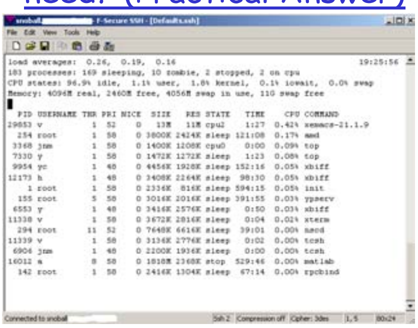
14: Memory Management

Last Modified:
10/31/2002 12:30:51 AM

Limited DRAM

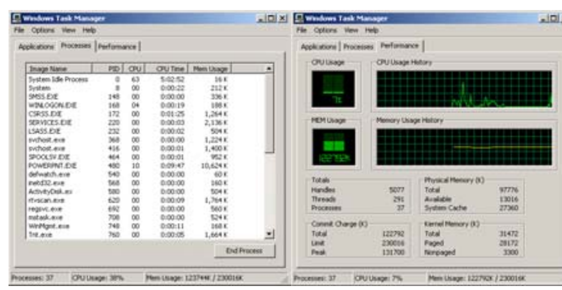
- With paging we could probably "function" with just one resident memory page for each process (and its Master Page Table)
- But reading and writing memory pages to disk is expensive so we don't want to do it very often
- So how much system DRAM do we really need for each process?
  - Do we give each process the same amount of memory?
  - Do they all need the same amount?
  - Do we have enough system DRAM to support all the processes we want to run?(We know we can do better than 4 GB for each one but to avoid constant paging how many do we need)
- Two ways to answer – practical and theoretical

How much memory do processes need? (Practical Answer)



- top
- SIZE vs RES
  - Absolute?
  - Relative?
- Total real memory
  - Free
  - Swap in use

Windows Task Manager



Observations About Actual Memory Usage

- Varies significantly per process
- Are any processes paging "too heavily"?
  - Could we tell just from these stats? How would we know?

How much memory do processes need? (Theoretical Answer)

- "Working set" of a process is the set of virtual memory pages being actively used by the process.
- Define a working set over an interval
  - $WS_P(w)$= {pages P accessed in the last w accesses}
  - If w = total number of P accesses P makes then $WS_P(w)$= every virtual memory page touched by P
- Small working set = accesses of a process have high degree of locality

Changes in Working Set

- Working set changes over the life of the process
  - Ex. At first all the initialization code is in the working set of a process but after some time if won't be any longer
- Intuitively, you need to keep the working set of a process in memory or the OS will constantly be bring pages on and off of disk
- Normally when we ask how much memory a given program needs to run, the answer is either its average or maximum working set (depending on how conservative you want to make your estimate)

Demand Paging

- When a process first starts up
  - It has brand new page table with all PTE valid bits set to false because no pages yet mapped to physical memory
  - As process fetches instructions and accesses data, there will be "page faults" for each page touched
  - Only pages that are needed or "demanded" by the process will be brought in from disk
- Eventually may bring so many pages in that must choose some for eviction
  - Once evicted, if access, will once again demand page in from disk

Demand Paging

- When working set changes (like at the beginning of a process), you will get disk I/O – really no way around that!
- BUT if most memory accesses result in disk I/O the process will run *painfully* slow
- Virtual memory may be invisible from a functional standpoint but certainly not from a performance one
  - There is a performance cliff and if you step off of it you are going to know
  - Remember building systems with cliffs is not good

Prepaging?

- Anticipate fault before it happens and prefetch the data
- Overlap fetch with computation
- Can be hard to predict and if predict wrong evict something useful in exchange
- Programmers can give hints
  - vm_advise

Thrashing

- Thrashing – spending all your time moving pages to and from disk and little time actually making progress
- System is overcommitted
- People get like this ☺

Avoiding Paging

- Given the cost of paging, we want to make it as infrequent as we can
- Function of:
  - Degree of locality in the application (size of the working set over time)
  - Amount of physical memory
  - Page replacement policy
- The OS can only control the replacement policy

Goals of Replacement Policy

- Performance
  - Best to evict a page that will never be accessed again if possible
  - If not possible, evict page that won't be used for the longest time
  - How can we best predict this?
- Fairness
  - When OS divides up the available memory among processes, what is a fair way to do that?
    - Same amount to everyone? Well some processes may not need that amount for their working set while others are paging to disk constantly with that amount of memory
    - Give each process its working set?
  - As long as enough memory for each process to have its working set resident then everyone is happy
    - If not how do we resolve the conflict?

Page replacement algorithms

- Remember all the different CPU scheduling algorithms the OS could use to choose the next job to run
- Similarly, there are many different algorithms for picking which page to kick out when you have to bring in a new page and there is no free DRAM left
- Goal?
  - Reduce the overall system page fault rate?
  - Balance page fault rates among processes?
  - Minimize page faults for high priority jobs?

Belady's Algorithm

- Evict the page that won't be used again for the longest time
- Much like ShortestJobFirst!
- Has provably optimal lowest page fault rate
- Difficult to predict which page won't be used for a while
  - Even if not practical can use it to compare other algorithms too

First-In-First-Out (FIFO)

- Evict the page that was inserted the longest time ago
  - When page in put on tail of list
  - Evict head of list
- Is is always (usually) the case that the thing accessed the longest time ago will not be accessed for a long time?
- What about things accessed all the time!
- FIFO suffers an interesting anomaly (Belady's Anomaly)
  - It is possible to increase the page fault rate by increasing the amount of available memory

Least-Recently Used (LRU)

- Idea: the past is a good predictor of the future
  - Page that we haven't used for the longest time likely not to be used again for longest time
  - Is past a good predictor
    - Generally yes
    - Can be exactly the wrong thing! Consider streaming access
- To do this requires keeping a history of past accesses
  - To be exact LRU would need to save a timestamp on each access (I.e. write the PTE on each access!)
  - Too expensive!

Approximating LRU

- Remember the reference bit in the PTE
  - Set if read or written
- At some regular interval (much much less often than for each access) clear all the reference bits
  - Only PTE without the ref bit clear are eligible for eviction
- More than 1 bit of state?
  - Associate some number of counter bits
  - At regular interval, if ref bit is 0 increment counter and if ref bit is 1 then zero counter
  - Counter tells you # intervals since the last reference
  - More bits you give to counter = more accurate approximation

LRU Clock

- Also called Second Chance
- Logically put all physical page frames in a circle (clock)
- Maintain a pointer to a current page (clock hand)
- When need to evict a page, look at current page
  - If ref bit off then evict
  - If ref bit on clear it and move on (second chance)

LRU Clock (con't)

- Arm moves as quickly as eviction are requested
- If evictions rarely requested then arm moves slowly and pages have a long time to prove their worth by being referenced
- If evictions frequently requested then arm moves fast and little time before the second chance is up

Fairness?

- All the replacement policies we've looked at so far just try to pick the page to evict regardless of which process the page belongs to
- What if demand page in from one process causes the eviction of another processes page? Is that fair?
- On the other hand is it fair for one process to have 2 times their working set while another process has ½ their working set and is paging heavily?

Fixed vs Variable Space

- Fixed space algorithms
  - Give each process a limit of pages it can use
  - When it reaches its limit, it replaces LRU or FIFO or whatever from its pages
  - May be more natural to give process a say in the replacement policy used for its pages
- Variable space algorithms
  - Processes set of pages grows and shrinks
  - One process can ruin it for the rest but opportunity to make globally better decisions

Use Working Set

- Could ask each process to inform the OS of the size of its working set
- OS only allow a process to start if it can allocate the complete working set
- How easy for processes to report this?

Page Fault Frequency (PFF)

- PFF is a variable-space algorithm that tries to determine the working set size dynamically
- Monitor page fault rage for each process
- If fault rate is above a given threshold, give it more memory
- If fault rate is below threshhold, take away memory
- Constant adjustment? Dampening factor so only changes occasionally

Best page replacement?

- Of course it depends ☺
- Interestingly if have too much memory it doesn't matter
  - anything you do will be ok (overprovisioning)
- Also doesn't matter if have too little memory
  - Thrashing and nothing you can do to stop it (overcommitted)
- So much does it cost just to overprovision?

Summary

- Demand paging
  - Start with no physical memory pages mapped and load them in on demand
- Page replacement Algorithms
  - Belady – optimal but unrealizable
  - FIFO – replace page loaded earliest
  - LRU – replace page referenced earliest
  - Working Set – keep set of pages in memory that induces minimal fault rate (need program specification)
  - PFF – Grow/shrink page set as a function of fault rate
- Fairness – globally optimal replacement vs protecting processes from each other?

Outtakes

- Shared memory machines
- Expanding address spaces 16 to 32 bit
- Inverted page tables
- Multics