COMP 3000
Lecture 10, October 12th 2004
Notes by James Francis

Terminology using the 'Office' Analogy

- **Frame** – The actual 'office'. Pages go inside.
- **Page** – The things inside the office.
- **Physical Address** – A particular address such as 1125 Colonel By Drive.
- **Logical address** – A reference to an office such as: 'My office'. The reference will refer to a different office when a different person says it.

In real terms

- **Page** – A fixed length block of memory that has a virtual address.
- **Frame** – A fixed length block of main memory used to hold one page of virtual memory.
- **Physical Address** – Refers to a specific memory address in global memory.
- **Virtual Address** – The address within a virtual address space, which will be mapped onto an address in physical memory by the OS. Typically there is one virtual address space per process, but only one physical address space for the entire computer.

Why Virtual over Physical?

- You can't load two programs into memory that refer to the same physical address.
- If two programs are too big to sit in main memory together, they can still be run using virtual memory.
- If the "office" moves, the physical address changes, but it could still be called "My office". In other words, if a physical address changes, the virtual address can be left the same as it will still be mapped to the correct physical address.

The Page Table: an array of Page Table Entries

P	M	Control	Frame#
"	"	"	"
"	"	"	"

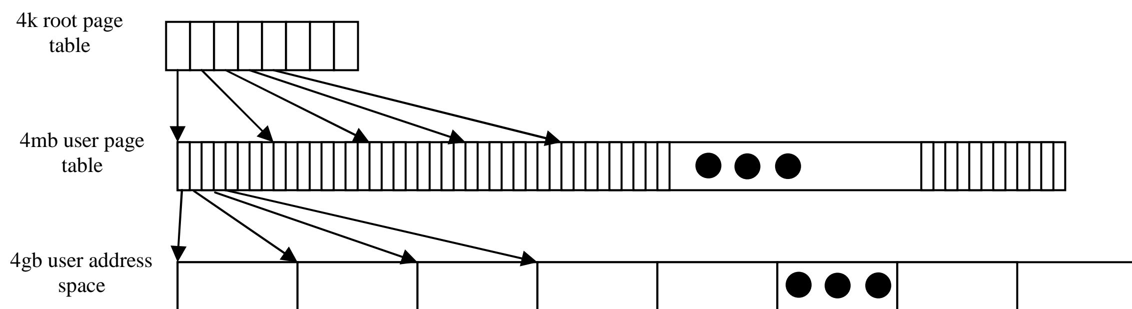
← Page Table Entry

- **P – present bit** – set to 1 if page is resident in main memory, 0 if it is stored in secondary memory. When this page is requested it can easily be seen if the page needs to be loaded into main memory or not.
- **M – modified bit** – set to 1 if page has been modified since it was last read from secondary memory. If the page needs to be transferred out of main memory, then it only needs to be written to secondary memory if it has been modified. Otherwise it is just overwritten. This cuts the use of slow I/O devices (disks).
- **Control** – often used to show page permissions: 'Can it be read?' or 'Can it be executed?' Good for distinguishing between code and data.

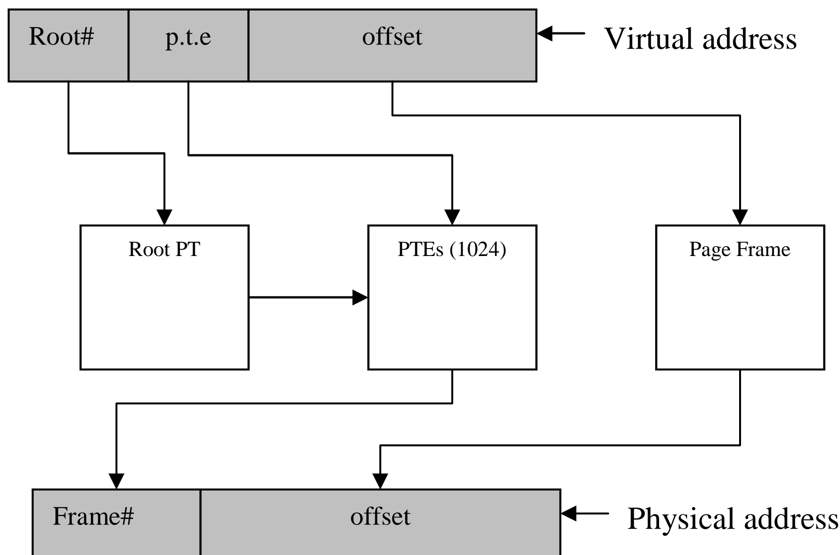
- **Frame #**: the first bits of the physical address for the page. With 32-bit addresses, this is typically 20 bits (12 bits remaining are for the offset into a 4K page).
- The page number is encoded in the array index. Thus, if you just stored page table entries using one array of PTE's, on a typical system this array would have 2^20 entries. Clearly we don't want to keep around something that big for most processes (that are much smaller than 4G in size), especially when most of the entries are going to be empty for a typical process. So, we use a tree, or…

Multi Level Page Tables

For 32-bit address spaces, two-level hierarchical page table such as the one illustrated below are generally sufficient:



To access memory locations, the page # of the virtual address is split in two: a root index (1024 entries, NULL or pointing to a page table), and a page table (1024 entries, each pointing to a page frame). Note that the root table and the 2nd-level page tables are all 4K, same size as our pages/page frames!



The process of resolving a logical to a physical address is expensive – index into the root table with the first 10 bits of the logical address to find the right secondary page table, index into the secondary page table with the next 10 bits to find the right frame, and then index into the actual frame to find the value in the physical address (which, by this process, is also the value present at the logical address). Phew! But this scheme is so useful, it has lead to hardware being designed to help increase the speed of this process.

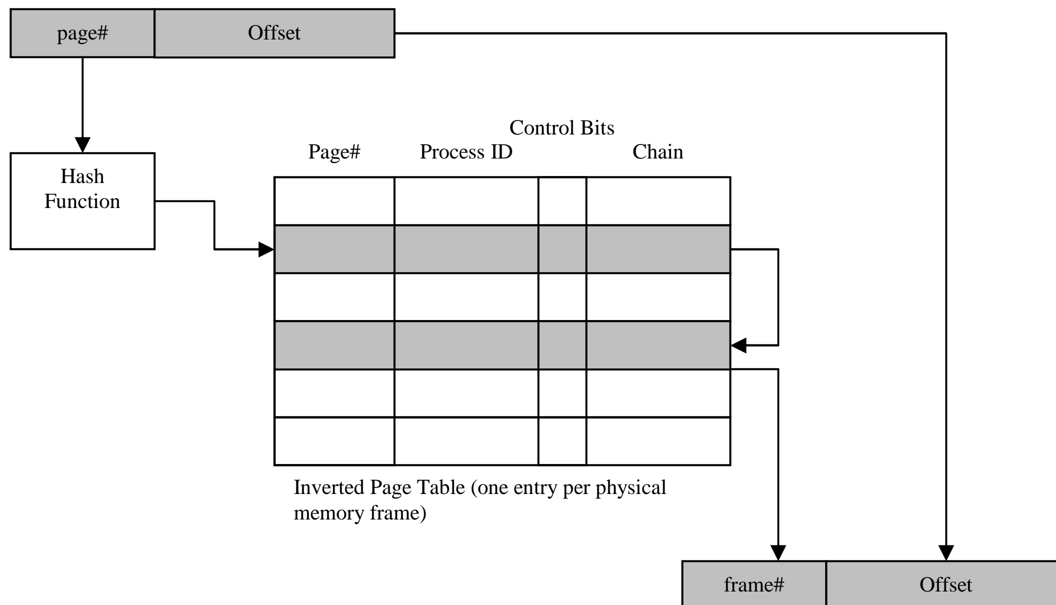
Note that if a process needs to be written to disk, only the root table needs to be kept in main memory, the secondary page tables can be written to disk and brought in as needed (into their own frames!).

With Linux, we actually use 3-level page tables so it can efficiently handle 64-bit address spaces. On 32-bit machines, the middle level is optimized away at compile time, so there is no runtime cost when it isn't needed.

Inverted Page Tables

Instead of having an entry in a table for each page in virtual memory, you have an entry in a table for each frame in physical memory. (This is good if you have a huge virtual address space, e.g. 64-bit – this is an alternative to a 3-level page table. Note how this saves a lookup step vs. a 3-level page table.)

To access the location, a hashing function is used. The diagram below shows the process.



The process ID field identifies the process that owns this page.

As more than one virtual address may map into the same hash table entry, the chain field provides a mechanism for managing the overflow that occurs.

Different inverted page table setups are supported by different hardware, therefore the OS must be fairly abstract in order to be portable across different architectures.

The TLB – Translation Lookaside Buffer

The TLB is a specialised cache for mapping virtual to physical memory locations. As most virtual memory systems will look up the physical address, then look up the data in that address, it is a lot faster to store at least some of the page table entries in a high-speed cache, thus halving the memory access time for those memory accesses.

The TLB will store the physical addresses of the most recently used pages. When a page is requested the TLB will be consulted first. If the page is there then this is a 'TLB hit' and the physical address is returned. If the page is not there then this is a 'TLB miss' and the page table is consulted to obtain the physical address.

The TLB is particularly important on most machines because the L1 cache (the cache closest to the registers, fastest, smallest, and nowadays on the CPU) is indexed by physical address – **not** logical address! So, without the TLB, we'd first have to do the logical->physical mapping (potentially going to main memory) before even checking the cache for the right code/data value. And we'd pay this penalty on every memory access! So, the TLB on most systems has to be consulted before the L1 cache. Thus, it has to be very, very fast.

It has to be so fast, it can't use hashing to find the right logical address prefix – instead, does "associative mapping" – it queries multiple entries at the same time to see which has the right logical->physical mapping. Sure, this consumes transistors, but this is in the critical path, so it is worth it.