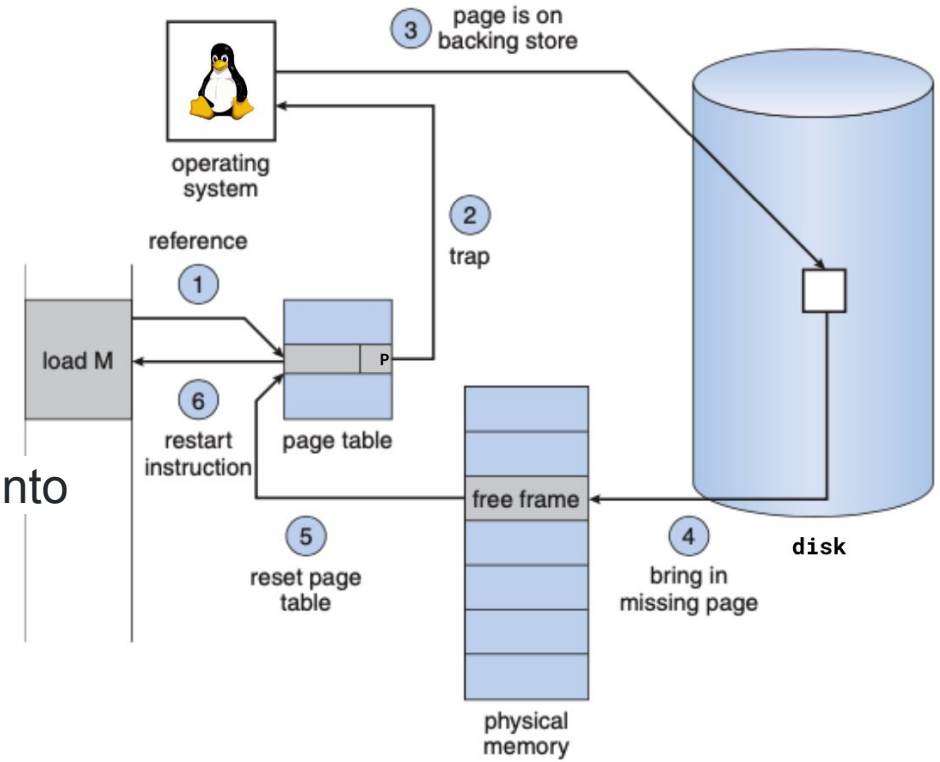
If the page is not present?

⇒ Need the OS to intervene

⇒ raise exception to trap to OS

Page Fault

4. I/O request satisfied, page written into main memory



Page Faults

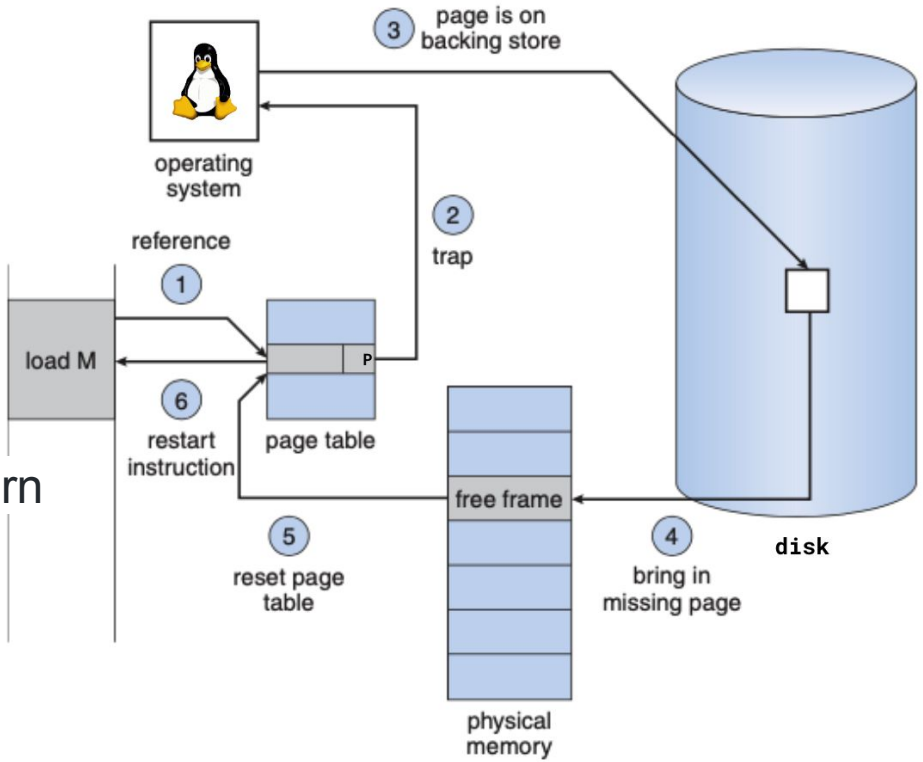
If the page is not present?

⇒ Need the OS to intervene

⇒ raise exception to trap to OS

Page Fault

5. update PTE to point to new PFN, turn on **present** bit



Page Faults

If the page is not present?

⇒ Need the OS to intervene

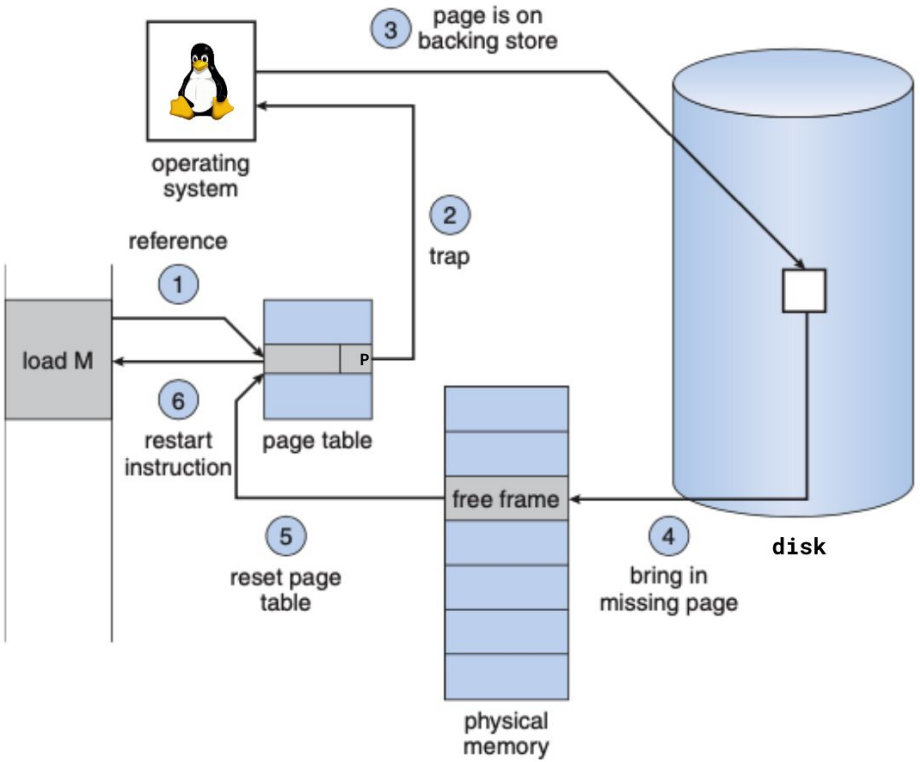
⇒ raise exception to trap to OS

Page Fault

6. restart `load` instruction

   TLB miss but page walk succeeds

   (or can install entry directly to TLB)



Page Faults

If the page is not present?

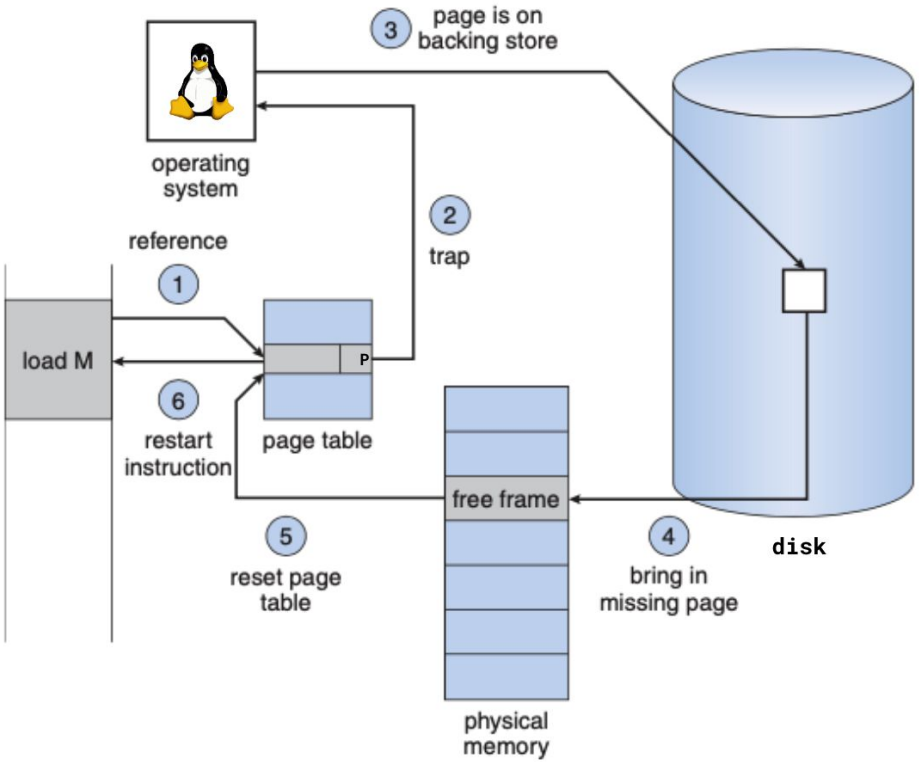
⇒ Need the OS to intervene

⇒ raise exception to trap to OS

Page Fault

The OS page fault handler also handles illegal accesses:

- user mode accessing kernel space
- write access on read-only region
- either send offending process SIGSEGV or handle COW



Restarting Instructions

Hardware must allow resuming after a fault

Hardware provides kernel with information about the page fault

- Faulting virtual address (in `%cr2` reg on x86)
- Address of instruction that caused the fault
- Was the access a read or write? Was it an instruction fetch? Was it caused by user access to kernel-only memory?

Idempotent instructions are easy to restart

- Just re-execute any instruction (e.g., load/store) that only accessed one address

Complex instructions must be restarted too

- E.g., x86 string move, we specify src, dst, count in `%esi`, `%edi`, `%ecx`
- On fault, registers adjusted to resume from where we left off

What to fetch?

Bring in the page the caused the fault

Prefetch surrounding pages?

- Reading two disk blocks is approximately as fast as reading one
- As long as no track/head switch, the seek time dominates
- If the applications exhibits spatial locality, then big win to store and read multiple contiguous pages

Also pre-zero unused pages in zero loop

- Need zero-filled pages for stack, heap, anonymously mmapped memory
- Zeroing them only on demand is slow
- Hence, many OSes zero freed pages while CPU is idle

Which page to swap out to disk?

Need to select a **victim** page to swap out to disk to make room in main memory

Page replacement algorithm should minimize # of page faults.

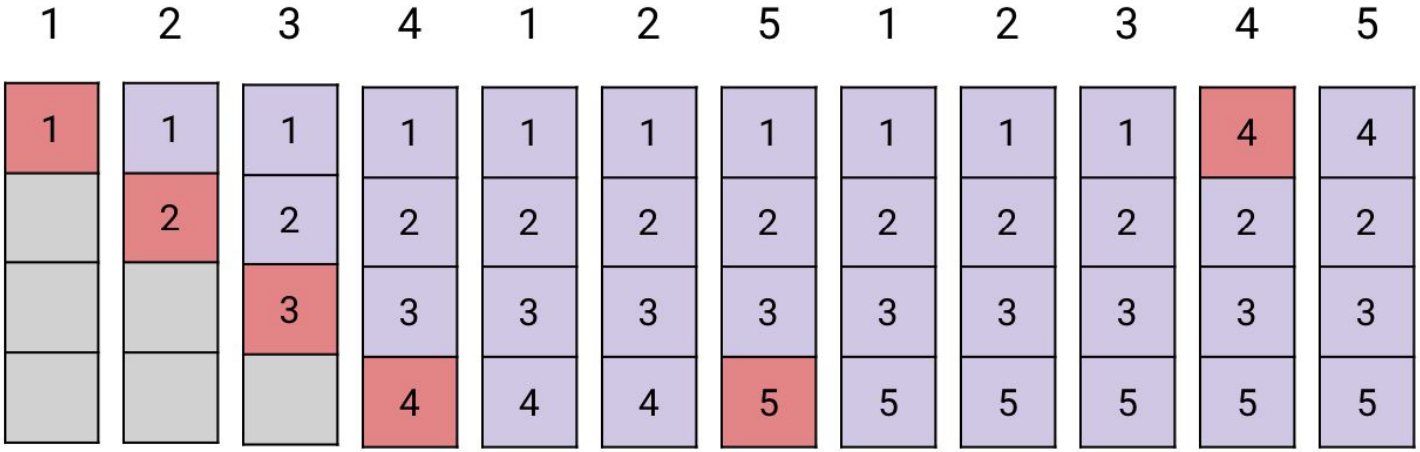
Evaluation: run algorithm on a reference string of page accesses assuming N frame slots in main memory. Compute number of page faults on that string.

Example: `1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5`

```
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
```

Optimal Page Replacement

Swap out the page that won't be used for longest time in the future!



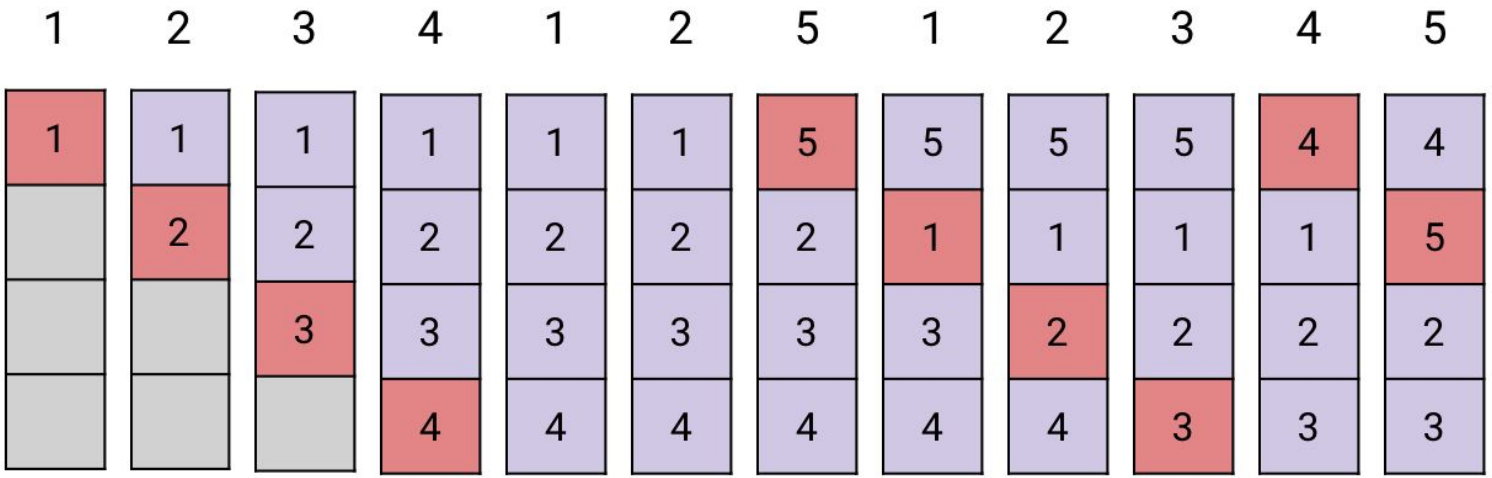
Only 6 page faults – first 4 were **compulsory misses** (had to happen)

Problem: difficult to accurately predict the future

```
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
```

FIFO Page Replacement

Treat pages as a FIFO queue, swap out the page that was loaded first

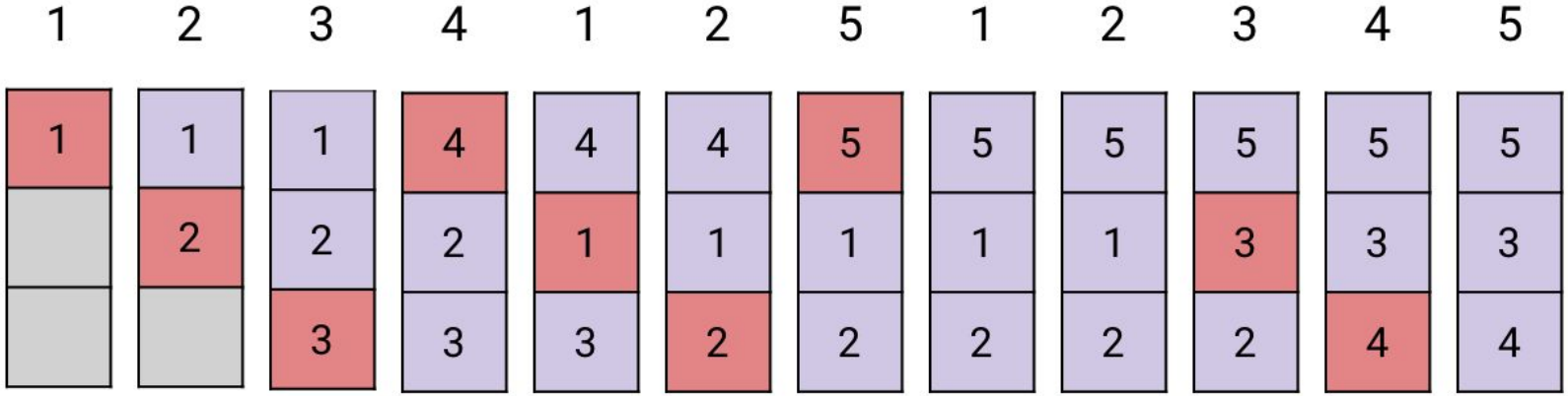


10 page faults – access patterns are ignored

```
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
```

FIFO Page Replacement

What if physical memory was smaller? (3 pages/frames)



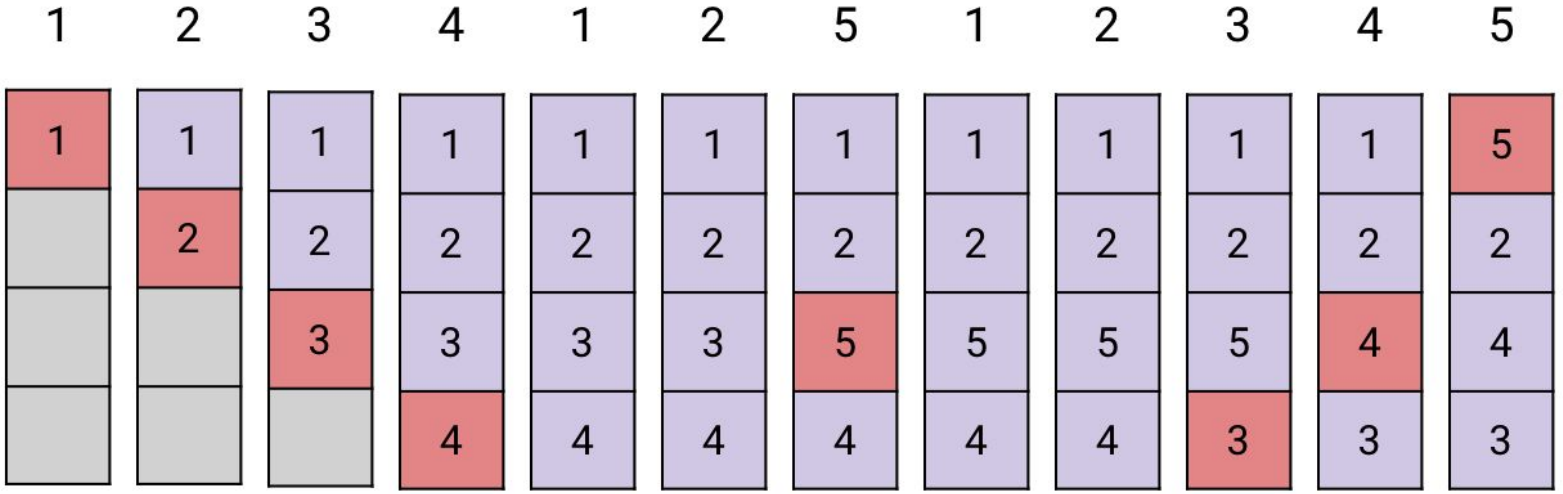
Only 9 page faults! Somehow we have MORE page fault with MORE memory!

Belady's anomaly: FIFO algorithm page fault rate may decrease as memory increases

```
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
```

Least-Recently-Used (LRU) Page Replacement

Swap page that hasn't been used in the longest time.



8 page faults – taking into account locality, LRU approximated OPT

Implementing LRU

Attempt 1: LRU Hardware Implementation

- Associate a **time counter** with each page
- Each time a page is referenced, save system clock into page time counter
- Page replacement: perform a linear scan to find page with oldest time counter

Can we do better than linear search time?

Implementing LRU

Attempt 2: LRU Software Implementation

- Maintain a doubly linked list of pages
- Each time a page is referenced, move to the front of the list
- Page replacement: remove page from the back of the list

Better than linear scan, but requires several pointer updates..

High contention on multiprocessor

Implementing LRU

Attempt 3: LRU Approximation: Clock/Second-chance algorithm

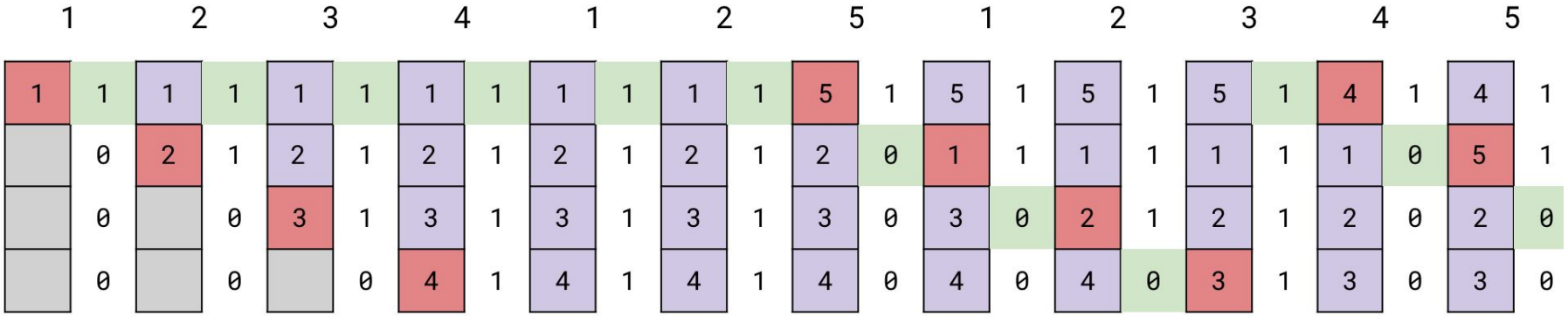
Find a not recently accessed page, but not necessarily the LR accessed

- Maintain a reference bit per page
- On memory reference: hardware sets bit to 1
- Page replacement: OS finds a page with referenced bit clear
- Over time: OS traverses all pages, clearing reference bits

next victim: a cursor over the circular list of pages (the “clock” hand)

If reference bit is 1, clear it and advance hand. Else, return current page as victim

Clock Page Replacement Algorithm



10 page faults

Simple to implement, performs reasonably

Consideration: Dirty Pages

Dirty Page: content has been modified since reading from disk and should be written back to disk to keep file in sync

Clock algorithm doesn’t take into account dirty pages when selecting the victim

- Dirty pages are more expensive to evict, since they need to be written to disk
  - Disk write is ~5ms
- Clean pages don’t need to be written to desk

Clock algorithm with dirty pages

Clock extension: use the dirty bit to prevent dirty pages from being evicted

On page reference:

- read: hardware sets reference bit
- write: hardware sets dirty bit

Page replacement policy:

- reference = 0, dirty = 0 → victim page
- reference = 0, dirty = 1 → SKIP
- reference = 1, dirty = 0 → clear reference bit
- reference = 1, dirty = 1 → clear reference bit
- advance hand, repeat
- if no victim page found, run **swap daemon** to flush unreferenced dirty pages

Advanced clock algorithm

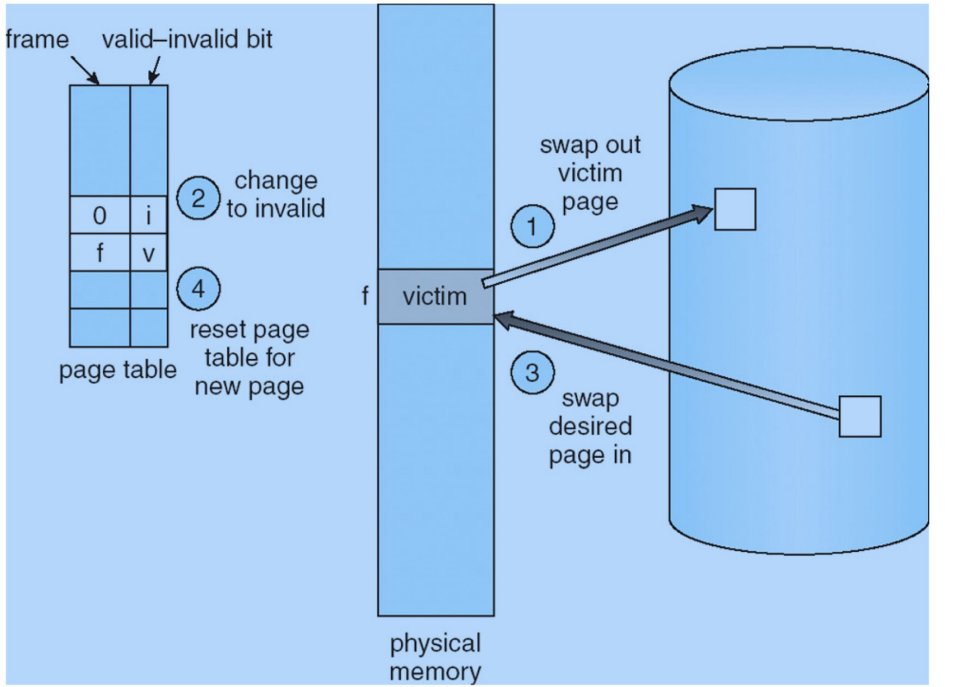
- **Large memory may be a problem**
  - Most pages referenced in a long interval
- **Solution: Add a second clock hand/cursor**
  - Two hands move in lockstep at a fixed distance
  - Leading hand clears reference bits
  - Trailing hand evicts pages with reference=0

Page Replacement Policy Summary

- **Optimal:** throw out page that won't be used for longest time in future
  - Best algorithm if we can predict future
  - Good for comparison, but not practical
- **Random:** throw out a random page
  - Easy to implement
  - Works surprisingly well. Why? Avoid worst case
  - Cons: random
- **FIFO:** throw out page that was loaded first
  - Easy to implement
  - Fair: all pages receive equal residency
  - Ignore access pattern
- **LRU:** throw out page that hasn't been used in longest time
  - Past predicts future
  - With locality: approximates Optimal
  - Simple approximate LRU algorithms exist (Clock)

Cost of paging

- **Naive page replacement: 2 disk I/Os per page fault**



Page Buffering

- **Idea: reduce # of I/Os on the critical path**
- **Keep pool of free page frames**
  - On fault, still select victim page to evict
  - But read fetched page into already free page
  - Can resume execution while writing out victim page
  - Then add victim page to the free pool
- **Can also yank pages back from free pool**
  - Contains only clean pages, but may still have data
  - If page fault on page still in free pool, recycle

Page Allocation

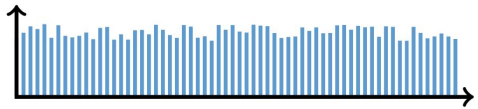
- **Allocation can be *global* or *local***
- **Global allocation doesn't consider memory ownership**
  - E.g., with LRU, evict least recently used page of any proc
  - Doesn't protect you from memory hogs
- **Local allocation isolates processes (or users)**
  - Separately determine how much memory each process should have
  - Then use LRU/clock to determine which pages to evict with a process

Thrashing

- **Processes require more memory than the system has**
  - Each time one page is brought in, another page, who contents will soon be referenced, is thrown out
  - Processes will spend all of their time blocked, waiting for pages to be fetched from disk
  - Disk at 100% utilization, but systems does not get much useful work done
- **What we wanted: virtual memory the size of disk with access time the speed of physical memory**
- **What we got: memory with access time of disk**

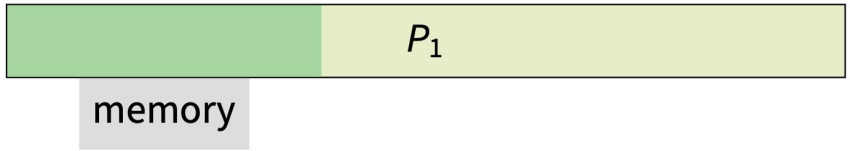
Why thrashing happens

- **Access pattern has no temporal locality**

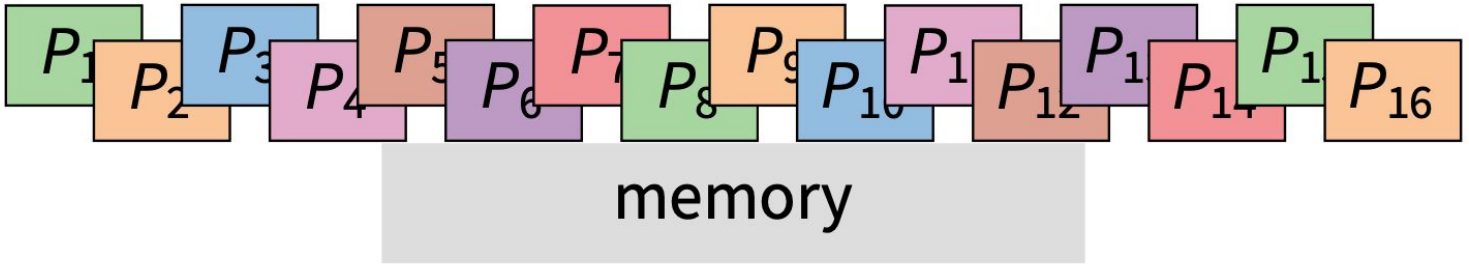


(80/20 rule has broken down)

- **Hot memory does not fit in physical memory**

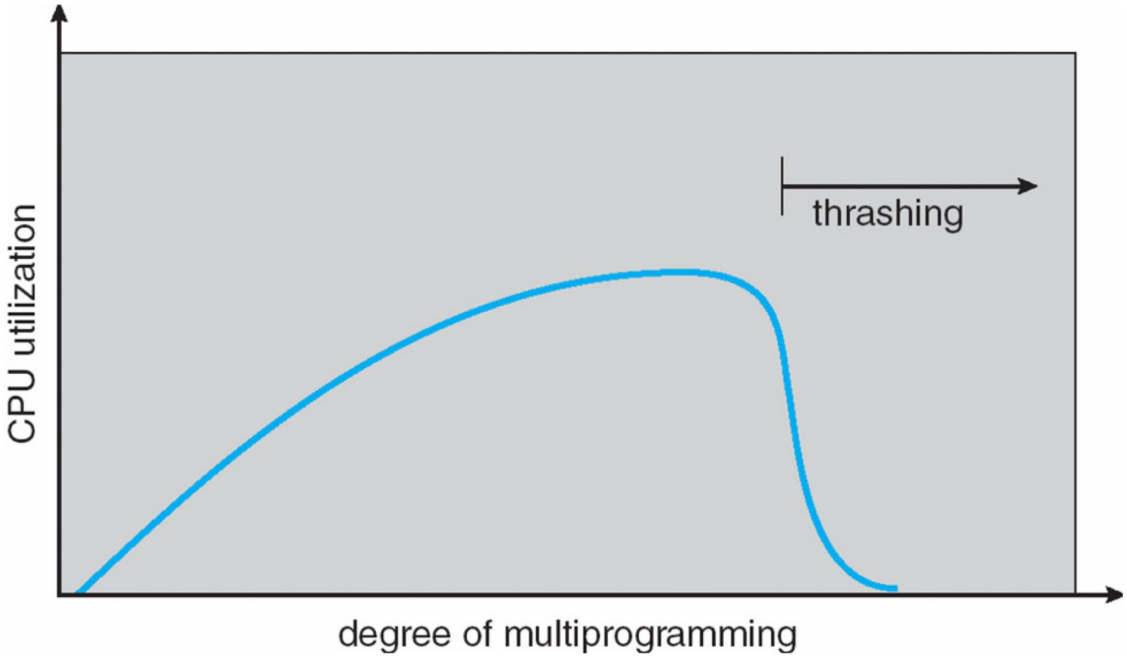


- **Each process fits individually, but too many for the system**



Multiprogramming & Thrashing

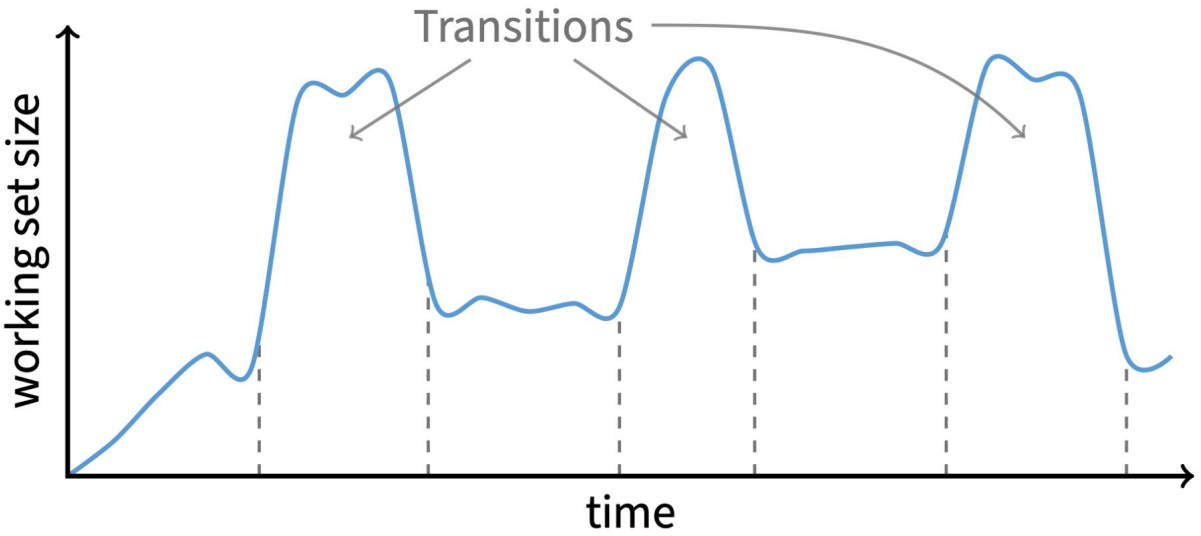
- **Must shed load when thrashing**



Dealing with thrashing

- **Approach 1: working set**
  - Thrashing viewed from a caching perspective: given locality of reference, how big a cache does the process need?
  - Or: how much memory does the process need in order to make reasonable progress?
  - Only run processes whose memory requirements can be satisfied
- **Approach 2: page fault frequency**
  - Thrashing viewed as poor ratio of fetch to work
  - PFF = page fault / instructions executions
  - If PFF rises above threshold, process needs more memory. Not enough memory? Switch the process out.
  - If PFF sinks below threshold, memory can be taken away

Working sets



- **Working set changes across phases and balloons during transitions**

Calculate the working set

- **Working set: all pages that process will access in next T time**
  - Can’t calculate without predicting future
- **Approximate by assuming past predicts future**
  - So working set ~= pages accessed in last T time
- **Keep idle time for each page**
- **Periodically scan all resident pages in system**
  - reference bit set? Clear it and clear the page’s idle time
  - reference bit clear? Add CPU consumed since last scan to idle time
  - Working set is pages with idle time < T

Two-level scheduler

- **Divide processes into active & inactive**
  - Active - means working set is resident in memory
  - Inactive - working set intentionally not loaded
- **Balance set: union of all active working sets**
  - Must keep balance set smaller than physical memory
- **Use two-level scheduling**
  - Move procs active → inactive until the balance set is small enough
  - Periodically allow inactive to become active
  - As working set changes, we must update the balance set
- **Complications**
  - How to choose idle time threshold T
  - How to pick processes for active set
  - How to count shared memory (e.g., shared libraries)