CSE321 Theory Assignment 02

NAME: MD. SABBIR AKON

ID: 22301242

SECTION: 07

Memory Organization and Structure

Memory Layout and Organization

The memory layout in the hashed page table is organized as a hash table. Each item in the hash table contains the virtual page number, the corresponding physical frame number and, often, a pointer to the next entry to deal with collisions. This enables the system to easily find mappings without needing to store an enormous table which is linear. For example, if a virtual page number is hashed, it directly points to corresponding bucket and from there the system checks the chain until found the right entry. This design makes hashed tables practical for 64-bit systems with huge address spaces.

In case of an inverted page table, the layout is very different. There is not one entry for each virtual page in memory, but only one for each physical frame in memory. Each entry is used to store the virtual page number (and process ID if more than one process is running) currently stored in that frame. The size of the inverted page table is then dependent on physical memory, rather than virtual space. For example, a machine with 8 GB of physical memory that is organized into 4 KB frames will have about two million entries, whatever the size of the virtual address space.

Virtual-to-Physical Address Translation

In hashed page table, translation starts by getting virtual page number and applying a hash function to it. This hash result is used to point to an index in the hash table, in which the system searches for an entry that matches the virtual page number. Once a match is found, the corresponding physical frame number is fetched. Finally the page offset of the virtual address is added to the frame number to make the physical address.

In inverted page tables, it's a bit more complicated. Since the table is not consulted by virtual pages, but by physical frames, the system cannot directly access the table by the virtual page number. Instead, the operating system hashes the entry or performs an associative search for the entry matching the virtual page number and process ID. Once the match is found the frame number from the entry is used with the offset to construct the physical address. This lookup step is what makes inverted page tables slower in comparison to hashed ones unless backed up by special hardware.

Memory Overhead and Space Efficiency

Hashed page tables minimize memory overhead over linear or multi-level page tables, but do still depend on the number of virtual pages in use. This means that in systems where there is sparse usage over a large virtual space, the hash table nonetheless consumes space proportional to virtual activity. Collisions may also cause some wasted space due to linked chains.

Inverted page tables, however, are extremely space efficient. Their size is not dependent upon the virtual address space, only the number of physical frames in the memory. For example, whether the virtual space is 32-bit or 64-bit, the size of the inverted page table does not change if physical memory is fixed. This efficiency makes inverted tables attractive in modern systems in which the physical memory is much smaller than the enormous virtual space.

Performance Characteristics

Impact of Hash Collisions and Search Overhead

In hashed page tables, the main performance problem is with hash collisions. If multiple virtual pages hash to the same index, the system has to traverse a chain of entries to locate the appropriate entry. This makes translation time longer. With a good hash function and a big enough table, it doesn't happen too often, but in worst cases, long chains can be quite painful when looking them up.

With inverted page tables, the performance issue is a different one. Since the table is sorted by frames, not pages, the system has to look up to determine which frame contains a given virtual page. This search can be slow if done linearly so most implementations use a hashing or associative search instead. While this cuts down the overhead, it is still more complex than hashed page tables and requires hardware support often.

TLB Miss Handling

On a TLB miss in a hashed page table the system will compute the hash of the virtual page and look up the chain at that index. This is usually fast unless there are many collisions in existence. Once the entry is found, the mapping is put into the TLB for use in the future.

On a TLB miss in an inverted page table, the process of lookup is heavier. Since the inverted table can't be directly indexed by the virtual page, the system must do a search using hashing or associative memory. This additional step makes TLB misses more expensive in the case of inverted tables than for hashed tables.

Page Fault Processing

For hashed page tables, when the virtual page is not in memory (page fault), as soon as the missing page is restored to memory, the OS adds a new entry to the hash table that associate the virtual page with the allocated physical frame of memory.

With inverted page tables, the process of update is much easier. Since the table is indexed by frames, the system updates the entry of the frame in which the page is loaded directly, and stores the virtual page number (and process ID). This makes page fault-handling simple, but the lookup cost during normal operation is higher.

Both hashed and inverted page tables have the goal of optimizing the mapping between virtual and physical addresses, but they are a different approach. Hashed page tables focus on faster average translation (through the use of hashing) but are at the risk of slower performance in the case of heavy collisions. Inverted page tables are very efficient in their memory usage in the sense that the size of the structure is only in relation to the physical memory, but they are inefficient in terms of higher look up costs and complicated TLB miss handling. In practice, operating systems use a balance of these trade-offs with TLBs, and hardware optimizations, to achieve both efficiency and performance.