Working Sets, Page replacement

1. Mini projects
2. Group presentation
3. Big Picture
   a. A lot of what an OS does is allocate & manage resources
   b. Key problems are:
      i. deciding how much resource a process needs
         1. Does it know?
         2. Does its programmer know?
      ii. Under resource constraints, deciding how to allocate resources
         1. Give everybody a little less than they need
         2. Admission control / load control: don’t even try to satisfy everyone at once
   a. Big idea:
      i. computers operate in two regimes:
         1. undercommitted: have enough memory, don’t have to worry about page replacement (becoming more common)
            a. Generally true on servers today
            b. why? Server programs have flexible memory usage (e.g. as a cache), so can use enough memory not to page
         2. Overcommitted – too many programs want memory
      ii. Overcommitment leads to thrashing and poor performance
         1. No program has enough memory to run
         2. Time is spent doing I/O but CPU is not busy
         3. Memory is not well utilized – rarely referenced because programs are busy doing I/O
      iii. Better solution: admission control
         1. Figure out how much memory processes need
         2. Run only enough processes that their needs fit in memory simultaneously
         3. Swap whole processes in and out
   b. Intro
      i. Context: automatic resource and scheduling a pretty new thing, as was multiprogramming. Previously, applications handled it themselves (e.g. explicitly swap memory, reserve fixed amount).
      ii. Key problem: efficiency, simple programming
      iii. Efficiency: want full utilization of CPU -- it was the critical resource
      iv. Simple programming: not have to deal with manual resource allocation.
      v. Prior work: automatically give pages to processes, use simple LRU or FIFO global page replacement.
c. Problems
   i. **SHOW CLOCK**
      ii. CLOCK problems: need to know how fast clock hands move to make memory available. If too fast for some programs, the memory it needs won't be there. – scan twice before re-run-- If too slow, not enough free memory when needed.
   iii. Problem: thrashing. Every program takes a page fault, swaps waiting for a page, meanwhile somebody takes another page from it, and when it comes back, it faults again.
   iv. **Programs may block just after being rescheduled** if data not in memory – not make progress
      1. **Displace** useful pages in memory to bring something in, but then immediately block waiting for other pages

d. **OVERALL GOAL/APPROACH:**
   i. Estimate memory & CPU needed by a program
   ii. Make sure you have that much available
   iii. Have models to back up the idea
      1. **QUESTION: value of a model?**
         a. Can figure out what the problem is fundamentally
            i. Can never run well with not enough memory
         b. Can figure out just about how much memory you need
            i. What does it mean to have “enough memory”?
   e. Other people: perhaps VM not worth it? Ask for advice? e.g. let programmer tell system how much memory it needs, when to bring in pages?
      i. Problem:
         1. Usage may depend on environment, hard for programmer
         2. Modular compilation makes it hard for compilers.
         3. **Prefetching has some use, but does not work well for irregular codes**
            a. Increases memory pressure by bringing in useless pages, uses disk bandwidth
            b.
      ii. Observation: not worth scheduling a process that doesn’t have enough memory to make progress. Need to make scheduler + memory manager work together.
   f. Working Set Approach: build a model of program behavior, use model + measurements to make scheduling + allocation decisions.
      i. **APPROACH (KEY):** instead of figuring out what page to discard, figure out what pages should be there, and make sure they are.
      ii. **Definition: working set** is the smallest collection of information that must be present in main memory to assure “efficient” execution of the program (i.e., not too much blocking)
   g. Working set MODEL:
      i. 2 level memory
ii. Traverse time T = time to transfer a page between memory

iii. Goal of mem mgmt:
   1. Decide which pages are in core -- decide which pages to REMOVE (not load)
   2. **Question: what to optimize to minize page traffic** = pages/unit time moved between memories.
   3. page in on demand only

iv. Prior work: LRU, Random, LFU, FIO, ... Problem: susceptible to overload under heavy competition. EXPLAIN WHY

h. Working set = minimal set of pages that must be in main memory for a process to operate efficiently without unnecessary page faults

i. Look at pages as the program executes (process time, not real time)

ii. For some window size tau ending at time t, some set of references (stack, heap, code, memory mapped files, kernel code and data) = W(t,tau), w(t,tau) = |W|

iii. What happens as you change the windows size tau?

iv. Model drives a notion that over some time scale tau, you can capture the locality of a program: what is referenced?

5. Properties of working set model
   a. Size: monotonically increasing, converging: w(t,2tau) <= 2w(t,tau)
      i. (show plot)
      
      ![Graph showing the relationship between w(t, t) and t]

      ii. Locality is both size of w-set and sensitivity - how much does it change as tau is changed. This lets you capture memory, cpu demand

      1. **QUESTION: How does graph change for different programs?**
         a. Column-major array traversal?
         b. Row major array traversal?
         c. Hash table?

      2. **QUESTION: Does changing memory size change the working set?**

      3. **QUESTION: Does changing the processor speed change the working set?**

      4. **QUESTION: does changing the page size change the working set?**
         a. Increase in # of pages, decrease in # of bytes

b. Prediction: immediate past behavior predicts the future:
   i. W(t,tau) is a good predictor for W(T+a,tau) for small chang a. For large
ii. QUESTION: but earlier say prefetching/look ahead not that useful?
   1. A: this says working set changes slowly, not that you know what is missing

c. a, it is not good.
   i. Probability that W(t+a,a) intersect W(t,tau) = null is small
      1. Observation: Lots of reuse

d. Reentry rate:
   i. As tau is reduces, w(t,tau) (size) reduces; probability that useful pages are not in W(t,tau) increases.
   ii. Lambda = process entry rate (only while process is running)
      1. mean process time between instants at which a given page reenters W(t,tau) = 1/lambda(tau)
      2. Lambda is frequency of page faults, in process time how often do you reference some new you didn’t reference in previous tau period?
      3. 1/lambda is mean time between faults
   iii. phi = real-time entry rate, mean real-time between reentry of a page = 1/phi(tau)]
      1. No process time—incluedes waiting time
   iv. A reentry point is a reference instant which finds the page not in W(t,tau)
      at such an instant, it reenters W.
      1. Times are reentry points if they reference a page not in W
      2. Let Xn = mean inter-reference interval of a page; higher means less frequent
      3. {tn} = sequence of instances when references to page occur. inter-reference interval xn = tn – tn-1.
   v. Idea: lambda(tau) = 1/m(tau), m == mean time between reenties
      1. lambda(tau) = (1 – Fx(tau))/ mean inter-arrival time, Fx(tau) = probability that interval between request < tau
      2. 1 – Pr[interval < tau]/mean interval
   vi. So: in interval A, single page reentiers is A*lambda(tau)

e. But, each reentry causes wait for time T to bring back page
   i. so: real time to retrieve execute that much = A + Alambda(tau) T
      1. (execution time A + page fault time Alambda(tau) * handling time T) assuming no unnecessary stalls

f. Return traffic rate phi(t) = A lambda(tau) / (A + A lambda(tau) T)
   i. = lambda(tau)/(1+lambda(tau)T)
   ii. = fault rate / total elapsed time

g. DEFINE: memory balance = sum(w(t,tau)) for all processes = betaM == some fraction beta of main memory size
   i. Total return traffic rate == total reentry rate i nreal-time to main memory when no unnecessary stalling
      1. PHI(tau) == betaMphi(tau) = betaM * lambda(tau)/(1+lambda(tau)T)
h. Key point: total traffic limited by bus/disk bandwidth.
   i. So, choice of Tau, beta informed by physical limitations
   ii. Cannot have higher rate of page faults than bus/disk can handle, or else system thrashes

6. Sensitivity
   a. sigma(tau) = how sensitive reentry rate is to tau.
      i. tau-sensitivity = -(d/dtau)lambda(tau)
      ii. Meaning: if tau is decreased by delta-tau, lambda(tau) increases by sigma(tau).delta-tau

b. NOTE: Choice of tau:
   i. too small means pages may be removed while still useful
   ii. too large, pages in memory too long, wasting memory (impacted by how many simultaneous working sets you need in memory)

c. Recommendation: tau comparable to traverse time T
   i. Example: inter-reference rate x:
      1. x < tau : page in memory 100% of time, always in W(t,tau)
      2. tau < x < tau + T : page in memory for tau/(tau+2T) fraction of time: in for tau, then paged out, during which it is referenced, so begins return trip (e.g. 2T for out to disk and back), page reappears tau+2T after previous reference
         a. Note: assumes you reclaim memory the instance you swap out, of not degenerates into next case
         b. Residency fixed independent of X – always wait for time to swap out + swap in
      3. x > t+tau. Page in memory for tau/(x+T): in memory for tau, swapped out, then referenced (at x) then brough back in (time T).
         Note: not wait to be swapped out, as reference happens when already out
   ii. Residency function: interval time vs. residency fraction:
      1. residency starts at 100, step drop to case 2, curve down to zero at T+tau

d. Residency function: interval time vs. residency fraction:
To keep cliff at tau smaller, set tau = 2T

**WHY? Just a nice guidelines**

**Implication: working set = # of pages you reference in 2x page fault time**

f. How do we detect W (the working set of pages)?
   i. \( W(t, \tau) = \text{pages process referenced in last } \tau \text{ interval} \)
      1. Or: reference bits on page, shift at intervals adding up to tau.
      2. Count = # refs since last interval
      3. OR: clock, etc.

g. Resource allocation with working set:
   i. DEMAND = intrinsic need of program (independent of system)
   ii. Memory demand of process \( i \): \( m_i = \min(w_i/M, 1) \) = fraction of main memory
       1. depends on tau for \( w_i \)
   iii. Processor demand = amount of quanta used before blocking.
       1. Only look at things outside process – external interactions
       2. \( = Q(t_i)/NA \) if was just given \( t_i \) seconds, how much more would it use before blocking / # of processors * standard time interval 
          (\( NA \) = total capacity)
       3. NOTE: demand per-cpu; ignores blocking time, because it always blocks for I/O or other processes.
          a. Is natural burst length of processes – e.g., how long take to respond to a keystroke in an editor, request in a web server
       4. Depends on quantum size (as it is a fraction of a standard quantum)
   iv. Balance = total mem demand < 1, total CPU demand < 1

h. SCHEDULING POLICY TO ACHIEVE BALANCE:
   i. block jobs on the ready list: pick the one that brings CPU, memory closest back to balance (e.g. amount used - beta, ...)

![Residency Diagram](image-url)
ii. avoid thrashing by balancing memory first, then balancing processor (avoid memory overcommit)

iii. When looking for a process to remove, pick one that creates the biggest amount of resources:
   1. Minimize usage $S - (\alpha, \beta)$ for the process being removed

i. OVERALL BIG PICTURE:
   i. track pages in use by a process. Throw out anything not referenced in last window
   ii. make sure every process gets its CPU, memory demand
   iii. swap out jobs to bring machine back to utilization goal
   iv. QUESTIONS:
       1. what happens during load, when no pages in memory? A: need to identify load phase, do something different

7. Implementing WS
   a. Need to keep track of last reference time of each page
   b. Replace pages where last ref time > tau (or oldest page)
   c. Only consider pages from same process
   d. QUESTION: How do for real?
       i. Scan/clear reference bit periodically
       ii. last-ref time set to current time if referenced
       iii. to get working set: scan all pages, take pages within Tau time of current virtual time

8. Comparison to Clock:
   a. Describe global clock
   b. QUESTION: What does it evict?
       i. Global not-recently used
       ii. Programs tend to start after I/O or other programs run without pages in memory, will immediately fault
       iii. leads to thrashing
           1. interactive tasks penalized because they run less often, memory is often removed, leading to poor response time
           iv. Poor isolation: programs all compete for global pool of memory
   c. Possible solutions?
       i. Explicitly select subset of tasks to run, guarantee enough memory

9. Relevance:
   a. Used in Windows – use page fault rate to estimate working set for programs to directly allocate # of pages to programs (not use clock)
   b. Used for scheduling – try to swap out programs that don’t have many pages to make more pages available since not worth running them