Computer Organization and Assembly Languages Term Report

B93902043 蔡鎮宇

January 17, 2006

1 Preface

Since my first semester in university, I have taken on an interest in the inner workings of operating systems and computer architectures. During this semester, having taken this course and a course on operating systems, I have learned a great deal on things such as virtual memory and paging. However, the projects we've done in the operating system course concentrates more on the way things should work (an simulation written in `C++`), instead of how they work physically. In doing this report, I wish to extend my knowledge of how modern operating systems utilize hardware features imeplement virtual memory. This report will focus on x86 architecture proteced mode, segmentation and paging.

2 Brief History of the x86 Architecture

The x86 architecture started with the release of the 8086 processor. Since then, it has recieved many updates, while being backward compatible. Many have critisised this decision, pointing out that the x86 architecture is filled with relics from old time. Throughout the years, the x86 architecture has been upgraded many times: protected mode was added in 80286; 32-bit architecture and paging support in 80386; superscaler architecture in Pentium; SIMD instruction sets MMX, SSE, SSE2, SSE3.

3 Real Mode and Protected Mode Basics

In the early days of 8086 and 8088, the processors only supported *real mode*. Then, *16-bit protected mode* was provided by the 80286, but was rarely used because existing code would not run and the instability in switching to and from protected mode. Thus we will not discuss it here. All future references of `protected mode` will refer to `32-bit protected mode on 80386s or higher`. With the release of the new 32-bit 80386, and the addition of a paging unit, 32-bit proteced mode became popular now that virtual memory techniques were supported and the addressing space expanded.

3.1 Real Mode

In real mode, all access to memory is *segmented*. There are $4 \sim 6$ segment registers (CS, DS, SS, and ES; FS and GS were added later on). Logical memory address have the form `segment:offset`. Each segment has a size of 64 KB. Every type of access has a default segment register, while data accesses can explicitly specify one as well. The segment is shifted left by 4 bits, then added to the offset to give a 20-bit physical address. In real mode, the processor can address as much as 2^{20} bytes, or 1 MB. With this addressing scheme, two different logical addresses can point to the same physical address, which can become an obstacle when tracing code.

3.2 Protected Mode

In protected mode, many features provide support for multitasking, such as memory protection, paging, and hardware virtual memory support. Memory protection is provided by segmentation and paging units. Paging also provides the ability to remap memory access. Protected mode also allows access beyond 1 MB, up to 4 GB. Logical addresses are converted to linear addresses by the segmentation unit, then converted to physical addresses by the paging unit. The segment size in protected mode can be 1$\sim$1M bytes or 4K$\sim$4G bytes. Segment registers now contain an index to a descriptor table holding information on all segments.

4 Memory Addresses

In the x86 architecture, we have three kinds of memory addresses:

Logical Addresses are addresses used throughout programs to specify the address of instructions and operands. Each logical address consists of a *segment* and an *offset*. The offset denotes the distance from the start of the segment

Linear Addresses are 32-bit unsigned integers that can be used to address up to 4 GB, from $0 \sim 2^{32} - 1$.

Physical Addresses are 32-bit unsigned integers which correspond to electrical signals sent along the CPU's memory address bus. They are used to address memory cells on memory chips.

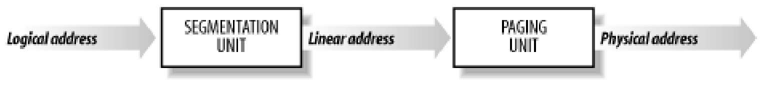

Figure 1: Logical Address Translation

5 Segmentation

5.1 Segmentation Registers

In protected mode, a *logical address* consists of a 16-bit *segment selector* and a 32-bit *offset*. Usually the CPU uses the segment registers to store the segment selectors for convenience. Some of the segment registers have special meanings:

cs The code segment register, which points to a segment containing instructions.

ss The stack segment register, which points to a segment with the stack.

ds The data segment register, which points to a segment with data.

The `cs` register also contains a 2-bit field that specifies the *Current Privilege Level (CPL)*. 0 denotes the highest privilege, while 3 denotes the lowest one.

5.2 Segment Descriptors

A segment is represented by a *segment descriptor*, which is stored in the *Global Descriptor Table (GDT)* or the *Local Descriptor Table (LDT)*. The segment selector is basicly an index to the descriptor table.

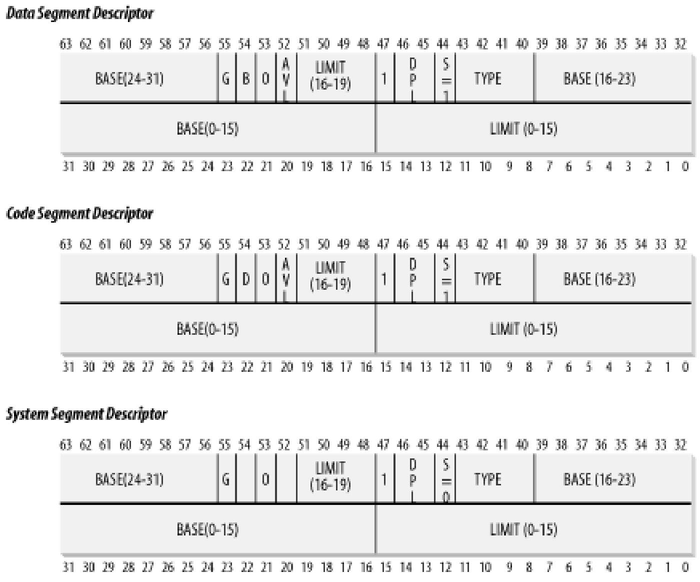

Figure 2: Segment Descriptor format

Normally, only one GDT is defined, while each process may have its own LDT if it needs to create additional segments. The address of the GDT is stored in the *GDTR* register, and LDT in the *LDTR* register.

A segment descriptor has the following fields:

- A 32-bit *Base* that contains the linear address of the beginning of the segment.
- A *G* Granularity flag. If it is set, the segment size is measured in units of 4KB; otherwise the unit is bytes.
- A 20-bit *Limit* field that denotes the size of the segment. A segment of size 0 is considered null.
- An *S* system flag, which indicates whether the segment is used to store system structures.
- A 4-bit *Type* field. The following list shows commonly used types.

  - *Code Segment Descriptor*
    Indicates the segment descriptor refers to a code segment. The `S` flag is set. It can be used in both the `GDT` and `LDT`.
  - *Data Segment Descriptor*
    Indicates the segment refered to is a data segment. Stack segments are implemented with this type as well. It can be used in both the `GDT` and `LDT`.
  - *Task State Segment Descriptor (TSSD)*
    Indicates a Task State Segment – a segment that is used to save the contents of the processors registers. It can only be used in the `GDT`. Ths `S` flag is set to 0.
  - *Local Descriptor Table Descriptor*
    Indicates a `LDT`. It can only be used in the `GDT`. The `S` flag is set to 0.

- A 2-bit *DPL (Descriptor Privilege Level)* field used to limit access. It represents the minimal CPL (refer to previous section) required to access the segment.
- A *Segment-Present* flag indicating whether the segment is currently in main memory.
- A flag named *D* or *B*, depending on the segment type.
- A reserved bit.
- An *AVL* flag that may be freely used.

The `Segment Selector`, which is 16-bits long, includes the following fields:

- A 13-bit *index* to the `GDT` or `LDT`
- A *TI (Table Identifier)* flag that specifies whether the descriptor is in the `GDT` or `LDT`.
- An 2-bit *RPL (Requester Privilege Level)* field, which is set to the `CPL` when the `cs` register was loaded.

Since a segment descriptor is 8 bytes long, the relative address in the descriptor table is obtained by shifting the index field of the selector by 3.

5.3 Fast Access

To speed up address translation, the processor has 4 ∼ 6 non-programmable registers to store the segment descriptors – one for each segment register. When a segment register is loaded, the corresponding segment descriptor is also loaded. All following address translations use the descriptor stored in the register; the `GDT` and `LDT` are only used when the segment registers are altered.

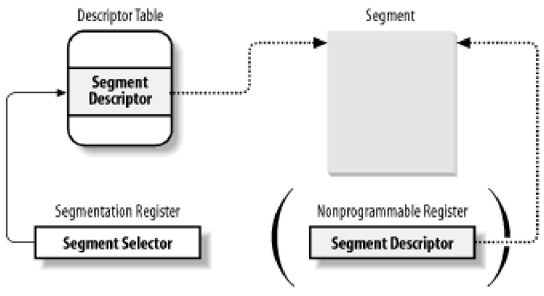

Figure 3: Segment Selector and Segment Descriptor

5.4 Segmentation Unit

The segmentation unit performs the following operations when it receives a logical address:

- Check the `TI` field to determine which descriptor table should be used.
- Computes the address of the segment descriptor from the `index` field and the `GDTR/LDTR`.
- Adds the offset of the logical address to the `Base` field of the segment descriptor. The result is the linear address.

Because of the non-programmable registers storing the segment descriptors, the first two operations are seldomly done.

6 Paging

The paging unit translate linear addresses into physical addresses. It checks the requested access against the access rights of the linear page. If the request

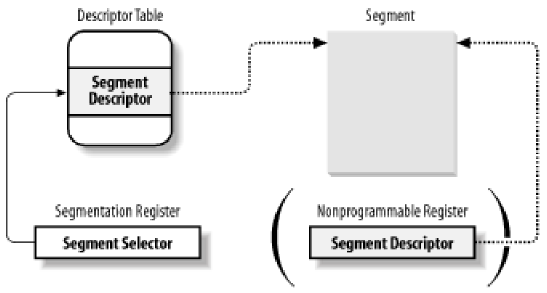

Figure 4: Translating a logical address

is not valid, it generates a *Page Fault exception*, which is processed by the operating system.

For efficiency, linear addresses are grouped in fixed-length units called *pages*; likewise, physical memory is divided into units that are the same size as `pages`, called *page frames* or *physical pages.*

The data structures storing the mapping between linear and physical addresses are called *page tables.* They are stored in main memory and must be initialized before enabling `paging`.

Paging is enabled by setting the *PG* flag in the control register *CR0.* When paging is not enabled, linear addresses are treated as physical addresses.

6.1 Normal Paging

The normal page size in the x86 architecture is 4KB. The `linear address` is split into 3 portions:

- Directory
  The 10 most significant bits represent the index in the `page directory`, which indicates a `page table`.
- Table
  The 10 intermediate bits represent the index in the `page table`, which indicates the `page frame`.
- Offset
  The 12 least significant bits represent the offset in the given `page frame`.

The translation is done in two steps, the first using the `page directory`, and the second using the `page table`.

The goal of this two-level paging scheme is to reduce the amount of RAM needed for per-process `page tables`, and to fit the `page table` in a `page frame`. Each page table entry is 4 bytes long. If one-level paging was used, then it would require 2^{20} entries, which sums up to 4MB of RAM, to represent the whole 4GB address space, even if only a small portion is used. With two-level paging, second level page tables are only required when the corresponding virtual addresses are used. The `page directory` is required for every process.

The physical address of the base of the `page directory` is set in the register *CR3*. The linear address's `directory` field determines the entry in the `page directory` that points to the appropriate `page table`. The `table` field then determines the entry in the given `page table` that points to correct `page frame`. Since the `offset` field is 12 bits long, it corresponds to a page size of 4KB.

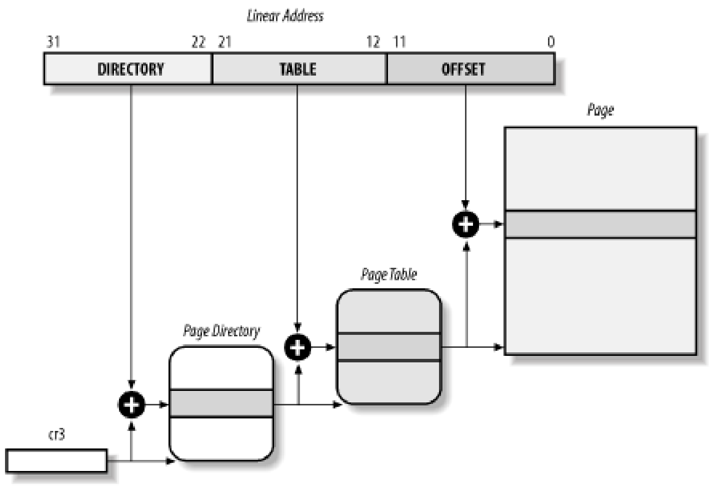

Figure 5: Paging by x86 processors

6.2 Page Tables

The entries of Page Directories and Page Tables have the same structure. Each entry consists of the following fields:

- *Present* flag
  If set, the referred-to `page` or `page table` is in main memory. If not,

the remaining bits are at the discretion of the operating system. If the `paging unit` encounters an entry with the `present` flag cleared, it stores the `linear address` in the *CR2* control register, and generates a `Page Fault` exception.

- Field containing the 20 most significant bits of a `page frame physical address`
  Since each page frame is 4KB in size, the 12 least significant bits of the physical address must be 0. The `page frame` contains a `page` of data or a`page table`.

- *Accessed* flag
  This flag is set each time the entry is looked up by the paging unit. It may be used by the operating system to track page frame usage, to implement optimal page replacement. The `paging unit` never resets the flag.

- *Dirty* flag
  This flag is set each time a write operation is done on the corresponding `page frame`. It may be used by the operating system to track page usage, just as the `Accessed` flag. It is also not reset by the `paging unit`.

- *Read/Write* flag
  Specifies the access rights of the `page` or `page table`.

- *User/Supervisor* flag
  Specifies the privilege level required to access the `page` or `page table`.

- *PCD* and *PWT* flags
  Controls the way the `page` or `page table` is handled by the caching hardware.

- *Page Size* flag
  Applies only to `page directory` entries. If set, the entry refers to a 4 MB jumbo page frame.

- *Global* flag
  Applies only to `page table` entries. This flag was introduced in the Pentium Pro to prevent frequently used pages from being flushed from the `TLB` cache. It works only if the *Page Global Enable (PGE)* flag in the *CR4* control register is set.

6.3 Extended Paging

Extended Paging is enabled by setting the *PSE* flag in the *CR4* control register. `Extended paging` is used to map large contiguous linear addresses onto corresponding physical ones. With this mechanism, RAM usage is reduced since there is no intermediate `page table`. It also saves `TLB` entries.

If the `page size` flag is set in a `page directory` entry, then the `paging unit` splits the linear address into two fields:

- Directory
  The most significant 10 bits.
- Offset
  The remaining 22 bits.

`Page Directory` entries for `extended paging` are the same as for `normal paging`, except that:

- The `page size` flag is set.
- Only the 10 most significant bits of the physical address field are used.

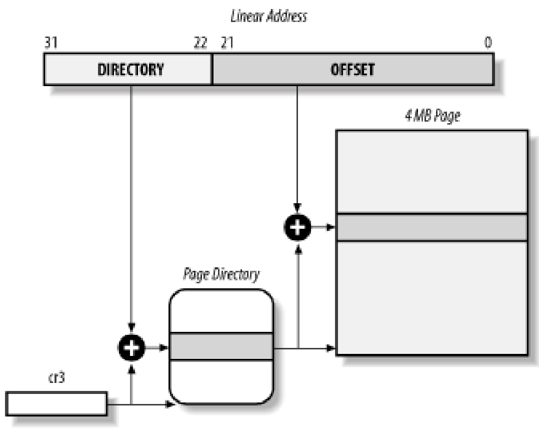

Figure 6: Extended Paging

6.4 Hardware Protection

The paging unit uses a different protection scheme from the segmentation unit. While a segment has four possible privilege levels, only two privilege levels are associated with `pages` and `page tables`, because privileges are controlled by the `User/Supervisor` flag mentioned earlier. When this flag is cleared, the page can be addressed only when the `CPL` is less than 3. When the flag is set, the page can always be addressed.

Furthermore, instead of the three types of access rights (Read, Write, and Execute) associated with segments, only two types of access rights (Read and Write) are associated with pages. If the Read/Write flag of a Page Directory or Page Table entry is equal to 0, the corresponding Page Table or page can only be read; otherwise it can be read and written.

Any illegal access will result in a `Page Fault` exception.

6.5 Hardware Cache Control

The *CD* flag of the *CR0* control register is used to enable or disable the cache circuitry. The *NW* flag, in the same register, specifies whether the `write-through`[1] or the `write-back`[2] strategy is used for the caches.

Another interesting feature of the `Pentium` cache is that it lets an operating system associate a different cache management policy with each page frame, which is controled by the two `page table` entry flags:

- `PCD` *(Page Cache Disable)*
  Specifies whether the cache must be enabled or disabled while accessing data included in the page frame.

- `PWT` *(Page Write-Through)*
  Specifies whether the write-back or the write-through strategy must be applied while writing data into the page frame.

6.6 Translation Lookaside Buffers (TLB)

Besides general-purpose caches, x86 processors include caches called *Translation Lookaside Buffers (TLB)* to speed up linear address translation. When a linear address is used for the first time, the corresponding physical address is computed through slow accesses to the `page tables` in RAM. The physical

[1]The cache controller always writes into both RAM and the cache

[2]Only the cache line is updated and the contents of the RAM are left unchanged. The RAM is updated later on.

address is then stored in a `TLB` entry so that further references to the same linear address are quickly translated.

When the `CR3` control register is modified, the hardware automatically invalidates all entries of the `TLB`.

7 Entering Protected Mode

Entering `protected mode` on a 386 or higher processor is quite simple. The following steps are required:

- Build the GDT
- Enable A20[3]
- Set the *PE (protection enable)* bit in the `CR0` control register
- Setup `segment registers` with valid `selectors`
- Flush the processor's instruction prefetch queue by executing a `JMP` instruction

8 Exiting Protected Mode

Exiting `protected mode` on a 386 or higher processor requires the following steps:

- Load the segment registers with real-mode compatible values
- Clear the *PE (protection enable)* bit in the `CR0` control register
- Execute a far jump
- Load the segment registers as needed by the real mode code
- Inhibit A20 from the address bus (gate A20 off)

[3]The 21st addressing pin of the processor, disabled by default for compatibility reasons

9 References

- Understanding the Linux Kernel, 2nd Edition By Daniel P. Bovet, Marco Cesati
- `http://my.execpc.com/~geezer/os/`
- `http://www.online.ee/~andre/i80386/`
- `http://x86.ddj.com/articles/pmbasics/tspec_a1_doc.htm`