Virtual Memory: Demand Paging and Page Replacement

Problems that remain to be solved:

- Even with 8K page size, number of pages /process is very large. Can't afford to keep the entire page table for a process in physical memory.
- Address spaces are becoming large: 64-bit address spaces in Sun's UltraSparc, DEC's Alpha processor, and Intel Itanium processor.
    - Don't really need to keep the entire process in memory: most processes use only a small part of their address space actively. We can thus potentially fit even more processes in memory, increasing the degree of multiprogramming and system throughput

Back-of-the-envelope calculations:

Assume 32-bit architecture. Page size of 4K bytes. What is the # of bits required to represent the page # and the offset?

What is the maximum number of pages per process?

What is the number of bits required in a page table entry to represent the frame number (physical page #)?

Assuming each PTE is 4 bytes, what is the maximum size of page table (in bytes) required to store the page table?

Assuming you have 1 GB of physical RAM, would you be able to page table in memory? What percentage does it take?

Is there something to worry about?

Solutions:

Draw view of virtual address space, translated to either a) physical memory (small, fast) or b) disk (backing store), which is large but slow.

The idea is to produce the illusion of virtual memory that is as fast as the physical memory but as big as the disk.

The reason that this works is that most programs spend most of their time in only a small piece of the code. Two principles:

- **Temporal locality:** same memory locations are often referenced repeatedly (e.g., code in a loop)
- **Spatial locality:** If location x is referenced, it is also likely that nearby locations will be referenced. (e.g., traversing an array, advancing program

counter).

Page faults:

If not all of process is loaded when it is running, what happens when it references a byte that is only in the backing store? Hardware and software cooperate to make things work anyway.

First, extend (in hardware) the page tables with an extra bit, called *present bit*. If present bit isn't set then a reference to the page results in a trap. This trap is given a special name, *page fault*.
Any page not in main memory right now has the present bit cleared in its page table entry.

When page fault occurs:

- Control transfers from the user process to the operating system and the operating system brings page into memory.

- Page table is updated, ``present'' bit is set.

The process can later continue execution.

Continuing a process after a page fault:

Continuing process is very tricky, since page fault may have occurred in the middle of an instruction. Don't want user process to be aware that the page fault even happened.

Can the instruction just be skipped?

Suppose the instruction is restarted from the beginning?

Without additional information from the hardware, it may be impossible to restart a process after a page fault. Machines that permit restarting must have hardware support to keep track of all the side effects so that they can be undone before restarting.

If you think about this when designing the instruction set, it isn't too hard to make a machine virtualizable. It's much harder to do after the fact. Almost all processors today are virtualizable.

Page Size

What happens if the page size is too small?

What happens if it is too large?

What is a good page size in practice?

Page sizes used to be about 512 bytes, but have been going up to about 4K as

memory becomes cheaper and more plentiful. Performance considerations outweigh space wastage.

Once the hardware has provided basic capabilities for virtual memory, the OS must make two kinds of scheduling decisions:

Page selection: when to bring pages into memory.
Page replacement: which page(s) should be thrown out, and when.
.

Page selection Strategies:

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. Almost all paging systems are like this.

Reques*t paging:* let user say which pages are needed. What's wrong with this?

Some systems, in addition to demand paging, will do:*Prepaging:* bring a page into memory before it is referenced (e.g. when one page is referenced, bring in the next one, just in case). What can go wrong in always doing this?

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. What is wrong with this?

MIN (optimal): as always, the best algorithm arises if we can predict the future.

How to approximate MIN: use the past to predict the future.

LRU: Throw out the page that hasn't been used in the longest time. If there is locality, then this is a good approximation to MIN.

LRU 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. Show the memory allocation state after each memory reference.

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.

Clock Algorithm (2nd Chance):

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.

Key observation:

Thus, 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):

- 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.
- 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. The hand stops once the page to be replaced is found.

Show a picture:

What happens if all the use bits are set? Will the OS find a page to replace?

Does the OS always start searching from page 0?

How to replace a page?

Step 1: The page frame being thrown out needs to be written to the backing store (disk).
Step 2: bring in the page on which page fault occured from the disk.

Step 1 can be avoided if the page frame being thrown out hasn't changed since it was last bought from the disk. Disk will still have a correct copy. Thus, we want to add a **dirty bit** to the page table to track whether a page has changed since it was last brought in.

One can modify clock algorithm so that we avoid replacing 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?

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.

The system will spend all of its time reading and writing pages. It will be working

very hard but not getting much done.

The progress of the programs will make it look like the access time of memory is as slow as disk, rather than disks being as fast as memory

Thrashing was a severe problem in early demand paging systems.
Thrashing occurs because the system doesn't know when it has taken on more work than it can handle. LRU-type mechanisms, such as clock, order pages in terms of last access, but don't give absolute numbers indicating pages that mustn't 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 are a solution to thrashing proposed by Peter Denning. 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.

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) – swap them out to disk, for example and bring them back later.

How to determine when a page was last accessed? 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 as follows:

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.

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?

(In Unix, tau is of the order of a minute or more.)