**CS162, Spring 2004** **Discussion #10** **Amir Kamil**
**UC Berkeley** **4/1/04**

**Topics: Inverted Page Tables, TLBs**

# 1 Inverted Page Tables

## 1.1 Introduction

Consider a system in which the virtual address space is 64 bits, the page size is 4KB, and the amount of physical memory is 512MB. How much space would a simple single-level page table take? Such a table contains one entry per virtual page, or $2^{64-12} = 2^{52}$ entries. Each entry would require about 4 bytes, so the total page table size is $2^{54}$ bytes, or 16 *petabytes* (peta- > tera- > giga-)! And this is for each process!

Of course, a process is unlikely to use all 64 bits of address space, so how about using multilevel page tables? How many levels would be required to ensure that each page table require only a single page? Assuming an entry takes a constant 4 bytes of space, each page table can store 1024 entries, or 10 bits of address space. Thus $\lceil 52/10 \rceil = 6$ levels are required. But this results in 6 memory accesses for each address translation!

But notice that there are only 512MB of memory in the system, or $2^{29-12} = 2^{17} = 128\text{K}$ physical pages. If we can somehow manage to store only a single page table entry per *physical* page, the page table size decreases considerably, to 2MB assuming each entry takes 16 bytes. And since processes share physical memory, we need only have a single global page table. This is the concept of an *inverted page table.*

## 1.2 Linear Inverted Page Tables

The simplest form of an inverted page table contains one entry per physical page in a linear array. Since the table is shared, each entry must contain the process ID of the page owner. And since physical pages are now mapped to virtual, each entry contains a virtual page number instead of a physical. The physical page number is not stored, since the index in the table corresponds to it. As usual, information bits are kept around for protection and accounting purposes. Assuming 16 bits for a process ID, 52 bits for a virtual page number, and 12 bits of information, each entry takes 80 bits or 10 bytes, so the total page table size is $10 \cdot 128\text{KB} = 1.3\text{MB}$.

In order to translate a virtual address, the virtual page number and current process ID are compared against each entry, traversing the array sequentially. When a match is found, the index of the match replaces the virtual page number in the address to obtain a physical address. If no match is found, a page fault occurs. This translation procedure is shown in figure 1.

While the table size is small, the lookup time for a simple inverted page table can be very large. Finding a match may require searching the entire table, or 128K memory accesses! On average, we expect to take 64K accesses in order to translate an address. This is far too inefficient, so in order to reduce the amount of required searching, hashing is used.

## 1.3 Hashed Inverted Page Tables

A hashed inverted page table adds an extra level before the actual page table, called a *hash anchor table.* This table is at least as large as the page table, and maps process IDs and virtual page numbers to page table entries. Since collisions may occur, the page table must do chaining. Since each member in the chain must map to a physical page and therefore must have a corresponding page table entry, the chain can be represented as a sequence of page table entries, with each entry pointing to the next entry in the chain. This requires an extra 4 bytes per entry, so the page table size is now $14 \cdot 128\text{KB} = 1.8\text{MB}$. The hash anchor table takes an additional $4 \cdot 128\text{KB} = 512\text{KB}$, for a total of 2.3MB.

Now in order to translate a virtual address, the process ID and virtual page number are hashed to get an entry in the hash anchor table. The entry's pointer to a page table entry is followed, and the process ID and virtual page number are compared against that stored there. If they don't match, the page table entry's next pointer is followed to get another page table entry, and the process is repeated until a match is found.

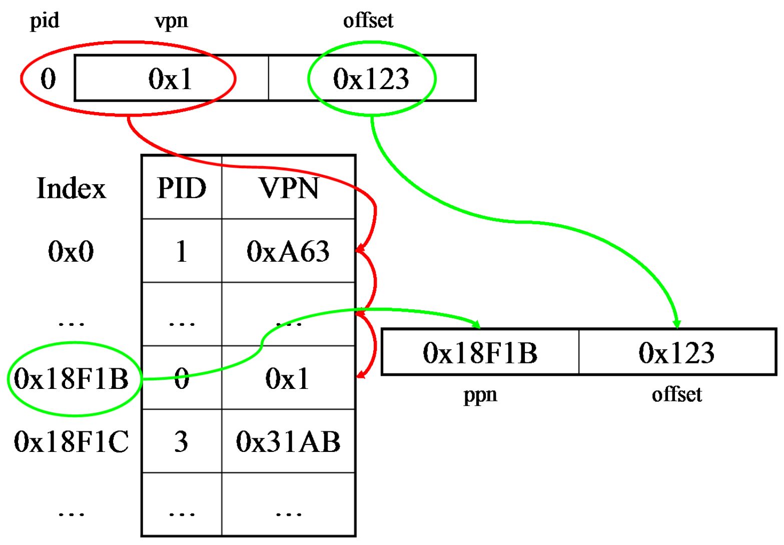

Figure 1: Translation procedure using a linear inverted page table. Info bits exist in each entry, though they are not shown.

If a chain is exhausted by hitting an invalid next pointer, a page fault results. The translation process is illustrated in figure 2.

Assuming a good hash function, the average chain length should be about 1.5, so only 2.5 memory accesses are required on average to translate an address [1]. This is pretty good, considering a two-level page table requires 2 accesses. Caching can still improve the access time considerably.

### 1.4 Nachos Requirements

In phase 3 of nachos, you will be required to implement a hashed inverted page table. You need not implement the full details of the scheme above; since Java already contains hash tables that do chaining, you can store physical page entries in the hash table itself rather than having a separate page table. In that case, you will likely need to implement in addition a separate *core map*, a structure that maps physical pages to virtual pages, in order to implement demand paging. We will discuss this in greater detail later.

### 1.5 References

[1] A. Chang and M. Mergen, “801 storage: architecture and programming,” *ACM Transactions on Computer Systems*, Vol. 6, No. 1, Feb 1998, pp. 32-33.

## 2 TLBs

While the virtual address space abstraction provides protection between programs, and the illusion of seemingly unlimited memory (the details of which we will see later), its performance is relative slow. Consider a system using a two-level page table in which each memory access takes 100ns. How long does it take to perform a single load/store operation? Two levels of address translation must be done, followed by the actual operation, for a total of three memory accesses or 300ns. The situation gets even worse when we introduce memory caches. Suppose we have an L1 cache with an access time of 10ns, and suppose that the actual physical address in the required memory operation is currently in cache. Then instead of taking 10ns for the operation to complete, the virtual memory translation increases the time to 210ns[1]!

[1]This is assuming a *physically addressed cache*, in which physical addresses are used in the cache, and memory translation must occur before accessing the cache. One solution to this problem is to use a *virtually addressed cache*, which uses virtual addresses in the cache, and in which memory translation occurs *after* accessing the cache. This does not help the case in which the actual address is in main memory, for which the overhead is still a hefty 200%.

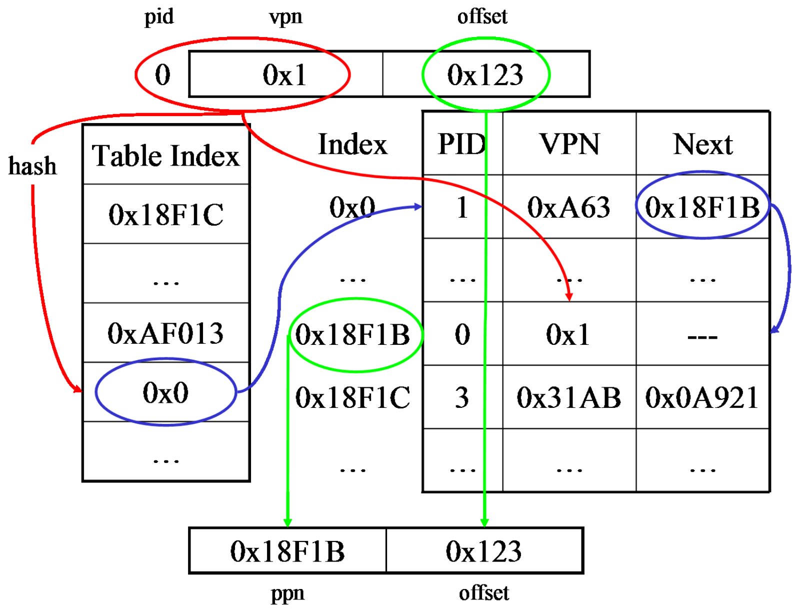

Figure 2: Translation procedure using a hashed inverted page table. Info bits exist in each entry, though they are not shown.

The translation penalty must be paid on all memory accesses. But recall the concept of *spatial locality*: consecutive memory accesses tend to be to locations near each other. In paging, these locations are likely to all be in the same page, so that the same translation is made repeatedly. How can we make this common case faster? By adding a cache of course.

A cache for virtual to physical address translations is known as a *translation lookaside buffer (TLB)*. A TLB consists of a small number of entries, each of which stores a virtual page number, the corresponding physical page number, and some control bits such as a valid bit. Then on translating a virtual address, the CPU first checks the TLB to see if it contains an entry for the address, and if not, continues on to the page table (or traps to the OS and has it check the page table).

As with any cache, the associativity of the TLB can be manipulated to decrease cost or increase hit rate. In a *direct mapped*, each virtual page number can only be mapped to a single TLB entry, usually determined by some hash of its low and high order bits. This results in very low search costs, since only one entry must be checked, but can result in a high miss rate. Suppose a process uses two pages simultaneously, both of which map to the same TLB entry. The entries for the two pages keep knocking each other out of the TLB, a situation called *thrashing*. We can reduce this problem by using a *set associative* cache, in which each virtual page maps to exactly one set of TLB entries, and in which each set consists of a small number of distinct entries each of which can map any page in the set. Such a cache still has low lookup costs, since only one set need be searched, and has a relatively high hit rate. We usually want to maximize hit rate, however, since the cost of a memory access is very high. This can be accomplished using a *fully associative* cache, in which a virtual page translation can go in any entry. In this scheme, all the TLB entries are searched in parallel to minimize lookup time, resulting in high hardware costs. Most TLBs are fully associative, and have a small number of entries, about 64 to 128.

An important measure of the performance of a cache is the *effective access time*. This is the average time required for a memory access, and is computed as the probability of a hit times the cost of a hit plus the probability of a miss times the cost of a miss ($P(hit) \times C(hit) + P(miss) \times C(miss)$). Suppose a TLB access takes 10ns and a memory access 100ns, and that we are using an inverted page table. What is the effective access time it the TLB hits 90% of the time? Computing only the translation time, the cost of a hit is 10ns and of a miss is 250ns, so the effective translation time is $10\text{ns} \cdot 0.9 + 250\text{ns} \cdot 0.1 = 34\text{ns}$. Including the actual memory operation (and ignoring memory cache), the effective access time is $110\text{ns} \cdot 0.9 + 350\text{ns} \cdot 0.1 = 134\text{ns}$, not much more than the 100ns required with no translation.

Besides associativity, other issues need to be considered when using TLBs. How does it affect context switching? Either the TLB must be flushed on a switch, since the translations are no longer valid, or the process ID must be contained in each entry. If the TLB is full, how do we decide which entry to knock out in a set or fully associative TLB? This depends on our *replacement policy*, which we will discuss later.