Virtual memory

- We've hinted that all the pages of an address space do not need to be resident in memory
  - the full (used) address space exists on secondary storage (disk) in page-sized blocks
  - the OS uses main memory as a (page) cache
  - a page that is needed is transferred to a free page frame
  - if there are no free page frames, a page must be evicted
    - evicted pages go to disk (only need to write if they are dirty)
  - all of this is transparent to the application (except for performance …)
  - managed by hardware and OS
- Traditionally called virtual memory

Reminder: Page Table Entries (PTEs)

<table>
<thead>
<tr>
<th>Letter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>Valid bit says whether or not the PTE can be used</td>
</tr>
<tr>
<td>R</td>
<td>Referenced bit says whether the page has been accessed</td>
</tr>
<tr>
<td>M</td>
<td>Modified bit says whether the page is dirty</td>
</tr>
<tr>
<td>P</td>
<td>Page frame number determines the physical page</td>
</tr>
<tr>
<td>prot</td>
<td>Protection bits control which operations are allowed</td>
</tr>
<tr>
<td>M</td>
<td>Read, write, execute</td>
</tr>
</tbody>
</table>

Page faults

- What happens when a process references a virtual address in a page that has been evicted?
  - when the page was evicted, the OS set the PTE as invalid and stored the disk location of the page either in the PTE or in a parallel data structure
  - when a process tries to access the page, the invalid PTE will cause an exception (page fault) to be thrown
  - OK, it's actually an interrupt!
  - the OS will run the page fault handler in response
    - handler uses invalid PTE to locate page on disk
    - handler reads page into a physical frame, updates PTE to point to it and to be valid
    - OS restarts the faulting process
    - there are a million and one details …

Demand paging

- Pages are only brought into main memory when they are referenced
  - only the code/data that is needed (demanded!) by a process needs to be loaded
  - what’s needed changes over time, of course… Hence, it’s called demand paging
- Few systems try to anticipate future needs
  - OS crystal ball module notoriously ineffective
  - it’s common to cluster pages
  - OS keeps track of pages that should come and go together
  - bring in all when one is referenced
  - interface may allow programmer or compiler to identify clusters
- when you read in a page, where does it go?
  - if there are free page frames, grab one
    - what data structure might support this?
  - if not, must evict something else
  - this is called page replacement

Page replacement

- When you read in a page, where does it go?
  - try to pick a page that won’t be needed in the near future
  - try to pick a page that hasn’t been modified (thus saving the disk write)
  - OS typically tries to keep a pool of free pages around so that allocations don’t inevitably cause evictions
  - OS also typically tries to keep some “clean” pages around, so that even if you have to evict a page, you won’t have to write it
  - accomplished by pre-writing when there’s nothing better to do
  - much more on this later!
How do you "load" a program?

• Create process descriptor (process control block)
• Create page table
• Put address space image on disk in page-sized chunks
• Build page table (pointed to by process descriptor)
  - all PTE valid bits 'false'
  - each entry indicates the disk location of the corresponding page
  - when process starts executing:
    - instructions immediately fault on both code and data pages
    - faults taper off, as the necessary code/data pages enter memory

Oh, man, how can any of this possibly work?

• Locality!
  - temporal locality
    - locations referenced recently tend to be referenced again soon
  - spatial locality
    - locations near recently referenced locations are likely to be referenced soon (think about why)
• Locality means paging can be infrequent
  - once you’ve paged something in, it will be used many times
  - on average, you use things that are paged in
  - but, this depends on many things:
    - degree of locality in the application
    - page replacement policy and application reference pattern
    - amount of physical memory vs. application “footprint” or “working set”

Evicting the best page

• The goal of the page replacement algorithm:
  - reduce fault rate by selecting best victim page to remove
    - “system” fault rate or “program” fault rate??
  - the best page to evict is one that will never be touched again
    - duh ...
    - “never” is a long time
  - Belady’s proof: evicting the page that won’t be used for the longest period of time minimizes page fault rate
• Rest of this module:
  - survey a bunch of page replacement algorithms
  - for now, assume that a process pages against itself, using a fixed number of page frames

#1: Belady’s Algorithm

• Provably optimal: lowest fault rate (remember SJF?)
  - evict the page that won’t be used for the longest time in future
  - problem: impossible to predict the future
• Why is Belady’s algorithm useful?
  - as a yardstick to compare other algorithms to optimal
    - if Belady’s isn’t much better than yours, yours is pretty good
    - how could you do this comparison?
• Is there a best practical algorithm?
  - no; depends on workload
• Is there a worst algorithm?
  - no, but random replacement does pretty badly
  - don’t laugh -- there are some other situations where OS’s use near-random algorithms quite effectively!

#2: FIFO

• FIFO is obvious, and simple to implement
  - when you page in something, put it on the tail of a list
  - evict page at the head of the list
• Why might this be good?
  - maybe the one brought in longest ago is not being used
• Why might this be bad?
  - then again, maybe it is being used
  - have absolutely no information either way
• In fact, FIFO’s performance is typically lousy
• In addition, FIFO suffers from Belady’s Anomaly
  - there are reference strings for which the fault rate increases when the process is given more physical memory

#3: Least Recently Used (LRU)

• LRU uses reference information to make a more informed replacement decision
  - idea: past experience is a decent predictor of future behavior
  - on replacement, evict the page that hasn’t been used for the longest period of time
  - LRU looks at the past, Belady’s wants to look at the future
  - how is LRU different from FIFO?
  - can you think of an example where LRU would be terrible?
    - in general, it works exceedingly well
• Implementation
  - to be perfect, must grab a timestamp on every memory reference, put it in the PTE, order or search based on the timestamps …
  - way too $$$ in memory bandwidth, algorithm execution time, you name it …
Approximating LRU

- Many approximations, all use the PTE reference bit
  - keep a counter for each page
  - at some regular interval, for each page, do:
    - if ref bit = 0, increment the counter (hasn’t been used)
    - if ref bit = 1, zero the counter (has been used)
  - regardless, zero ref bit
  - the counter will contain the # of intervals since the last reference to the page
  - page with largest counter is least recently used
- Some architectures don’t have PTE reference bits
  - can simulate reference bit using the valid bit to induce faults
    - hack, hack, hack

#4: LRU Clock

- AKA Not Recently Used (NRU) or Second Chance
  - replace page that is “old enough”
  - logically, arrange all physical page frames in a big circle (clock)
    - just a circular linked list
    - a “clock hand” is used to select a good LRU candidate
  - sweep through the pages in circular order like a clock
    - if ref bit is off, it hasn’t been used recently, we have a victim
      - so, what is minimum “age” if ref bit is off?
    - if the ref bit is on, turn it off and go to next page
  - arm moves quickly when pages are needed
  - low overhead if have plenty of memory
  - if memory is large, “accuracy” of information degrades
    - add more hands to fix

Allocation of frames among processes

- FIFO and LRU Clock each can be implemented as either local or global replacement algorithms
  - local
    - each process is given a limit of pages it can use
    - if “pages against itself” (evicts its own pages)
  - global
    - the “victim” is chosen from among all page frames, regardless of owner
    - processes’ page frame allocation can vary dynamically
- Issues with local replacement?
- Issues with global replacement?
  - linux uses global replacement

Hybrid algorithms
- local replacement
- an explicit mechanism for adding or removing page frames

Issues with all 3 approaches?

The working set model of program behavior

- The working set of a process is used to model the dynamic locality of its memory usage
  - working set = set of pages process currently “needs”
  - formally defined by Peter Denning in the 1960’s
- Definition:
  - \( WS(t,w) = \{ \text{pages P such that P was referenced in the time interval (t, t-w)} \} \)
  - t: time
  - w: working set window (measured in page refs)
    - a page is in the working set (WS) only if it was referenced in the last w references
    - obviously the working set (the particular pages) varies over the life of the program
    - so does the working set size (the number of pages in the WS)
Working set size

- The working set size, |WS(t,w)|, changes with program locality
  - during periods of poor locality, more pages are referenced
  - within that period of time, the working set size is larger
- Intuitively, the working set must be in memory, otherwise you’ll experience heavy faulting (thrashing)
  - when people ask “How much memory does Netscape need?”, really they’re asking “what is Netscape’s average (or worst case) working set size?”

#5: Hypothetical Working Set algorithm

- Estimate |WS(0,w)| for a process
- Allow that process to start only if you can allocate it that many page frames
- Use a local replacement algorithm (LRU Clock?) make sure that “the right pages” (the working set) are occupying the process’s frames
- Track each process’s working set size, and reallocate page frames among processes dynamically
- Problem? Solution?
- What the heck is w?

#6: Page Fault Frequency (PFF)

- PFF is a variable-space algorithm that uses a more ad hoc approach
- Attempt to equalize the fault rate among all processes, and to have a “tolerable” system-wide fault rate
  - monitor the fault rate for each process
  - if fault rate is above a given threshold, give it more memory
    • so that it faults less
  - if the fault rate is below threshold, take away memory
    • should fault more, allowing someone else to fault less

Thrashing

- Thrashing is when the system spends most of its time servicing page faults, little time doing useful work
  - could be that there is enough memory but a lousy replacement algorithm (one incompatible with program behavior)
  - could be that memory is over-committed
    • too many active processes

Number of active processes vs. System throughput (requests/sec.) with zero overhead

Number of active processes vs. System throughput (requests/sec.) with thrashing

Why?
Where is life interesting?

• Not if system has too much memory
  – page replacement algorithm doesn’t much matter (over-provisioning)
• Not if system has too little memory
  – page replacement algorithm doesn’t much matter (over-committed)
• Life is only interesting on the border between over-provisioned and over-committed
• Networking analogies
  – Aloha Network as an example of thrashing
  – over-provisioning as an alternative to Quality of Service guarantees

Summary

• Virtual memory
• Page faults
• Demand paging
  – don’t try to anticipate
• Page replacement
  – local, global, hybrid
• Locality
  – temporal, spatial
• Working set
• Thrashing

Page replacement algorithms

• #1: Belady’s – optimal, but unrealizable
• #2: FIFO – replace page loaded furthest in the past
• #3: LRU – replace page referenced furthest in the past
  • approximate using PTE reference bit
• #4: LRU Clock – replace page that is “old enough”
• #5: Working Set – keep the working set in memory
• #6: Page Fault Frequency – grow/shrink number of frames as a function of fault rate