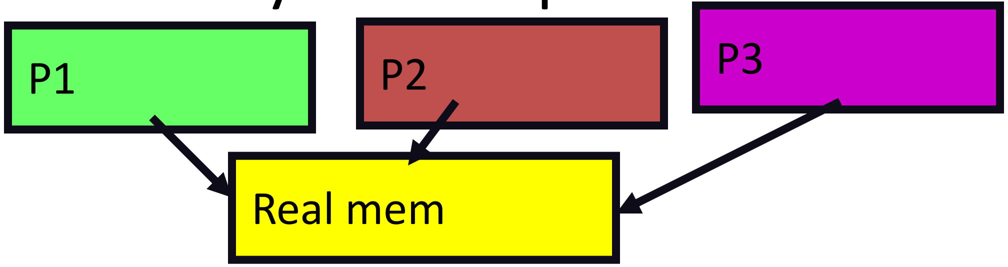
Lecture 13: Thrashing

Thrashing: exposing the lie of VM

- Thrashing: processes on system require more memory than it has.

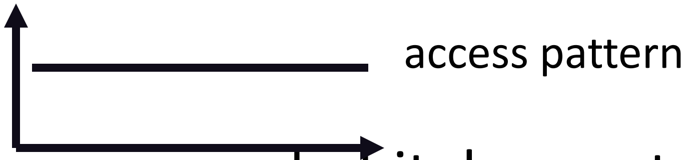


  - Each time one page is brought in, another page, whose contents will soon be referenced, is thrown out.
  - Processes will spend all of their time blocked, waiting for pages to be fetched from disk
  - I/O devs at 100% utilization but system not getting much useful work done

- What we wanted: virtual memory the size of disk with access time of of physical memory
- What we have: memory with access time = disk access

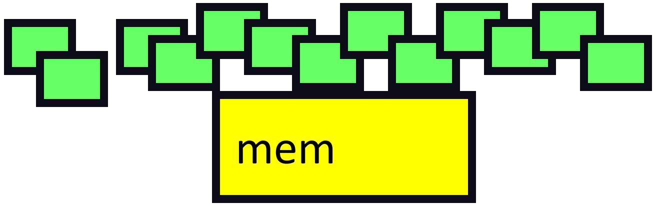
Thrashing

- Process(es) “frequently”reference page not in mem
  - Spend more time waiting for I/O then getting work done
- Three different reasons
  - process doesn’t reuse memory, so caching doesn’t work (past != future)

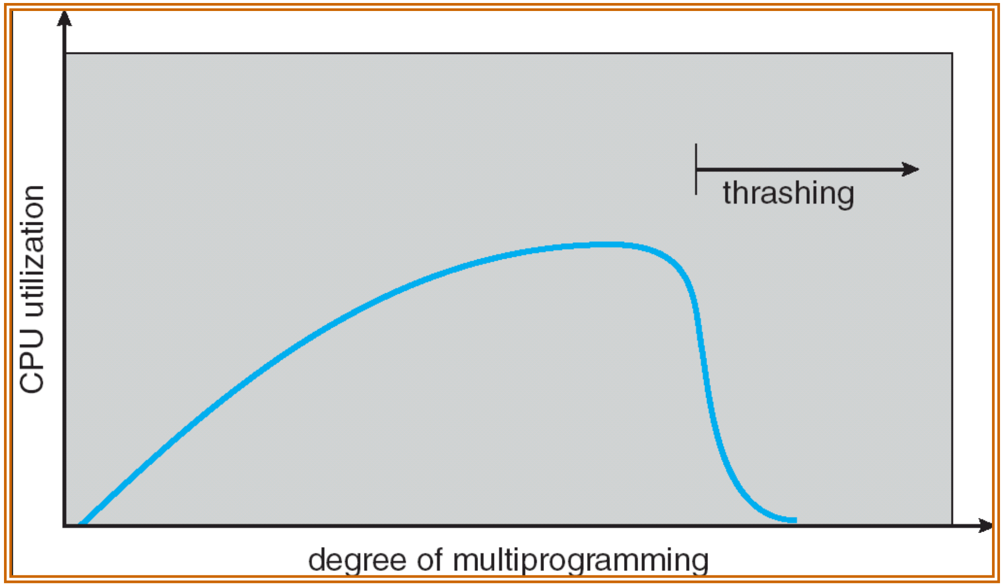


  - process does reuse memory, but it does not “fit”

  - individually, all processes fit and reuse memory, but too many for system.



Thrashing



- If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system spends most of its time swapping to disk
- Thrashing ≡ a process is busy swapping pages in and out
- Questions:
  - How do we detect Thrashing?
  - What is best response to Thrashing?

When does thrashing happen?

- (Over-)simple calculation of average access time:

Let h = percentage of references to pages in memory
Then average access time is
  h * (cost of memory access)
    + (1-h) * (cost of disk access + miss overhead)
For current technology, this becomes (about)
 h * (100 nanoseconds) + (1-h) * (10 milliseconds)
Assume 1 out of 100 references misses.
  = .99 * (100ns) + .01 (10ms)
  = .99 (100ns) + .01 (10,000,000ns)
  = 99 + 100,000 ~ 100 microseconds

  – or, 1000x slower than main memory.

- Even small miss rates lead to unacceptable average access times. What can OS do???

Making the best of a bad situation

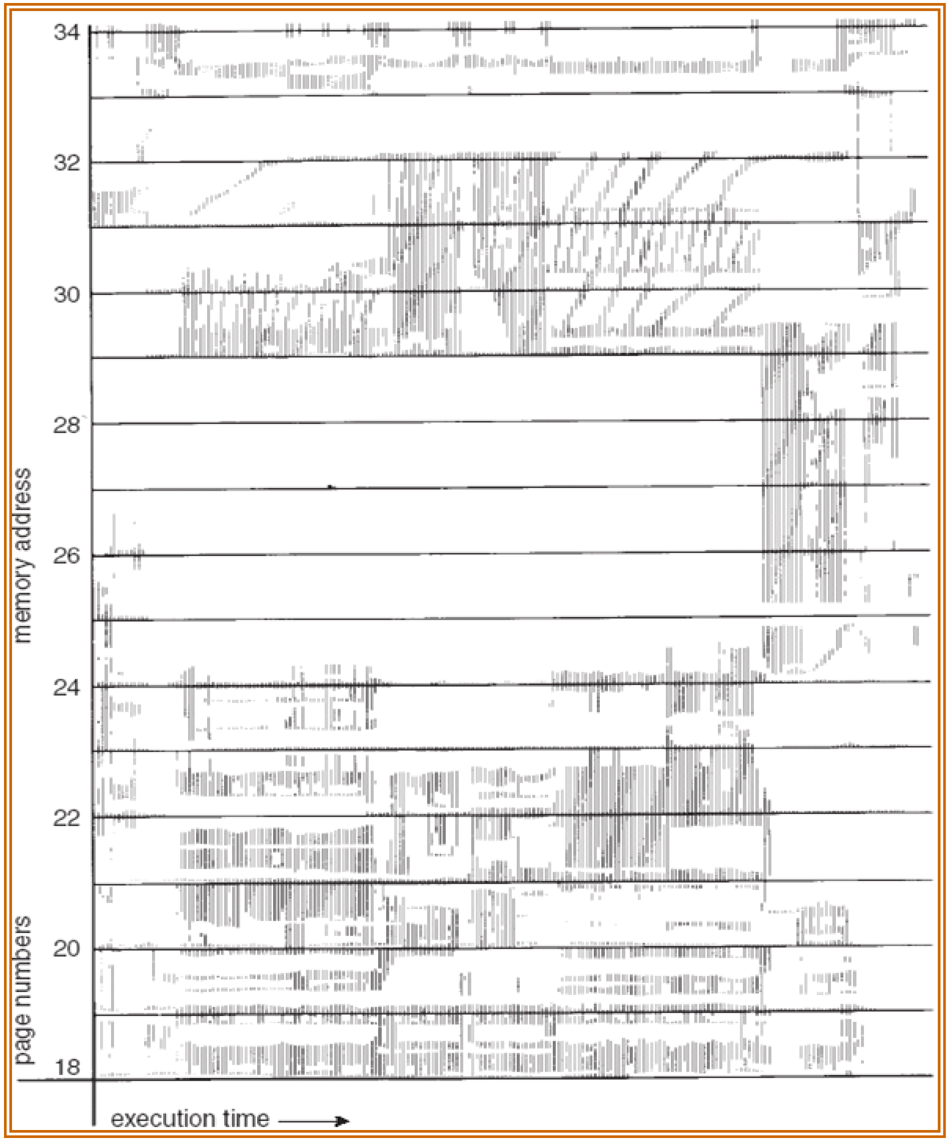
- Single process thrashing?
  - If process does not fit or does not reuse memory, OS can do nothing except contain damage. (cs140?).
- System thrashing?
  - If thrashing arises because of the sum of several processes then adapt:
    - figure out how much memory each process needs
    - change scheduling priorities to run processes in groups whose memory needs can be satisfied (load shedding)
    - if new processes try to start, can refuse (admission control)
- Careful: example of technical vs social.
  - OS not only way to solve this problem (and others).
  - “Social” solution: buy more memory.
  - Another: use ‘ps’ to find idiot killing machine and yell

Methodology for solving?

- 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 process need in order to make "reasonable" progress (its working set)?
  - 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 faults / instructions executed
  - if PFF rises above threshold, process needs more memory
    - not enough memory on the system? Swap out.
  - if PFF sinks below threshold, memory can be taken away

Locality In A Memory-Reference Pattern

- Program Memory Access Patterns have temporal and spatial locality
  - Group of Pages accessed along a given time slice called the “Working Set”
  - Working Set defines minimum number of pages needed for process to behave well
- Not enough memory for Working Set⇒Thrashing
  - Better to swap out process?



Working set (1968, Denning)

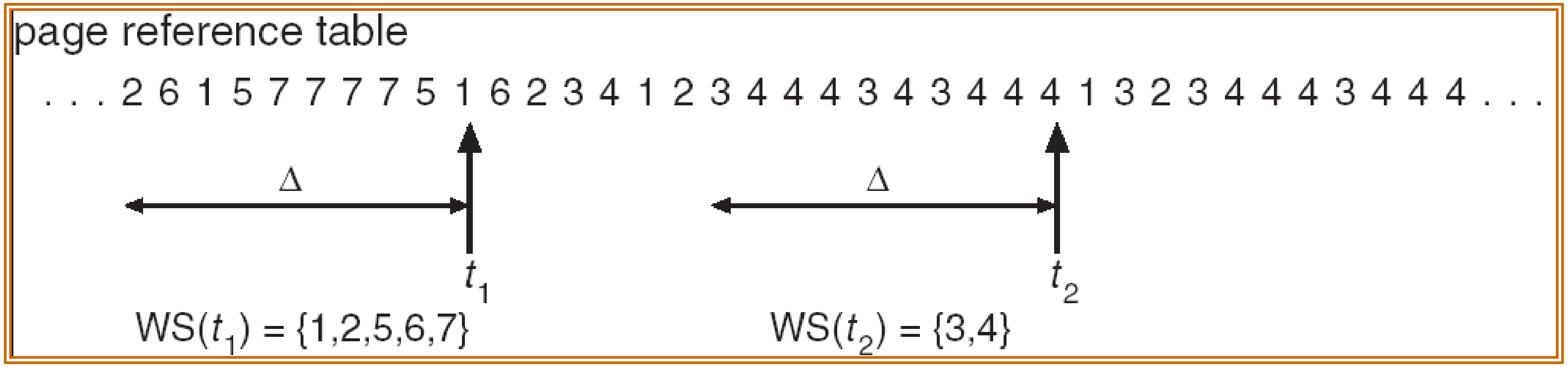
- What we want to know: collection of pages process must have in order to avoid thrashing
  - This requires knowing the future. And our trick is?
- Working set:
  - pages referenced by process in last T seconds of execution considered to comprise its working set
  - T: the working set parameter

- Uses?
  - Cache partitioning: give each app enough space for WS
  - Page replacement: preferentially discard non-WS pages
  - Scheduling: process not executed unless WS in memory

Scheduling details: The balance set

- Sum of working sets of all runnable processes fits in memory? Scheduling same as before.
- If they do not fit, then refuse to run some: divide into two groups
  - active: working set loaded
  - inactive: working set intentionally not loaded
  - balance set: sum of working sets of all active processes
- Long term scheduler:
  - Keep moving processes from active -> inactive until balance set less than memory size.
  - Must allow inactive to become active. (if changes too frequently?)
- As working set changes, must update balance set…

Working-Set Model



- $\Delta \equiv$ working-set window $\equiv$ fixed number of page references
  - Example: 10,000 instructions
- WS_i (working set of Process P_i) = total set of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \Sigma|WS_i| \equiv$ total demand frames
- if $D > m \Rightarrow$ Thrashing
  - Policy: if $D >$ m, then suspend/swap out processes
  - This can improve overall system behavior by a lot!

What about Compulsory Misses?

- Recall that compulsory misses are misses that occur the first time that a page is seen
  - Pages that are touched for the first time
  - Pages that are touched after process is swapped out/swapped back in
- Clustering:
  - On a page-fault, bring in multiple pages “around” the faulting page
  - Since efficiency of disk reads increases with sequential reads, makes sense to read several sequential pages
- Working Set Tracking:
  - Use algorithm to try to track working set of application
  - When swapping process back in, swap in working set

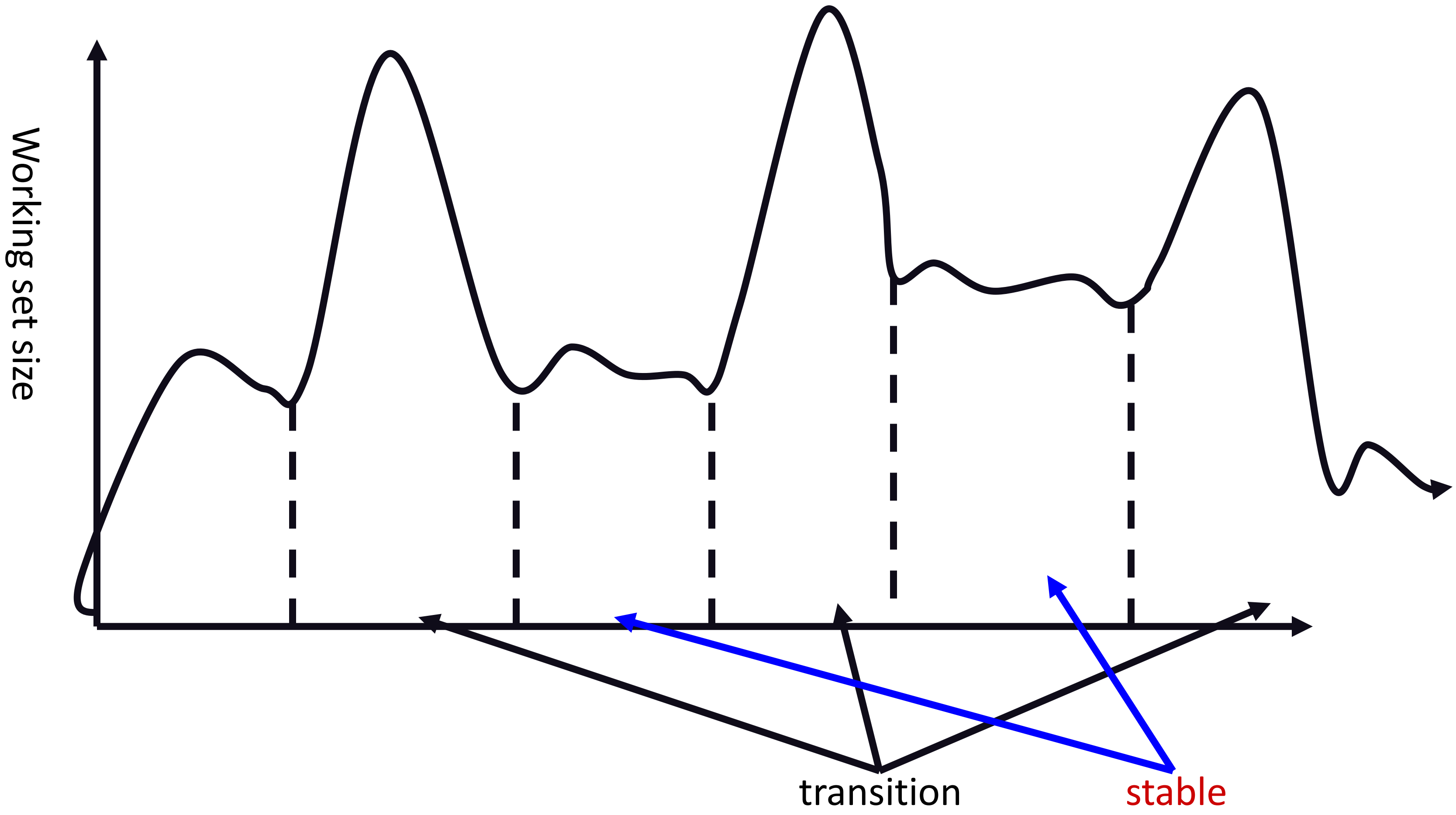
How to implement working set?

- Associate an idle time with each page frame
  - idle time = amount of CPU time received by process since last access to page
    - (why not amount of time since last reference to page?)
  - page's idle time > T? page not part of working set
- How to calculate?
  - Scan all resident pages of a process
    - use bit on? clear page's idle time, clear use bit
    - use bit off? add process CPU time (since last scan) to idle time
  - Unix:
    - scan happens every few seconds
    - T on order of a minute or more

Some problems

- T is magic
  - what if T too small? Too large?
  - How did we pick it? Usually “try and see”
  - Fortunately, systems aren’t too sensitive
- What processes should be in the balance set?
  - Large ones so that they exit faster?
  - Small ones since more can run at once?
- How do we compute working set for shared pages?

Working sets of real programs



- Typical programs have phases:
  - working set of one may have little to do with other
  - balloons during transitions….

Working set less important

- The concept is a good perspective on system behavior.
  - As optimization trick, it’s less important: Early systems thrashed lots, current systems not so much.

- Have OS designers gotten smarter? No. It’s the hardware guys (cf. Moore’s law):
  - Obvious: Memory much larger (more to go around)
  - Less obvious: CPU faster so jobs exit quicker, return memory to freelist faster.
  - Some app can eat as much as you give. The percentage of them that have “enough” seems to be increasing.

  - Social implication: while speed very important OS research topic in 80-90s, less so now (should it be more important again?)