Protected Mode Operation of Intel x86 Processors

Protected Virtual Address Mode, commonly called Protected Mode, is an advanced mode of operation of 80x86 processors beginning with the 80286. The 80286 was the last 16 bit x86 processor, and very few operating systems supported protected mode on the 286, and so most of this discussion is about protected mode as implemented in the 32 bit x86 processors, the 80386 and later.

The operation of protected mode on all 32 bit x86 processors is virtually identical, and so no distinction is made in general between 386, 486 and Pentiums.

There were three primary motivations for the development of protected mode.
1) Ability to address larger address space
2) Ability to implement protection of the operating system from misbehaving applications, and to protect applications from other misbehaving applications.
3) Ability to implement virtual memory systems.

The 8086 was a 16 bit processor that used 16 bit memory addressing. The 8086 supports 1MB of memory, and the segmentation mechanism was introduced to allow access to 1MB (20 bit address) of memory with 16 bit addresses. In protected mode, the segmentation model was expanded with additional features to support all three of the above goals.

In addition to the enhancements to the segmentation mechanism, the 32 bit x86 processors also support a memory paging mechanism. Memory paging provides a more flexible mechanism for managing memory and implementing a virtual memory system. The paging mechanism works in conjunction with the segmentation mechanism.

Virtual Memory

The purpose of a virtual memory system is to provide the illusion that a computer system has more memory than it actually does.

In a virtual memory system, the processor will have a logical address space that provides a large amount of logically accessible memory. This large logical address space will be mapped via some kind of memory management hardware and software onto a smaller physical memory address space.
The extra logical memory space that doesn’t correspond to actual physical memory will be stored on some kind of external storage (typically a disk drive) while it is not being accessed. The virtual memory manager will support some kind of demand loading mechanism so that when a program tries to access addresses that aren’t physically present, the correct data can be read from disk and loaded into physical memory so that the access can succeed.

In a virtual memory system, some portions of the logical memory space of the program will be present in physical memory, and the other portions will be kept in the backing store. When access to memory data that isn’t resident is attempted, the system needs to detect this and take steps to load the requested data from the backing store into physical memory. Generally, this will require freeing up some piece of physical memory by writing its contents to the backing store before the new data can be loaded.

A virtual memory system requires several kinds of hardware and software support.

- It is necessary to be able to map logical memory addresses onto physical addresses in such a way that the logical address remains constant as the physical address changes.
- There needs to be a way to indicate that a particular portion of the logical memory space is resident in the backing store and where in the backing store it is located.
- There needs to be a way for the execution of an instruction to be interrupted and restarted if it attempts to access a logical address that is not resident in physical memory.
  - This requires some kind of interrupt or trap to occur when instruction starts to execute.
- The trap handler in the operating system needs to be able to determine what address was being accessed so that the correct chunk of data can be loaded from the backing store into physical memory.
- The address mapping tables need to be updated to reflect the new configuration of memory, and then the failed instruction restarted, so that it appears to the running program as if the memory being accessed was present all along.

On a 32 bit, x86 processor, virtual memory can be implemented using either facilities in the segmentation mechanism or the paging mechanism. Virtual memory is usually implemented using the paging mechanism as it is more flexible, and the segmentation mechanism is used to implement the protection features.

**Segment Descriptors and Descriptor Tables**
The segment is the basic element used to control memory addressing, protections, and virtual memory (at least until paging is introduced).

A segment is a contiguous block of memory that is described by a segment descriptor. The attributes of a segment are defined by fields in the segment descriptor. These include the following:

- **Base Address**: The memory address of the beginning of the segment (32-bit)
- **Segment Limit**: The size of the segment (20-bit)
- **Segment Type**: The type of segment (various kinds of code or data or system segments) (4-bit)
- **Segment Privilege Level**: This is part of the protection mechanism and establishes the privilege level of the segment
- Status and control bits (2-bits)

See section 16.2 (pages 584-589) of the text for a detailed description of these fields.
From the perspective of memory addressing, the important fields of the descriptor are the segment type, base address and the segment limit fields. These define what kind of segment it is, where in memory the segment is located, and how large it is.

A segment descriptor table is an array of segment descriptors. In order for the processor to make use of a segment descriptor, it must be contained within a descriptor table, and the processor must know to use the descriptor table and where it is located. There are new processor registers that identify the descriptor tables in use.

There are three segment descriptor tables that can be in use at any one time. These are called the Global Descriptor Table, Local Descriptor Table, and Interrupt Descriptor Table.

- The Interrupt Descriptor Table, IDT, is only used by the interrupt system.
- The Global Descriptor Table, GDT, and Local Descriptor Table, LDT, are used to contain segment descriptors for code and data segments, and system descriptors for other system data structures.

In order to use a descriptor table, the table address must be loaded into the appropriate descriptor table register.

- GDTR contains the address of the current GDT,
- LDTR contains the address of the current LDT,
- IDTR contains the address of the current IDT.

Segment Selectors

Addresses in x86 processors always involve a segment portion and an offset portion. The segment portion of the address specifies which segment to access, and the offset portion specifies a location within the segment. In order to access a segment, some segment register must be loaded with a value that identifies the segment.

In real mode, a segment register will be loaded with the paragraph base address of the segment, which specifies a physical address.

In protected mode, the nature of the values loaded into a segment register changes.

Rather than the segment register containing a physical base address, the segment register contains a Segment Selector. This selector specifies which segment is to be accessed, and specifically, it selects a segment descriptor from a descriptor table.
The segment selector is divided into three fields as follows:

<table>
<thead>
<tr>
<th>Descriptor Index</th>
<th>TI</th>
<th>RPL</th>
</tr>
</thead>
</table>

The **RPL field** is a two bit field that specifies the **Requestor Privilege Level**. This will be discussed when privilege levels are discussed.

The **TI (or Table Indicator)** bit is a single bit field that specifies which descriptor table contains the segment descriptor. If this bit is:
- 1, the descriptor will be taken from the LDT,
- 0, the descriptor will be taken from the GDT.

The **Descriptor Index field** is a 13 bit field that indicates which descriptor within the selected descriptor table is to be used. It is simply an array index into the array of descriptors in the specified descriptor table.

The selector value is loaded into a segment register and then used to form addresses into memory. When a selector is loaded into the segment register, the processor performs some check to make sure that the selector specifies a valid segment descriptor.

Later, when the selector is used to specify the segment part of an address, the processor performs additional privilege level and protection checks to make sure that the memory access is valid.

An attempt to load an invalid selector into a segment register will generate a fault at the time the selector is loaded.
This is one key difference between real mode operation and protected mode operation.

- **In real mode**, segment values are physical base addresses, and any value can be loaded into a segment register.
- **In protected mode**, the selector value specifies a segment descriptor in a descriptor table, and only values which specify valid descriptors can be loaded into a segment register without generating a fault.

**Privilege Levels**

One of the requirements for most modern processors and operating systems is having a **mechanism for restricting access by user code to critical system facilities**. It is necessary to be able to

- protect the operating system from attack or damage by ill behaved user code,
- prevent direct access to hardware resources so that the operating system can control sharing of those resources between multiple user processes.

On the x86 processors, the **privilege level mechanism is used to restrict access to system resources** to those processes that are supposed to have access.

On x86 processors, there are **four privilege levels**. One way that these privilege levels are often viewed is as **four concentric rings, with each inner ring representing a higher level of privilege**. Each privilege level is called a ring, with Ring 3 being the least privileged and Ring 0, the most privileged.

- **Ring 3** is distinguished as being the user privilege level, and non-operating system code is run at ring 3.
- **Rings 2 through 0** are considered system privilege levels, with the most critical system code run at ring 0.

On x86 processors, a higher numerical value corresponds to lesser privilege, and a smaller numeric value greater privilege.
The privilege level at which code is executing determines what instructions can be executed and what data can be accessed.

Certain instructions in the instruction set are considered privileged instructions. Execution of these instructions would allow user code to modify the operation of the system in such a way as to bypass the protections mechanisms, and so execution of these instructions should be restricted so that only the operating system can execute them.

Ring 3 code is user code, and so cannot execute any privileged instructions.

Ring 1 and 2 code is restricted from executing privileged instructions, but can execute i/o instructions under certain circumstances.

Ring 0 code is system code, and can execute any instruction, both privileged and non-privileged.

Data access also requires control so that the operating system can prevent user code from accessing system data.

When accessing data, code at a particular privilege level is allowed to access data at the same privilege level or below.

- Therefore, ring 3 code can only access ring 3 data. Any attempt by ring 3 code to access ring 2 or higher data will generate a fault.
- Ring 2 code can access ring 3 or ring 2 data.
- Ring 0 code can access any data in the system.

Since it is not possible to access memory without using a segment selector to form the address, the operating system can have complete control over access to data by controlling what segment descriptors are in the descriptor tables and the privilege levels associated with them.

There are number of different ways that the privilege level is specified as follows:

DPL – Descriptor privilege level. This is a field in the segment descriptor, and defines the privilege level of the segment. If the DPL field of a segment descriptor contains 3, for example, that segment is a ring 3 segment, and can be accessed by any code in the system that has a selector to that segment. If the DPL field in a descriptor contains 0, that is a ring 0 segment, and can only be accessed by ring 0 code.

CPL – Current privilege level. This is the privilege level at which the processor is currently executing. This is determined by the DPL of the current code segment, i.e. the DPL of the segment whose selector is currently in
CS. If the DPL of the current code segment is 0, then CPL is 0, the processor is executing at ring 0, and has ring 0 privileges for access to instructions and data. If the DPL of the current code segment is 3, then the CPL is 3, the processor is executing at ring 3 and has ring 3 privileges for access to the system.

**RPL – Requestor privilege level.** This is the privilege level at which an access to memory is being requested. The requested privilege level is specified by the RPL field (the low two bits) in a segment selector. A memory access will be attempted at the privilege level specified by the RPL field of the selector used in the access. The RPL field will be compared against the CPL value, and then degraded if necessary so that the request occurs at the CPL or lower. This degraded RPL is then compared against the DPL of the segment being accessed. If the DPL of the segment is at or below the RPL of the request, then the access will be allowed. If the adjusted RPL is above the DPL a fault will be generated.
IOPL – I/O privilege level. This is the privilege level at which execution of i/o instructions is allowed. Some instructions are privileged, and can only be executed by ring 0 code. I/o instructions are conditionally privileged and can be executed by code running at the IOPL or higher. This allows the operating system designer to decide what privilege level of access is required to interact with i/o devices.

For example, Windows 98 sets IOPL to 3 and allows any code running on the system to perform direct port i/o. Window 2000 sets IOPL at 0, and only allows the most trusted operating system code to perform i/o.