Pagetables

by Mike Rieker

This discussion makes reference to Intel hardware, but is generally applicable to other processors that implement pagetables.

When an usermode application runs, it addesses memory using virtual addresses. So the value in a typical C pointer is a virtual address. The application has no idea where in physical memory it actually is, and for the vast majority of cases, it really doesn't need to.

This has been necessary so that you can, for example, have two users doing a 'copy' command at the same time. Each 'copy' command has its own set of physical memory pages that it is running in, but each 'copy' command is running at the same virtual address as the others. This gives us these advantages:

1. It can be linked to run at the one virtual address and not have to be adjusted for a different address depending on where in physical memory it happens to load.
2. It only has access to it's virtual address space and can't interfere with other 'copy' commands, or any other command for that matter.

So now the CPU is running that 'copy' command. And let's say the copy command is written in C, and a routine in there has a pointer variable. I mentioned that all addresses, such as those in a pointer are virtual addresses. Somehow the CPU has to translate all virtual addresses to an actual physical memory addresses so it can read or write the memory. It performs a translation of a virtual address to the corresponding physical address by way of the page table.

A pagetable is basically an array of pagetable entries that is indexed by a virtual address. Each entry of the pagetable contains items such as:

1. Whether or not the page is accessible at all
2. If so, the corresponding physical address
3. Allowable access modes (user, kernel, read, write)

Now I said the pagetable is an array that is indexed by a virtual address. Let's say our virtual addresses are 32 bits (as in an Intel pentium type chip). It would be impractical to have a pagetable that had 2^32 entries. So instead of using all 32 bits of the virtual address as an index, only the most significat 20 bits are used, the lower 12 bits are ignored. So that means there is one pagetable entry for 4K worth of virtual addresses. For example, virtual addresses 00003012 and 000039EC both reference the same pagetable entry, specifically index 3. It is said then that Pentiums have a page size of 4K, that is, one pagetable entry (PTE) maps 4K of virtual addresses onto 4K physical addresses. The low 12 bits of the virtual address that were discarded earlier are now used to index the specific byte in the 4K physical memory page.

Most programs do not need all 2^32 bits of possible virtual address space to execute. They would have to be 4 gigabytes in size to do that! Usually they are at most a few megabytes. This implies then that most of the pagetable contains entries indicate nothing is supposed to be at a given virtual address. If we needed to supply a pagetable entry (PTE) for each 4K page in that 4G address space, that is 1Meg entries. Each PTE takes a long, so you need 4Meg just for the pagetable. Well, the designers of the Pentium realized that would be silly, as back then, that's all the memory there was in the whole computer. So there is a pagetable page for the pagetable pages themselves, called a pagetable directory page. This means you only have to have a page of

pagetable entries for addresses you are actually using. So if a page of pagetable entries is all unused, you don't have to supply a pagetable page. Each entry of the pagetable directory page has a flag saying whether or not there is a corresponding pagetable page, and if so, the physical address of the corresponding pagetable page.

The virtual address now breaks down into:

- Bits 31..22 : index into the pagetable directory page
- Bits 21..12 : index into the pagetable page pointed to by the directory entry selected by bits 31..22
- Bits 11..00 : offset in the physical page pointed to by the pagetable entry selected by bits 21..12

The above is Intel specific, but other processors have similar stuff. Like Alphas have 8K pages, and IIRC, they have 3 levels of pagetable pages, instead of just 2 (because they have 64-bit virtual addresses). VAX's do it a completely different way, but the result is the same.

When we say that a particular process is active on a CPU, we are saying that that process' pagetables are being accessed by that CPU. In the Pentiums, this is done by loading the physical address of the pagetable directory in %CR3. So to switch processes on a Pentium, all one has to do is change the contents of %CR3 and you are now executing with different pagetables.

One more thing. The Intel hardware requires only that the pagetable directory and the pagetable pages exist in physical memory. It does not require that the pagetable stuff actually be mapped to virtual memory, it will run fine without that. However, it will make your coding miserable (and inefficient). So you *should* map the pagetables themselves to virtual addresses. In my OS, I put them at the high end. So the pagetable directory page for a given process is at that process' virtual address FFFFExxx. (I map FFFFFxxx to no-access to catch bad pointers at the high end). The pagetable pages are just below that and occupy virtual addresses FFCFE000 to FFFFDFFF (4 Meg). Just work out how to do it with pencil and paper, and in a few short hours of pure joy you'll have it.

Pagefaults

Now we have to fill in our pagetable with something useful. Let's say someone types in a 'copy' command or something like that. What happens (in regards to pagetables)?

1. The OS creates a pagetable directory page for the new process
2. The OS creates pagetable pages sufficient to point to the memory pages for the copy executable
3. The OS allocates memory pages for the copy executable
4. The OS reads the copy executable into those pages
5. Additional pagetable entries and memory pages are added for stack and heap memory

Now OS's usually don't do this for the whole image at once, they typically just set up structures to facilitate this. The reason for this is you may have a large image (such as MS Word), but you are only going to use a small part of it this time. So why load in all 10Meg when you are only going to be using say 1.5Meg? And if we only had 8Meg of ram available, it wouldn't even fit in there all at the same time!

So what an OS does is:

1. Create the pagetable directory page, with all entries marked as 'not-present', ie, there are no associated pagetable pages yet
2. Create a kernel struct that says where on disk and how big the executable is, and what virtual address it is to be loaded at
3. Create a kernel struct that says how big the stack is and what virtual address it starts at. Same for heap

memory.

Now when a CPU tries to execute instructions or data in that image, the CPU will not be able to access it, because the pagetable directory contains a null entry indicating the instructions or data are not in memory. So what the CPU does is generate a Page Fault exception. This passes the virtual address of the attempted access, and whether it was a read or a write, and whether it was by kernel mode or user mode, to a Page Fault Exception handler, which (usually) is a kernel mode routine that you are going to supply. A prototype for the handler might look something like:

```
int pagefault_exception_handler (void *virtual_address,
                                 int writing,
                                 int usermode)
```

Input:

virtual_address = address that the program was trying to access but couldn't because either:

1. the directory indicates there is no pagetable page present
2. the pagetable entry indicates no physical page is allocated
3. the pagetable entry indicates that the access attempt is illegal (eg, writing to a page that is marked read-only)

writing = 0 : the program is attempting to read the memory location
writing = 1 : the program is attempting to write to the memory location

usermode = 0 : the routine was executing in kernel mode
usermode = 1 : the routine was executing in user mode

Output:

pagefault_exception_handler = SUCCESS : the page was faulted in and the instruction should be retried
pagefault_exception_handler != SUCCESS : unable to fix it, signal a pagefault exception

So the pagefault handler routine looks at the kernel structs that were created to determine whether the access was legitimate, ie, is there supposed to be something there or not. If not, typically, it signals the exception (like a 'segment violation' in Linux). If there is supposed to be something there, that's when it allocates a pagetable page (if there isn't one there already), and allocates the page for the instructions or data.

Now it needs to look at the struct to see what goes into the page. If the struct says it is part of the executable, then the page is read in from disk. If it is a stack or heap page, it is filled with zeroes (or some other fill pattern).

Once the page is all set up, the pagefault handler returns back to the instruction that caused the fault. The CPU re-executes the instruction, and this time, it succeeds, because the pagetable entry is all filled in.

Some details

So the things the hardware requires us to tell it in a pagetable entry are:

- whether or not the page is in memory
- if so, what physical page it is in

- what kind of access is currently allowed (read, write, user, kernel)

Now, it turns out that your OS may find it handy to save some other info in there, too. Fortunately, the hardware usually leaves unused bits in the pagetable entries just for this very purpose. If it doesn't, you'll have to make an array that parallels the pagetable array to store these extra bits.

So here are additional things I store in the pagetable entries:

- software page state (valid and readable, valid and readable but dirty, valid and writable and dirty, page being read in from disk, page being written out to disk, page read/write from/to disk failed, page faulted out)
- what the requested protection is (read, write, user, kernel)

  Differs from current protection. Current protection should be 'no-access' when the page is faulted out, read-only when the page is being written out to disk, etc. Requested protection is what the page should be when it is fully realized.

Here are some low-level routines you will probably need for accessing pagetable entries:

read_pte : given a virtual address, determine if the corresponding pagetable entry exists in memory for the current process.

If not, return the virtual address of the missing pagetable entry (this will allow the caller to fault in the pagetable page then retry)

If it is in memory, return:
whether or not there is a memory page associated with it
if so, what physical page is there
current access modes
software page state
requested access modes

Now in my OS, my kernel's pagetable pages are always in memory, so I provide another routine, read_ptesys, that panics if the entry is not in memory, so I don't have to check the return value.

write_pte : given a virtual address, write the corresponding pagetable entry

For my OS, this routine could assume the pagetable page was in memory, as I always attempted to read the pagetable entry before writing it.

Translation Buffers

You may wonder if the CPU actually goes out and reads the pagetable entry for every memory access so it can translate the virtual address to the physical address. Well, they don't. CPU's typically have a special cache where they save the most recently used pagetable entries so they don't have to constantly go out and read pagetable entries.

These caches, typically, are somewhat 'manually operated'. The CPU will automatically load entries into the cache, but if you change a pagetable entry, it won't unload or update the corresponding cache entry. They do provide an instruction to unload it, though. So this just means your 'write_pte' subroutine has to also invoke this instruction after it sets up the new pagetable entry. For Intel chips, the magic instruction is INVLPG. The

cache also gets completely flushed when you change which pagetable you are using, ie, you write %CR3.

Multiprocessor stuff

Now if you are running on an multiprocessor system, things can get a little messy. Let's say two (or more) CPU's are executing different threads in the same process. Suppose one of the CPU's does something like an 'munmap' call that removes pages from the virtual address space and frees the physical pages for use by other users. So it executes INVLPG instructions for itself to remove those bad cache entries. It also has to tell any of the other CPU's that are mapped to this pagetable to remove their bad cache entries, too! Typically, this is done (in Intel processors), by setting a value in memory indicating which virtual address to invalidate, then issuing an NMI to the other processors, and then waiting for them to acknowledge that they have done the invalidate.

Now there is a little trick you can use to minimize NMI-ing everytime you write an pte. Suppose, again, we have two CPU's executing different threads in the same process. And one of them gets a pagefault because part of the executable needs reading in. That pagetable entry will be 'no-access' because there is nothing there in memory. So the faulting CPU will read the page in from disk and set up the new pagetable entry, and issue an INVLPG to itself. Now it doesn't have to NMI the other CPU's. If the other CPU attempts to access that same page, the worst that will happen is it will get a pagefault too. Then your pagefault handler will see that some other CPU faulted the page in and it's ok, so it just does an INVLPG on itself and returns. So it turns out the only time you need to do the NMI thing is when you are 'downgrading' the protection on a page, ie, you are disallowing some type of access that was previously allowed. Examples are changing a page from read/write to read-only, or marking an entry as non-existant.

So my OS actually has two pte write routines, pte_writecur which invalidates only the calling CPU's cache entry, and pte_writeall which invalidates all CPU's cache entries. I must call pte_writeall for downgrades in protection, pte_writecur can be called for upgrades only.

Mike's home page can be found at http://www.o3one.org/