80386 Protection Mode

1 Introduction and Initial Discussion

For Teacher: Let us start with an analogy here: The projector in the classroom should be protected, and only authorized users can turn on the projector in my class. I will perform the access control (because I have the remote control). Whoever needs to turn on the projector during my class time needs to send me a request, and I will check whether you are on the authorized user list. If yes, I will use the remote, push the ON button, and send a signal to the projector; if not, the request will be denited.

- Can I actually prevent unauthorized users from turnning on the projector?
- What prevents them from bypassing me and directly send the signal to the projector (e.g. recording the signal I sent to the projector)?

For Teacher: We can then proceed to ask students what prevents normal users from modifying the `/etc/passwd` file.

- Students may say "access control" in the operating system.
- Why should we go through the access "controller"?
- Why can't we directly jump to the functions in the device driver, and access the disk through the device driver?
- Why can't we write our own code (i.e. device driver) to directly access the raw disk?

Question 1 (Execution Emulation): Assume that to write to `/etc/passwd` file, the CPU instructions (Machine codes) that get executed are c_1, c_2, . . ., c_n. And also assume that the instructions related to access control is a_1, . . ., a_s. Now let's construct a new program $p' = c_1$, . . ., c_n - a_1, ..., a_s, and let run it directly on CPU, can we succeed in writing to `/etc/passwd` file?

Answer: In 8086, you can do this. In 80386, you cannot!

Question 2 (Code Access): Assume that we know the address of the code for system calls `write()`, which can write data to disks. There are two ways to call it:

1. Go through the system-call approach, which is subject to access control.
2. Directly jump to that code.

We know the first approach works, but can the second choice succeed? If not, what prevents a program from jumping to that code?

Answer: the hardware access control disallows it. There is a security policy to prevent the above direct jumping from happening, and the policy is enforced by hardware. We are interested in how such access control policy is defined and how hardware enforces the policy.

Question 3 (Data Access): We know that when we use `open()` to open a file, a file descriptor will be returned. This file descriptor is the index to the "capability" that is stored in the kernel. Assume that we know the address of this capability. What prevents us from directly modify the capability, and thus giving us additional permissions?

Answer: the access is disallowed. There is a security policy to prevent the above direct access of the kernal memory from user space, and the policy is enforced by hardware at each memory access. How does such an access control work?

Discussion: From the above questions and their answers, it seems certain kind of access control is protecting the systems. If you get chance to design such a protection scheme, how would you design an access control like this?

- Four components of a security policy: subject, object, action, and rule.
- Action: instructions.
- Objects: things that need to be protected.
  - Memory: at what granularity, byte, word, or block? What are the disadvantages and advantages of your choices?
  - Registers
  - I/O Devices
- What can be used as subject?
  - Can we use user ID or process ID as subjects? No we cannot use things that are defined in an operating system, because this access control is not part of an OS, it is underneath an OS. Processes and users are meaningful in an OS, but the underlying hardware does not know what those are.
  - In other words, how to give each instruction an identity?
- How to design the rules (or policies)?
  - How to represent the policies?
  - Where to store the policies?
  - When to enforce the policies?
  - Access matrix: high cost, inflexible, etc.
- Mandatory versus Discretionary Access Control
  - If MAC is used, system-wise mandatory access control polices are enforced.
  - If DAC is used, the owner of an object can set up security polices.

  - *80386 Protection Mode chose MAC:* DAC puts the security of a system at user's hands, because in DAC, users define their own discretinary access control policies for the objects that they own. If users make a mistake, the system can become flawed. MAC does not put the security at users' hands; instead, it defines a global policy that are enforced in the entire system. The policy are usually defined by authorities (e.g. super users). With MAC, even if users make a mistake (either intentionally or accidentially), the system-level security policy will always be enforced due to MAC. Such property of MAC is so appealing that many modern operating systems start to have MAC. For example, `SELinux` and `Windows Vista` all have built-in mandatory access control mechanisms.

    80386 picks MAC so the policies can only be set by the authorities, instead of by the owners of objects. An example of authorities is the operating system that runs on 80386, i.e. once the operating system set the policies, 80386 will enforce those policies.

- In MAC, security policies are usually based on groups of subjects/objects, instead of on individual subjects/objects. Grouping reduces the number of distinct subjects/objects, and thus making management much easier. Grouping in MAC is done by labeling, i.e. assigning labels to subjects and objects; access control policies are defined based on these labels. One may choose many labels to achieve finer granularity, or choose few labels to simplify management and access control logic.

  If you were to design a MAC for CPU, what do you plan to use for labeling, how many labels do you plan to use, and where do you store the labels?

2 The Ring Architecture and Segments

- History
  - Late 70's: 8086, Real Mode and has no protection.
  - 1982: 80286, Real Mode and 16b Protected Mode.
  - 1985: 80386, Real Mode and 32b Protected Mode.
- The Ring architecture: the labels used by MAC.
  - 80386 has four rings. Each ring is associated with different privileges. Ring 0 is the most privileged ring, and the OS kernel usually resides within this layer.
  - Each object and subject is associated with a label, called ring. This label is used as the subject in access control policies.
  - Whether a subject can access an object is decided by the mandatory access control policy that are implemented in the hardware.

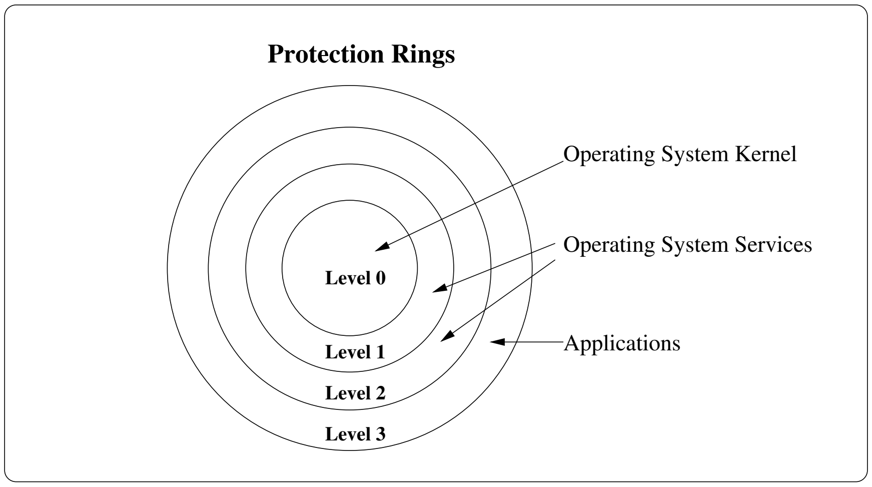


Figure: Rings

- **Memory protection across ring boundaries**: once we divide the memory into several rings, we can define security policies based on rings. For example, we can prevent code in ring 3 from accessing data in ring 0, etc. The question is that, when conducting access control, *how CPU learns the ring labels of a subject and an object.*
  - When CPU enforces the access control policies, it must know the ring label of both the subject and object in an efficient way.
  - CPL: Current Privilege Level, the label on subjects.
    - ∗ CPL is stored in a register (Bits 0 and 1 of the CS and SS segment registers).
    - ∗ CPL represents the privilege level of the currently executing program or procedure.
    - ∗ Normally, the CPL is equal to the privilege level of the code segment from which instructions are being fetched (there is one exception, and we will talk about later when we talk about conforming code segments).
    - ∗ The processor changes the CPL when program control is transferred to a code segment with a different privilege level.
  - DPL: Descriptor Privilege Level, the label on objects.
    - ∗ DPL is the privilege level of an object. When the currently executing code segment attempts to access an object, the DPL is compared with CPL.
    - ∗ Where should DPL be stored?
      - · Discussion: stored in each byte? Stored for each block (at the beginning of a block)? or somewhere else?

- **Memory protection within the same ring**: Rings can achieve memory protection across ring boundaries, but they cannot memory protection within the same ring. For example, when we develop an operating system for 80386, we would like user processes to run at ring 3, but we do not want one process to access another process's memory (all within ring 3). Rings cannot achieve this kind of protection (memory isolation). We need another access control mechanism for this protection.
  - Let us divide memory into segments. Each process can take one or more segments. Whenever a process tries to access a memory, access control should be enforced to achieve memory isolation.
  - Discussion: What access control model do we use? ACL or Capability?

- *ACL Approach*: we associate each segment with an access control list. Each memory access will go through this list. This is too time consuming, because the list might be long. The processor cannot afford to go through a long list for each memory access.
- *Capability Approach*: each process is assigned a list of capability, each corresponding to one of its segments. There are two important issues in capability-based access control. First, where should the capabilities be stored? They cannot be forged by users. Privileged rings are good places for storing capabilities. Second, there are two common ways to implement capability-based access control:
  * *Capability List*: the code does not need to explicitly show its capabilities when access a memory; instead, the processor searches the capability list of the process to find the one that is appropriate, if any. This approach has the same problem as the ACL approach: list might be too long.
  * *Index of Capabilities*: when a code tries to access a memory, it should present a “ticket”, which is the index of the actual capability stored in a privileged ring. This way, the processor only needs to check this specific capability. The performance is much better than the capbility list approach. This is similar to how the file descriptor is implemented.
- 80386 chooses the capability as its access control model to achieve memory isolation; it uses the index approach.

• Logical and Linear Address

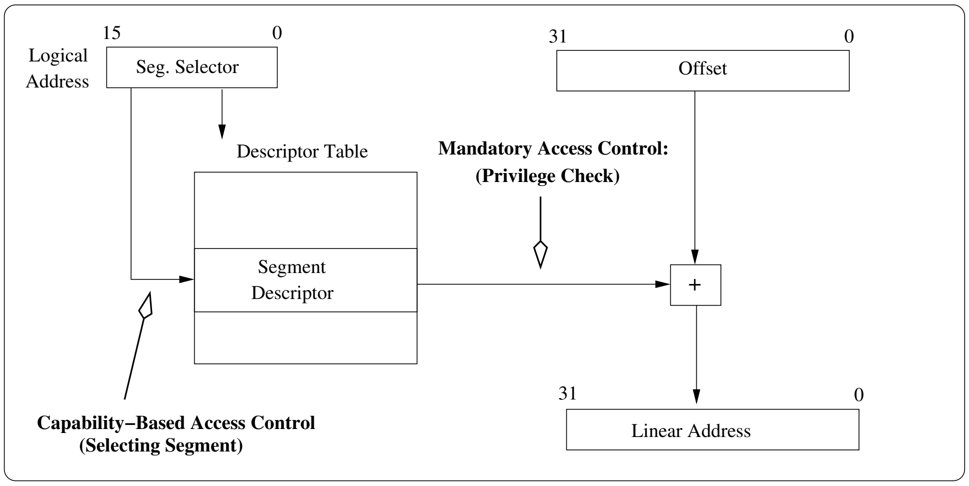
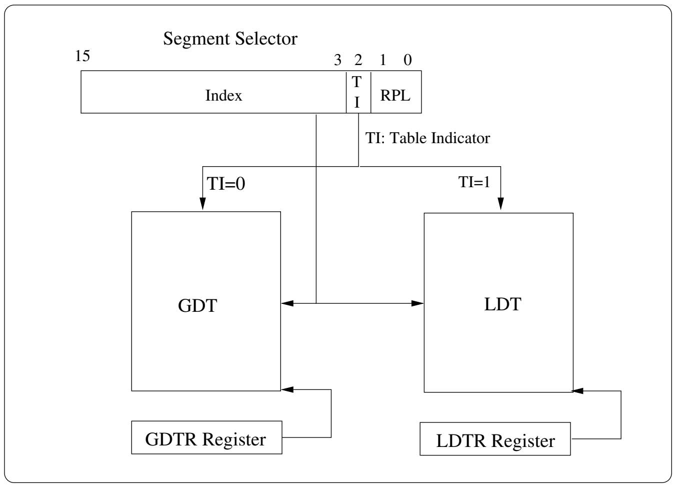

Logical Address
15
0
Seg. Selector
31
0
Offset
Descriptor Table
Mandatory Access Control: (Privilege Check)
Segment Descriptor
+
31
0
Linear Address
Capability−Based Access Control (Selecting Segment)

Figure: Logincal Address to Linear Address Translation

- Logical address: consist of *segment selector* and offset. The processor converts the logical address to linear address using the segment descriptor indicated by the segment selector.
- Linear address: when the paging is disabled, the linear address is actually the physical address; when the paging is enabled, the linear address is converted to physical address through paging mechanisms.
- Segment selectors are provided by segment registers: For example, in the following instruction, the segment selector is provided by the register `DS`, and the offset is `80`:
  `MOV %DS:[80], %EAX.`
- Two access control mechanisms are used here:

  * Capability-based Access Control: the segment selector and segment descriptor are actually the capability concept. The segment descriptor is the capability, while the segment selector is the index to the descriptor. Segment selectors are accessible to user programs, but not segment descriptors.
  * Mandatory Access Control: even if a process has a capability, its access right is further restricted by another level of access control that is based on MAC. This level of access control ensures that ring based access policies are enfored.
  * Note: one might wonder whether the second-level of access control is redundant; if the access is not allowed, why bother to create a capability (descriptor) for a process at the first place? There are two reasons for that: (1) In capability-based access control, it is desirable if subjects can turn on/off their capabilities to reduce the risk. 80386 uses a mechanism called RPL (Request Privilege Level) to temporarily turn on/off the capabilities while executing some instructions; RPL relies on the mandatory access control mechanism to work (we will talk about RPL later). (2) 80386 also allow each task to use a Global Descriptor Table (GDT), which contains capabilities shared by all processes. All tasks can access these capabilities, but a capability is effective depends on the subject's CPL and the object's DPL (i.e., depending on the mandatory access control).

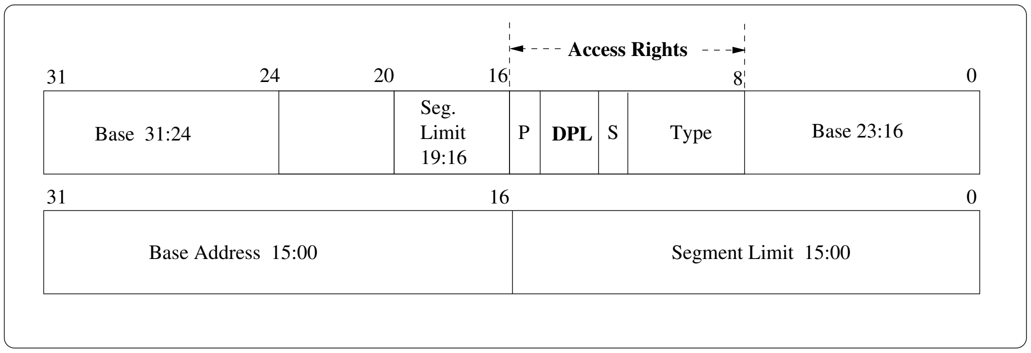
- Segment Selector



Segment Selector

  - **TI**: Table Indictor. Indicate whether GDT or LDT is used.
  - **Index**: The processor multiplies the index by 8 (the number of bytes in a segment descriptor), and add the result to the base address of the GDT or LDT based on the TI value (the base addresses are stored in the GDTR or LDTR register, respectively).
    * **GDT**: Global Descriptor Table. Each system must have one GDT defined.
    * **LDT**: Local Descriptor Table. One or more LDT can be defined. For example, an LDT can be defined for each task being run, or some or all tasks can share the same LDT.
  - **RPL**: Request Privilege Level. Specifies the privilege level of the selector. We will explain this later.

- Segment Descriptor



Segment Descriptor

- Base (32 bits): the base address of the segment.
- Segment Limit (20 bits): the size of the segment. The processor will ensure that the offset of the address does not go beyond the segment limit.
- Type (4 bits): specify the type of segment. The processor will enforce type rules. For example, no instruction may write into a data segment if it is not writable, no instruction may read an executable segment unless the readable flag is set, etc.
  - Data Type: Read-Only, Read/Write, etc.
  - Code Type: Execute-Only, Execute/Read
- DPL (2 bits): Descriptor Privilege Level. It specifies the ring level of the segment. DPL is used in access control.

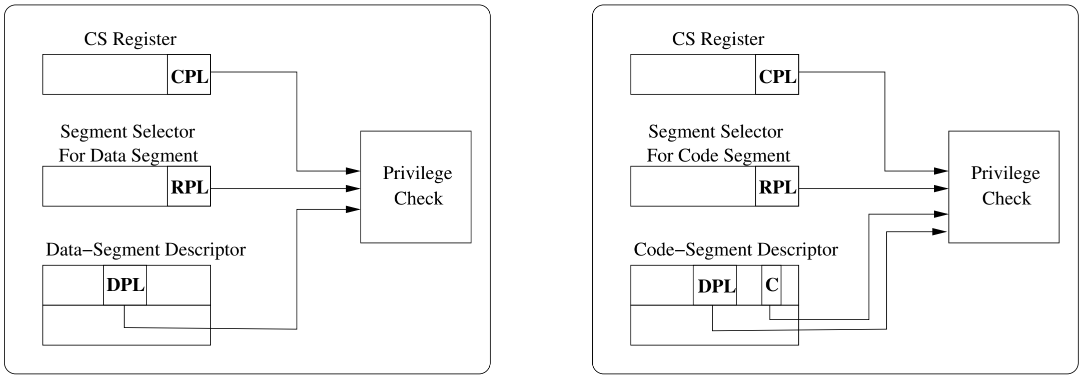
- Segment Registers
  - Due to the address translation step, accessing data or code in memory involves two memory access, one for retrieving segment descriptor from the descriptor table, and the other for accessing the actual memory. To avoid consulting a descriptor table for each memory acess, 80386 caches information from descriptors in segment registers.
  - A segment register has a "visible" part and a "hidden" part.
    - "Visible" part: segment selector.
    - "Hidden" part: descriptor cache; it caches the descriptor indicated by the segment selector, including base address, limit, and access information. This cached information allows the processor to translate addresses without taking extra bus cycles.
  - Segment registers in 80386: CS (code segment), DS (data), SS (stack), ES, FS, and GS. By default, for a code address, the processor uses the segment selector contained in CS, and therefore fetch the code from the code segment. For an data address, the processor by default uses the segment selector contained in DS, and for a stack address, the processor uses the segment selector contained in SS. If one wants to use other segment registers, they can use them as a prefix: e.g. `MOV EAX, FS:[0]`.
  - For a program to access a segment, the segment selector for the segment must have been loaded in one of the segment registers. The operations that load these registers are normal program instructions; they are of two classes:
    - Direct load instructions: e.g. `MOV`, `POP`, `LDS`, `LSS`, `LGS`, `LFS`.
    - Implied load instructions: e.g. far `CALL` and `JMP`. These instructions implicitly reference the CS register, and load it with a new value.

When segment registers are modified, the processor *automatically* fetches the base address, limit, type, and other information from a descriptor table and loads them into the "hidden" part of the segment register.

– Loading a segment register under 80386 Protected Mode results in special checks and actions, to make sure the access control policies are satisfied. We will talk about the policies later.

3 The Mandatory Access Control on Data and Code Access

- Privilege Check for Data Access (see Figure)
  - We temporarily ignore RPL.
  - Policy: CPL $\leq$ DPL of code segment.
  - A subject can only access data objects with the same or lower privilege levels.



(a) Privilege Check for Data Access (b) Privilege Check for Control Transfer Without Using a Gate

Figure: Access Control

- RPL: Request Privilege Level.
  - *Potential Risk*: At ring 0, code can access data at any ring level. This poses a risk when the code (say A) is invoked by some other code (say B) in a less privileged ring, and B passes a pointer to A. Normally, the pointer refers to a memory space that belongs to B (and of course A can also access). However, if B is malicious, B can pass a pointer of a memory that does not belong to B (B does not have privileges to access the memory). Because A is a privileged code, access control cannot prevent A from accessing the memory. This way, B can use A to corrupt the targeted memory in a privileged space.
  - **Principle of Least Privilege**: in the above case, it is really unnecessary to run A with the ring-0 privilege when accessing the pointed memory passed by B. According to the principle of least privilege, A should drop its privilege to B's ring level when accessing the memory.
  - How does RPL works:
    - ∗ Assume that A is in ring 0 and B is in ring 3, and the memory address's selector is S.
    - ∗ The last two bit of a selector is used for **RPL**. It means that *when accessing this memory, the code's privilege is droped to the RPL level*. Therefore, if S's RPL=3, when A tries to access the memory in ring 0 (i.e. DPL=0), the access will be denied. If S's RPL is not dropped to 3 (instead it is set to 0), the access will succeed because A's CPL is 0.

    - In other words, $\max(RPL, CPL)$ is actually used for access control.
    - RPL is usually less than or equal to CPL.
    - If RPL = 0, RPL will have no effect.
  - *Policy of access control*: $\max$(CPL, RPL) $\leq$ DPL of data segment.

- Privilege Check for Control Transfer without Using a Gate
  - Policy:
    - For non-comforming segment: transfer is allowed when CPL $=$ DPL.
    - For comforming segment: trasfer is allowed when CPL $\geq$ DPL.
    - RPL does not have much effect here.
  - Why can't we access code with a higher DPL (i.e., lower privilege)?
    - Possible reason 1: It is easy to jump (lower the CPL) to the code with higher DPL, but it is difficult to return back, because on returning, we jump from a lower privileged ring to a higher privileged ring. This violates the mandatory access control policy.
    - Possible reason 2: Another reason is the data access. If a code A jumps to another code B at a lower privilege level, B cannot access A's data because the data are most likely in A's ring level.
    - Possible reason 3: Is there really a need to allow jumping from a higher privilege to a lower privilege?
  - Why can't we jump to code with a lower DPL (i.e., higher privilege)?
    - For security reasons, we cannot do this.
    - Is this type of jump necessary? Yes, we definitely need this. For the device driver code is usually in a privileged ring. User-level program should be able to jump to the device driver code somehow.
    - How can we achieve jumping to lower DPL? *Gates* are designed for this purpose. We will talk about gates later.

- The conforming bit
  - Permits sharing of procedures, so they can be called from various privilege levels.
  - Usually for math library and exception handlers.
  - If you have a procedure that you want everybody to be able to call it, where do you put it? in which ring?
    - Ring 3: ring 0,1,2 cannot call it.
    - Ring 0: ring 1,2,3 cannot call it.
    - Ring 0 and call gate: you need to build a call gate for each library call.
  - The conforming segment mechanism permits sharing of procedures that may be called from various privilege levels but should execute at the privilege level of the calling procedure. When control is transferred to a conforming segment, the CPL does not change.

4 Call Gates

- *Supporting system calls*:
  In an operating systems, all the privileged operations are carried out in the privileged rings, such as modifing kernal data structure, interacting with devices, etc. OS does not allow user programs to invoke them in an arbitrary way, such as jumping to the middle of a privileged operations. Instead, OS provides a number of interfaces to users programs, which can only invoke those provileged operations via the interfaces. These interfaces are often called *system calls*. Invoking system calls is quite different from invoking a normal function. In the latter case, the call is within the same ring; however in the former case, the call is from a less privileged ring to a prvileged ring. 80386's ring protection does not allow a direct jump like this. Some special mechanism must be provided to allow the control transfer from a less privileged ring to a privileged ring.

- How to invoke system calls?
  - **Call Gate**: Call gates allow a program to directly call system calls. However, since system calls are often in a privileged ring, calling them directly is not allowed because of the ring protection. The 80386 protection mode uses a call-gate concept to allow this kind of transfer. Call gates enable programs in a lower privileged ring to jump to designated places in a higher privileged ring.
  - **Software Interrupt** or **Trap**: In many operating systems, such as `Linux` and `Minix`, programs use `int 0x80` to trap to the kernel. Namely, when a program wants to call a system call, it saves the system call number in the EAX register, and then execute `int 0x80` to raise a software interrupt. Interrupts transfer control to the kernel, so the kernel can execute the intended system call based on the number stored in EAX. This approach is quite popular in OS designs.
  - **SYSENTER/SYSEXIT**: The Intel Pentium II processor introduces another new facility for faster system call invocation. The facility uses the instruction `SYSENTER` to enter the system call entry point at ring 0; it uses `SYSEXIT` to return back. Starting with version 2.5, `Linux` kernel introduced started taking advantage of this new system call facility.

- The Call Gate concept.

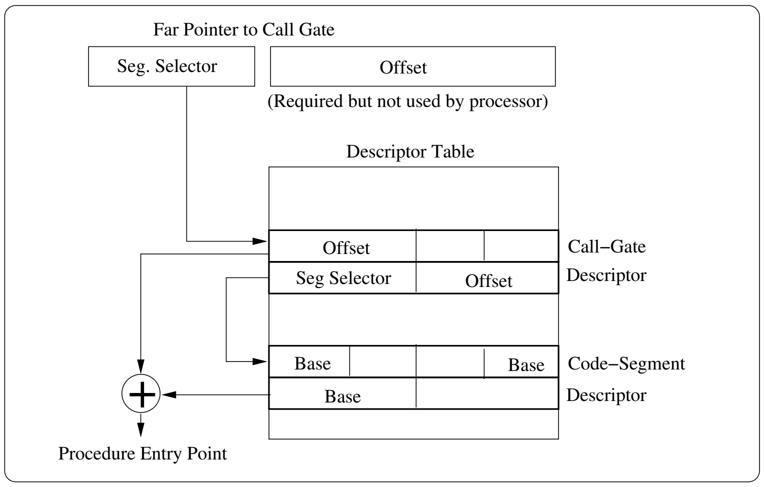


Figure: Call Gate

- The idea of *Call Gate*: 80386 does allow a program to jump to a more privilege ring, but a program cannot jump to an arbitrary place, it must go through Call Gates, which basically define the entry points for the privileged code. Corresponding security checks will be conducted at those entry points to decide whether the invoking code has sufficient right. These security checks are enforced by operating systems.
- Like segment descriptors, call-gate entries (call-gate descriptors) are also stored in the GDT (or LDT) tables. Gates define an entry point of a procedure.
- Call-Gate Descriptor contains the following information:

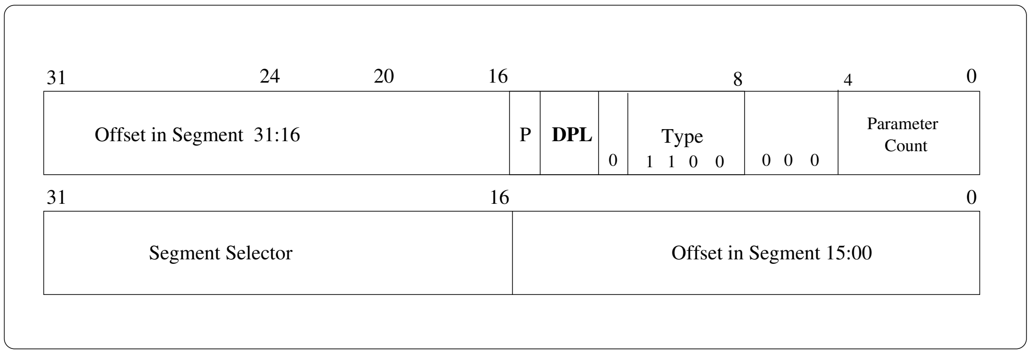


Figure: Gate Descriptor

  * Code segment to be accessed (segment selector)
  * Entry point for a procedure in the specified code segment (offset in segment)
  * Privilege level required for a caller trying to access the procedure (DPL)
  * Parameter Count: if a stack switch occurs, it specifies the number of optional parameters to be copied between stacks.
  * etc.
- How to use call gates?
  * Call xxxxxx or JMP xxxxxx
  * xxxxxx specifies the call gate entry in the GDT (or LDT) table
  * From the table, the entry point of the procedure will be obtained.
  * DPL of the gate descriptor allows the CPU to decide wither the invocator can enter the gate.

- Access Control Policy for Call Gates
  - CPL $\leq$ DPL of the call gate.
  - For CALL: DPL of the code segment $\leq$ CPL (only calls to the more privileged code segment are allowed).
  - For JMP: DPL of the code segment = CPL.
  - Q: why can't we CALL a less privileged code segment using Gates? Still returning will be a problem, because returning will be from the less privileged code to the more privileged code, and it violates the mandatory access control.

- Returning from a Called Procedure
  - The RET instruction can be used to perform a near return, a far return at the same privilege level, and a far return to a different privilege level

  - A far return that requires a privilege-level change is only allowed when returning to a less privileged level

5 Protecting Registors and I/O

- Protecting descriptor tables (via protecting their registers)
  - GDT, LDT, IDT are very important. They contain the followings.
    * Call gates
    * Code Segment Descriptors
    * Data Segment Descriptors
  - These tables should be in a protected memory
  - The GDTR, LDTR, and IDTR registers (they store the locations of these tables) can only be set by privileged code, i.e., the following instructions can only be executed in ring 0:
    * LGDT - Load GDT Register
    * LLDT - Load LDT Register
    * LIDT - Load IDT Register
  - **Questions:** What is the problem if these registers are not protected?
- I/O Protection
  - Introduction Question: everything can be boiled down to I/O operations. Is Direct I/O from ring 3 possible?
  - How to prevent any arbitrary code from conducting I/O operations?
  - Instructions: IN, INS, OUT, OUTS
  - IOPL (IO Privilege Level) is stored in EFLAGS. It shows the I/O privilege level of the current program or task.
  - The CPL (Current Privilege Level) of the task or program must be $\leq$ IOPL in order for the task or program to access I/O ports.
  - The IOPL can be changed using POPF only when the current privilege level is Ring 0, i.e., only by kernel code.
  - This way, the OS kernel decides which ring can run I/O operations. Usually, OS sets IOPL=0, meaning that I/O operations can only be conducted by the code in the kernel.
  - **Questions:** What is the problem if I/O operations are not protected?

6 Page-Level Protection

- Paging mechanism (the following figure is for 4-KByte pages)

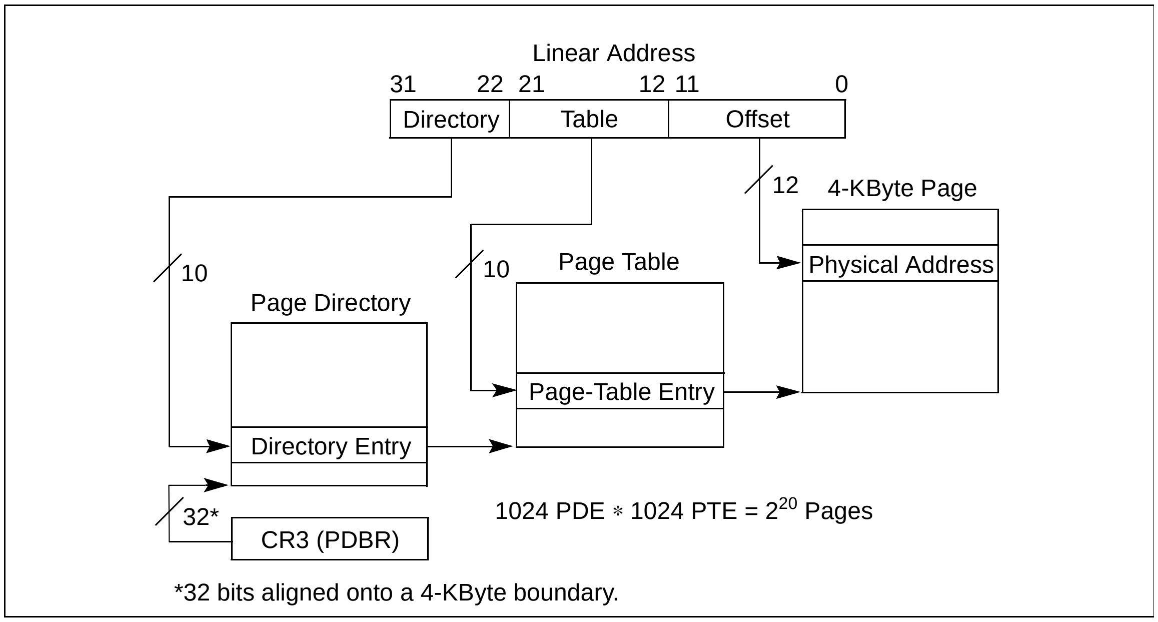


Figure: Paging Mechanism

- How does segmentation and paging work together?
  - Page-level protection can be used alone.
  - Page-level protection can be used together with the segmentation protection. In this case, the linear address produced by the sementation mechanism will be fed into the paging mechanism, and is eventually translated into physical address.
  - When paging is enabled, the 80386 first evaluates segment protection, then evaluates page protection. If the processor detects a protection violation at either the segment or the page level, the requested operation cannot proceed; a protection exception occurs instead.

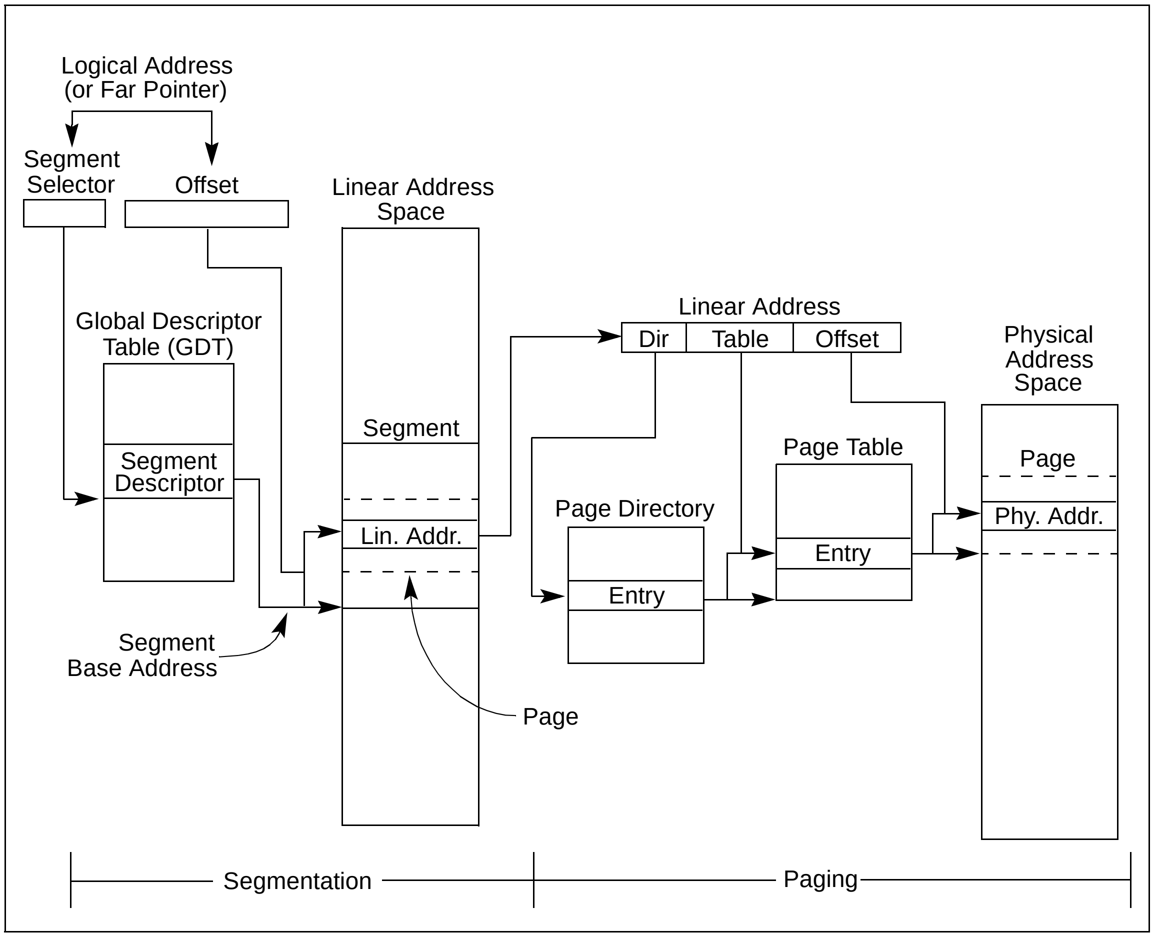


Figure: Segmentation and Paging

- Page-level protection: restrict access to pages based on two privilege levels:
  - Supervisor mode (U/S flag is 0)(Most privileged) For the operating system or executive, other system software (such as device drivers), and protected system data (such as page tables).
  - User mode (U/S flag is 1)(Least privileged) For application code and data.
  - When the processor is in supervisor mode, it can access all pages; when in user mode, it can access only user-level pages.
- The segment privilege levels map to the page privilege levels as follows:
  - If the processor is currently operating at a CPL of 0, 1, or 2, it is in supervisor mode;
  - If it is operating at a CPL of 3, it is in user mode.
- Page Type (read/write protection)
  - The page-level protection mechanism recognizes two page types:
    * Read-only access (R/W flag is 0).
    * Read/write access (R/W flag is 1).

  - When the processor is in supervisor mode and the WP flag in register CR0 is clear (its state following reset initialization), all pages are both readable and writable (write-protection is ignored).
  - When the processor is in user mode, it can write only to user-mode pages that are read/write accessible.
  - User-mode pages which are read/write or read-only are readable.
  - Supervisor-mode pages are neither readable nor writable from user mode.
  - A page-fault exception is generated on any attempt to violate the protection rules.

- Page-directory entry and table entry
  - The user/supervisor and read/write protections are applied to both page-directory entry and page entry
  - Bit 2 (U/S bit) is used for user/supervisor protection
  - Bit 1 (R/W bit) is used for read/write protection

7 Homework Questions

1. Why do we need to have access control in CPUs, while we already have access control in operating systems. What needs to be protected by 80386? If we do not protect them, what could go wrong?
2. In Linux, normal users cannot modify `/etc/shadow`. This is enforced by the access control in the operating system. If the CPU does not enforce any access control, please explain how a normal user can modify `/etc/shadow`. Please describe at least two different methods.
3. Why can't a program directly write to a kernel memory? What if this program is running with the root privilege?
4. Are there needs for a user-level program to modify kernel memory? Please list at least 3 scenarios where kernel memory is modified as the results of user-level programs.
5. Why do we have system calls in the operating systems? Why can't they be implemented as library functions. What are typical ways to implement system calls in operating systems?
6. What registers in 80386 CPUs need to be protected, so only privileged code can modify them? What if they are not protected?
7. Before designing an access control system, one needs to identify the subjects, objects, actions, and security policies. In 80386's access control, what are used as subjects, objects, and actions? What types of security policies are selected, and why?
8. What is the purpose of conforming bit in 80386?
9. What is the purpose of RPL?
10. What information in a segement descriptor is used for access control?
11. Is it possible for the same physical memory to belong to two different rings?

12. If a program is copying data to a buffer located towards the end of a segment, is it possible to overflow the segment as the result of buffer overflow?
13. How is the memory space of one process isolated from another process?
14. Why can two different processes use the same address (e.g. 0xF8A60000) without worrying about overwriting each other's data?
15. Does the 80386 Protection mode use capability in its access control? Where is the capability used?
16. How can operating systems restrict all I/O operations to be executed in the kernel only.
17. Please describe the design process that we went through in the class when discussing the 80386 protection mode. What are the key design questions that we asked, and how did we resolve them?