Still VM: Global/Local Allocation, Page Table Management and Optimizations

Local and Global Allocation Strategies

Up until now we have tacitly assumed that only one program has been faulting pages in and out of memory; this is not usually the case. One question that the OS has to answer on a page fault is which process will lose a page.

- local allocation assigns each process a fixed number of pages and a process can only evict its own pages. This is the same as each process having a fixed size memory allocation.

- global allocation allows processes to fight it out among each other.

Even global allocations need to control how many pages are allocated to a process. Processes need a minimum number of pages to make any forward progress (a single instruction might require 6 pages to be present!). Furthermore, taking steps to keep processes' page allocations close to the size of their working set prevents *memory thrashing,* which is the memory analog to CPU thrashing we talked about earlier. Some ways to do this:

- Assign equal shares of memory (with a few left over to speed page faults).[1] This is not very effective if process working sets vary widely.

- Use the wsclock algorithm to estimate working set size. This fares poorly if working sets change quickly.

- Use *page fault frequency (PFF)* to determine the allocations. If a process page rate is higher than some high-water mark, increase its allocation, if it gets too low, the process is a candidate to have it's allocation reduced.

Note that it's possible to have too many processes competing for memory under any one of these systems. In that case, a process must actually be swapped out. Even paging systems can benefit from swapping.

Page Tables

The format of page tables in memory is generally determined by the expectations of the memory management system. The MMU expects page tables in a given format, and the OS has to abide by that. The OS may keep its own per-page information in separate tables. The OS may want to keep track of the address of the pages in backing store, the process to which each page belongs, etc.

As we've alluded to, hardware page tables keep track of a variety of nuts and bolts kinds of things, for example:

Referenced bit
Modified bit
Cached bit
protection
present/absent
Page frame (disk sectors are elsewhere)

There are 2 important things to consider when managing page tables: they are big, and access to them needs to be fast. Most paging designs come from trading these two attributes against each other. Let's address size first.

---

[1] Most paging algorithms try to leave a few free page frames so that a new page can be put there immediately, or so that a new page can be put in from backing store without having to empty a dirty page.

Multi-level Page Tables

One way to squeeze space in the page table is to avoid allocating space for parts of the address space that are not in memory. Multilevel page tables are a good way to do this. The page identifier is broken down into sub-indices, each of which corresponds to a sub-page-table. The more levels of division, the more precise the allocation (although there's a limit because of overhead). However, the more levels of division, the more memory references are needed to resolve an address.

Second Level Page Table

Top Level Page Table
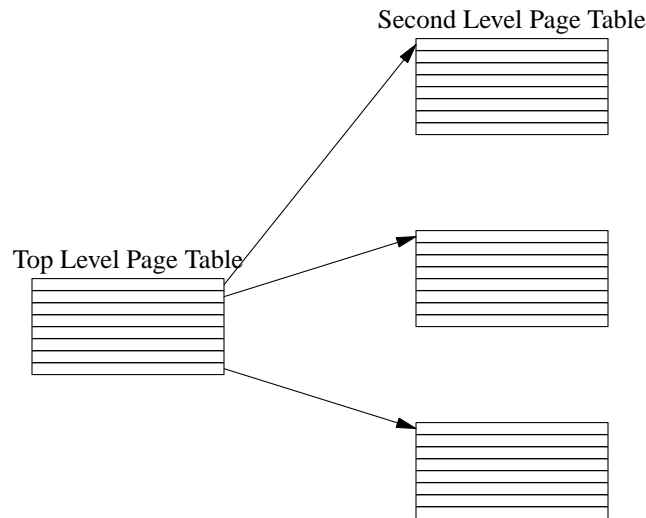
Multilevel page tables effectively address the space concerns of page tables.

Paging Efficiency

Paging is expensive, and the more OS and hardware designers can speed up the common case of access to memory, the better the system performs. One of the most powerful tools in this never-ending battle is associative memory.

Most memory is a set of named places to put data - an address maps to a word of data. Associative memory is a fast, keyed search in hardware. Data is inserted into the associative memory with a key, and retrieved by key, not address. The size of the key and the size of the memory are unrelated (unlike standard memory which reserves enough space for the whole address space). Small associative memories can be made fast and can use variable sized keys. They're expensive, though.

Lots of memory management hardware takes advantage of associative memory to avoid doing page table reads (which are in slower main memory)[2]

When a page is accessed, the virtual address is looked up in the associative memory. If the mapping isn't in the memory, the correct mapping is read from the page table in main memory and put into the associative memory. The physical address is the data and the virtual address the key. Subsequent accesses of that page will use the fast copy in the associative memory. This is an example of *caching*, an important technique in OS. Such an associative memory is called by many names, but one of the most common is the translation lookaside buffer (TLB).

Note that the TLB must also be managed by the OS and/or the hardware. A TLB miss is analogous to a page table miss. In most TLB systems, the hardware automatically manages the TLB using an LRU algorithm. Some systems, like the MIPS CPU, decided to save the hardware space and rely on the OS to manage both page tables and the TLB. You'll get to do that in assignment 3.

Other Interesting Paging Variants

There are several paging studies in Tannenbaum that are worth your time. One of the more interesting are the PDP's use of 2 address spaces to double the amount of memory addressed by its 16 bit

[2] Associative memories are closer to the top of the storage hierarchy than main memory.

addresses. Code and data inhabited 2 distinct address spaces: a data fetch for am address and a code fetch for the same address would return different values.

Another interesting study is the inverted page tables used by several machines (System 38, HP Spectrum, and one of the big IBM parallel machines whose name escapes me). Machines with 64-bit virtual address spaces can address more memory space than there are particles in the known universe. Even using multi-level page tables these address spaces can potentially create monstrous page tables. The solution offered by inverted page tables is not to track the virtual pages, but rather the physical frames. On a memory access, rather than scanning the page table to see where the referenced virtual page is, the hardware searches the physical pages looking for the one that holds the virtual page of interest. The advantage is that the page table size is always bounded by the size of physical memory.