

CS 414 : Operating Systems

UNIVERSITY OF VIRGINIA
Department of Computer Science

Spring 2008

Topic 14: Virtual Memory, Demand Paging, and Working Sets

- Readings for this topic: Ch.9
- Separation of the programmer's view of memory from the system's view using a mapping mechanism. Each sees a different organization. This makes it easier for the OS to shuffle users around and simplifies memory sharing between users.
- However, until now a user process had to be completely loaded into memory before it could run. Is it absolutely necessary?
- What is virtual memory? A technique that allows ...
 - Is it the same as demand paging?
- Introduce the third party into the scene.
 - The idea is to produce the illusion of a disk as fast as main memory.
- The reason that this works is ...
 - What is 90-10 rule?
 - Is past is a good predictor of future?
 - What if processes access pages randomly?
- Two issues:
 - How does it work? Mechanisms to handle when it references a page that is only in the backing store.
 - Scheduling decisions: When to move which pages between memory and disk.

- Basic mechanism: what to do when the page is not in memory.
 - First, extend the page tables with an extra bit “present” (also called the valid bit). If valid bit isn’t set then a reference to the page results in a trap. What is this special trap called?
 - Any page not in main memory right now has the valid bit cleared in its page table entry.
 - When page fault occurs:
 - Operating system brings page into memory
 - Page table is updated, valid bit is set.
 - The process continues execution.
- Continuing process is very tricky. Why?
 - Can the instruction just be skipped?
 - Try to re-execute the instruction from the beginning. Problems?
 - Without additional information from the hardware, it may be impossible to restart a process after a page fault.
 - Non-virtualizable machines
- Scheduling decisions:
 - Page selection: when to bring pages into memory.
 - Page replacement: which page(s) should be thrown out, and when.
- Page selection Algorithms:
 - Demand paging: start up process with no pages loaded, load a page when a page fault for it occurs, i.e. wait until it absolutely MUST be in memory.
 - Request paging: let user say which pages are needed. What’s wrong with this?
 - Prepaging: bring a page into memory before it is referenced. Hard to do effectively without a prophet, may spend a lot of time doing wasted work.
- Page Replacement Algorithms:
 - Random: pick any page at random (works surprisingly well!).
 - FIFO: throw out the page that has been in memory the longest. The idea is to be fair, give all pages equal residency.
 - MIN: throw out the page that won’t be used for the longest time into the future. This re-

quires a prophet, so it isn't practical, but it is good for comparison. As always, the best algorithm arises if we can predict the future.

- LRU: throw out the page that hasn't been used in the longest time. Use the past to predict the future. With high locality, it's a good approximation to MIN.
- Example: Try the reference string A B C A B D A D B C B, assume there are three page frames of physical memory.
- MIN is optimal (can't be beaten), but the principle of locality states that past behavior predicts future behavior, thus LRU should do just about as well.
- Belady's anomaly (for non-stack algorithms)
- Implementing LRU: need some form of hardware support in order to keep track of which pages have been used recently.
 - Perfect LRU? Keep a register for each page, and store the system clock into that register on each memory reference. To replace a page, scan through all of them to find the one with the oldest clock. This is expensive if there are a lot of memory pages.
 - In practice, nobody implements perfect LRU. Instead, we settle for an approximation that is efficient. Just find an old page, not necessarily the oldest.
- Clock algorithm (also called "second chance" algorithm): keep "use" bit for each page frame, hardware sets the appropriate bit on every memory reference. The operating system clears the bits from time to time in order to figure out how often pages are being referenced. Introduce clock algorithm where to find a page to throw out the OS circulates through the physical frames clearing use bits until one is found that is zero. Use that one.
- Some systems also use a "dirty" bit to give preference to dirty pages. This is because it is more expensive to throw out dirty pages: clean ones need not be written to disk.

- What does it mean if the clock hand is sweeping very slowly?
- What does it mean if the clock hand is sweeping very fast?
- Three different styles of replacement:
 - *Global replacement*: all pages from all processes are lumped into a single replacement pool. Each process competes with all the other processes for page frames.
 - *Per-process replacement*: each process has a separate pool of pages. A page fault in one process can only replace one of that process's frames. This relieves interference from other processes.
 - *Per job replacement*: lump all processes for a given user into a single replacement pool.
 - In per-process and per-job replacement, must have a mechanism for (slowly) changing the allocations to each pool. Otherwise, can end up with very inefficient memory usage.
 - Global replacement provides most flexibility, but least "pig protection".
- Thrashing: consider what happens when memory gets overcommitted.
 - Suppose there are many users, and that between them their processes are making frequent references to 50 pages, but memory has 49 pages.
 - Each time one page is brought in, another page, whose contents will soon be referenced, is thrown out.
 - How serious? Compute average memory access time.
 - What is a more serious (real) problem? The system will spend all of its time reading and writing pages. It will be working very hard but not getting anything done.
- Thrashing occurs because the system doesn't know when it has taken on more work than it can handle. LRU mechanisms order pages in terms of last access, but don't give absolute numbers indicating pages that *must not* be thrown out.
- What can be done?

- If a single process is too large for memory, there is nothing the OS can do. That process will simply thrash.
- If the problem arises because of the sum of several processes:
 - Figure out how much memory each process needs.
 - Change scheduling priorities to run processes in groups whose memory needs can be satisfied. Shed load.
- *Working Sets* proposed by Peter Denning. An informal definition is “the collection of pages that a process is working with, and which must thus be resident if the process is to avoid thrashing.” The idea is to use the recent needs of a process to predict its future needs.
 - Choose tau, the working set parameter. At any given time, all pages referenced by a process in its last tau seconds of execution are considered to comprise its *working set*.
 - A process will never be executed unless its working set is resident in main memory. Pages outside the working set may be discarded at any time.
- Working sets are not enough by themselves to make sure memory doesn't get overcommitted. We must also introduce the idea of a *balance set*:
 - If the sum of the working sets of all runnable processes is greater than the size of memory, then refuse to run some of the processes (for a while).
 - Divide runnable processes up into two groups: active and inactive. When a process is made active its working set is loaded, when it is made inactive its working set is allowed to migrate back to disk. The collection of active processes is called the *balance set*.
 - Some algorithm must be provided for moving processes into and out of the balance set. What happens if the balance set changes too frequently?
- Problem with the working set: must constantly be updating working set information.
 - One of the initial ideas was to store a capacitor with each memory page. The capacitor would be charged on each reference, then would discharge slowly if the page wasn't referenced. Tau would be determined by the size of the capacitor.

- Actual solution: take advantage of use bits.
 - OS maintains *idle time* value for each page: amount of CPU time received by process since last access to page.
 - Every once in a while, scan all pages of a process. For each use bit on, clear page's idle time. For use bit off, add process' CPU time (since last scan) to idle time. Turn all use bits off during scan.
 - Scans happen on order of every few seconds (in Unix, tau is on the order of a minute or more).

- Other questions about working sets and memory management in general:
 - What should tau be?
 - What if it's too large?
 - What if it's too small?
 - What algorithms should be used to determine which processes are in the balance set?
 - How do we compute working sets if pages are shared?
 - How big should memory be? It's a bottleneck issue: CPU, memory, or something else.

- Other useful control method: use PF frequency as metric

- Issue of page size
 - Advantages and disadvantages of small vs large page size