UMBC

CMPE - 310

Lecture 19 – Paging And Segmentation

Outline

Concepts in Paging and Segmentation

Privilege levels

Call gates

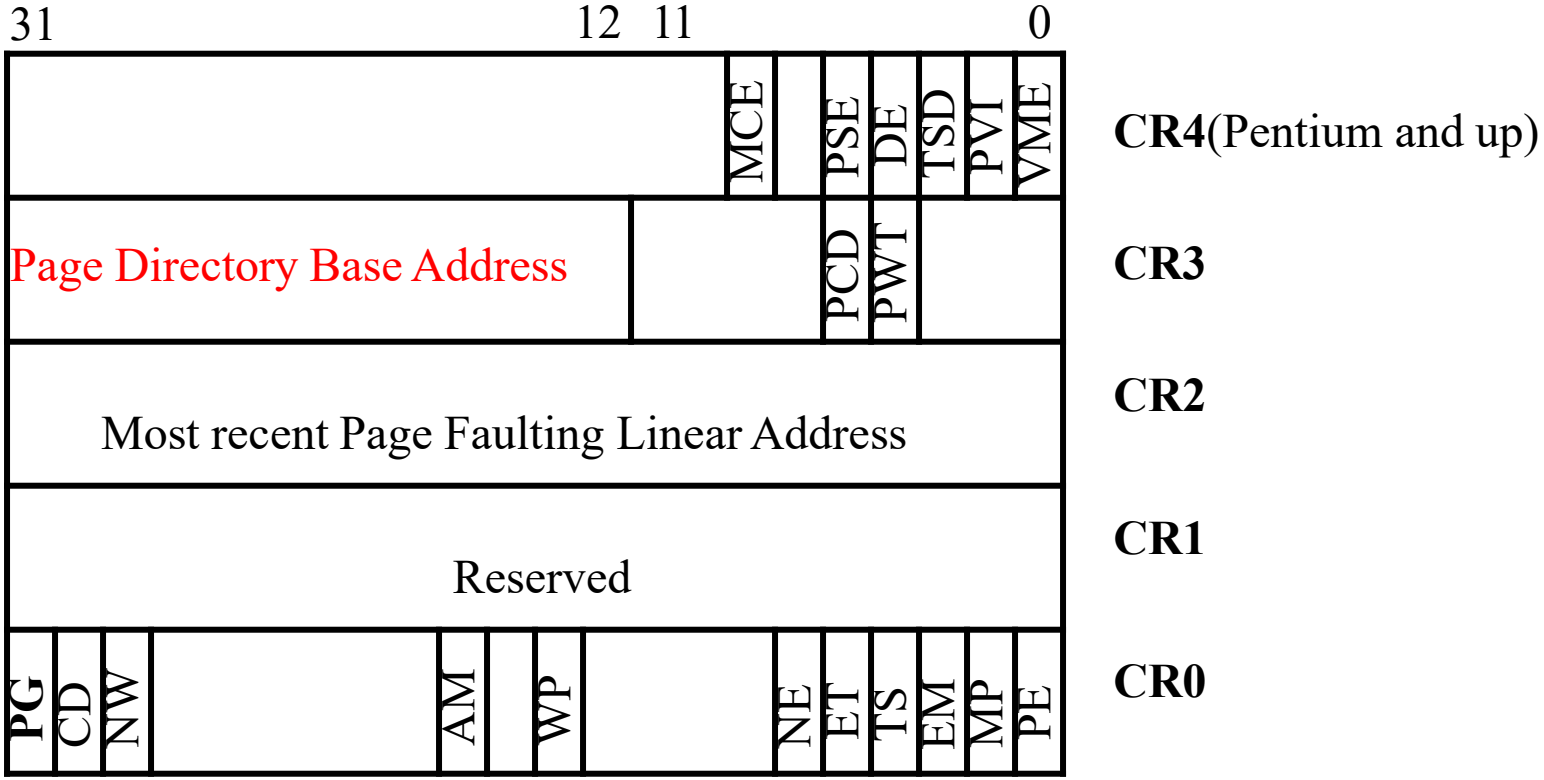
Memory Addressing

Memory Paging:

Available in the 80386 and up.

Allows a *linear address* (*virtual address*) of a program to be located in any portion of *physical* memory.

The **paging unit** is controlled by the microprocessors control registers:



Memory Addressing

Memory Paging:

The paging system operates in both real and protected mode.

It is enabled by setting the **PG** bit to 1 (left most bit in **CR0**).

(If set to 0, linear addresses are physical addresses).

CR3 contains the **page directory** 'physical' base address.

The value in this register is one of the few 'physical' addresses you will ever refer to in a running system.

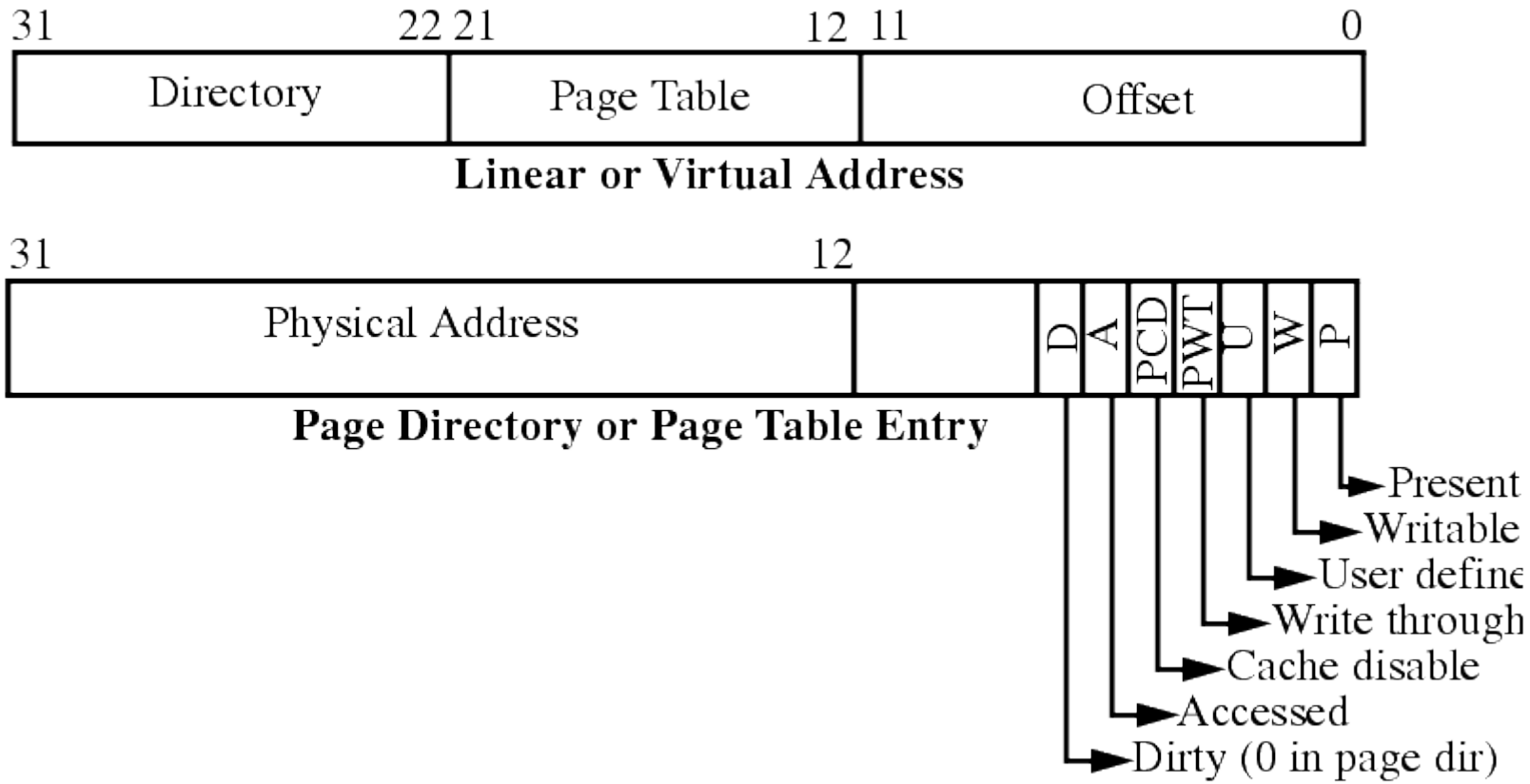
The **page directory** can reside at any 4K boundary since the low order 12 bits of the address are set to zero.

The **page directory** contains 1024 directory entries of 4 bytes each.

Each **page directory** entry addresses a **page table** that contains up to 1024 entries.

Memory Addressing

Memory Paging:

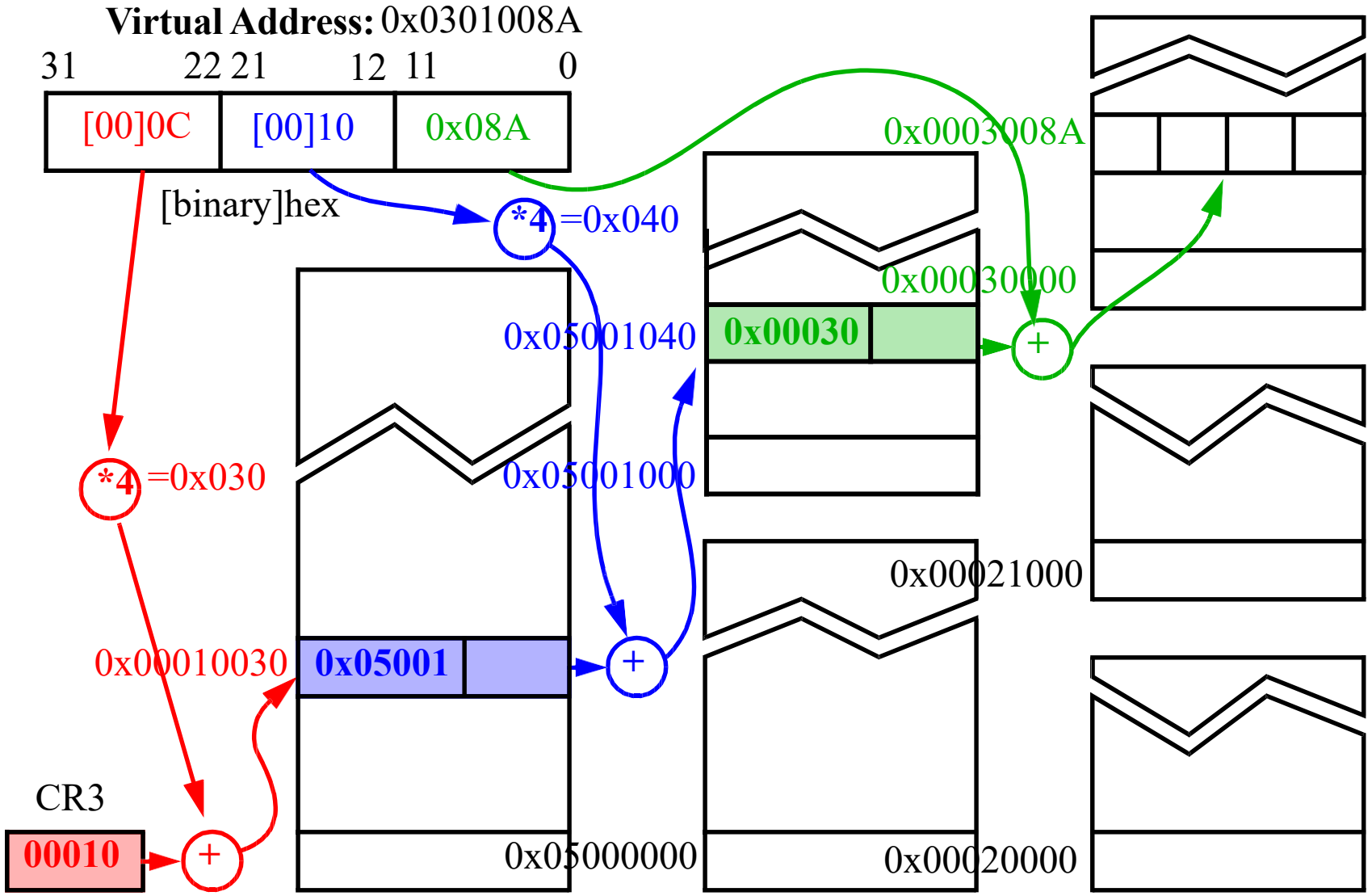


The virtual address is broken into three pieces:

- *Directory*: Each **page directory** addresses a 4MB section of main mem.
- *Page Table*: Each **page table** entry addresses a 4KB section of main mem.
- *Offset*: Specifies the byte in the page.

Memory Addressing

Memory Paging:



Memory Addressing

Memory Paging:

The **page directory** is 4K bytes.

Each **page table** is 4K bytes, and there are 1024 of them.

If all 4GB of memory is paged, the overhead is 4MB!

The current scheme requires three accesses to memory:

One to the **directory**, one to the appropriate **page table** and (finally) one to the desired data or code item. Ouch!

A **Translation Look-aside Buffer** (**TLB**) is used to cache page directory and page table entries to reduce the number of memory references.

Plus the data cache is used to hold recently accessed memory blocks.

System performance would be extremely bad without these features.

Much more on this in OS (CMSC 421).

Paging and Segmentation:

These two addresses translation mechanism are typically combined.

Segmentation and the User Application

The application programmer loads segment register values as before in Real Mode, but the values that he/she puts in them are very different.

Since knowledge of the GDT and LDT is **not** generally available at compile time, the programmer must use symbolic names.

The **loader** is responsible for resolving the actual values at run time.

In general, the segment values are *16-bit tags* for the address spaces of the program.

Instructions such as **LDS** (load DS), **LAR** (load access rights), **LSL** (load segment limit), **VERR** (verify for read) are available to retrieve *descriptor* attributes, if the process is privileged enough.

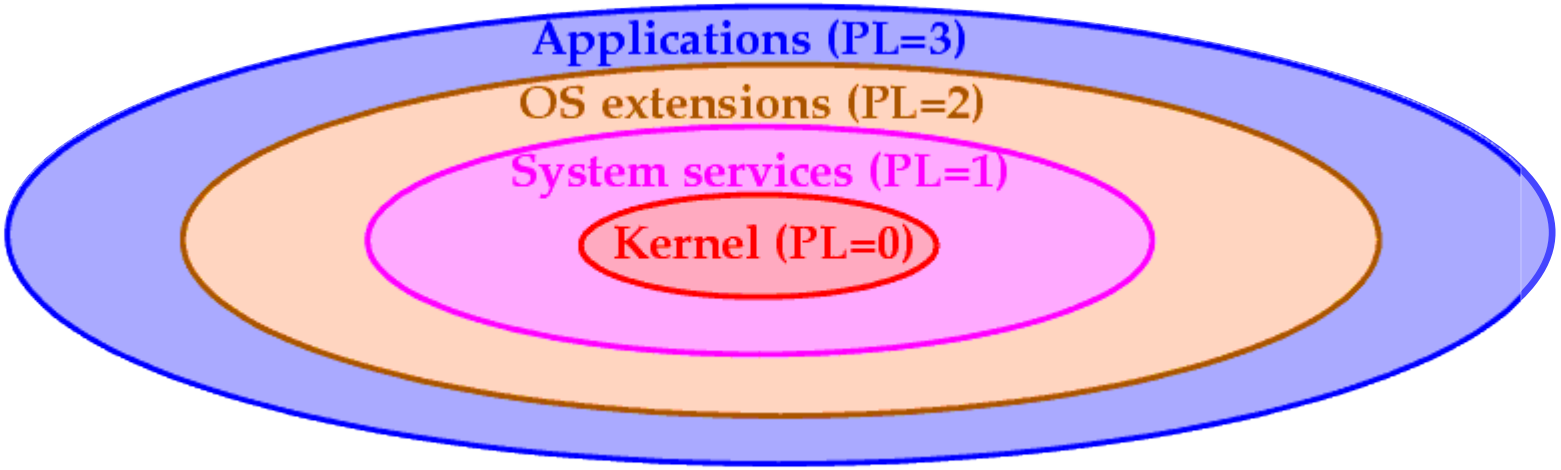
Whenever a segment register is changed, *sanity checks* are performed before the descriptor is cached.

- The index is checked against the limit.
- Other checks are made depending on the segment type, e.g., data segments, DS cannot be loaded with pointers to execute-only descriptors, ...
- The present flag is checked.

Otherwise, an exception is raised and nothing changes.

Privilege Levels

0: highest privilege, 3: lowest privilege



The privilege protection system plays a role for almost every instruction executed.

Protection mechanisms check if the process is privileged enough to:

- *Execute certain instructions*, e.g., those that modify the Interrupt flag, alter the segmentation, or affect the protection mechanism require PL 0.
- *Reference data other than its own*. References to data at **higher** privilege levels is not permitted.
- *Transfer control to code other than its own*. CALLs or JMPs to code with a **different** privilege level (higher or lower) is not permitted.

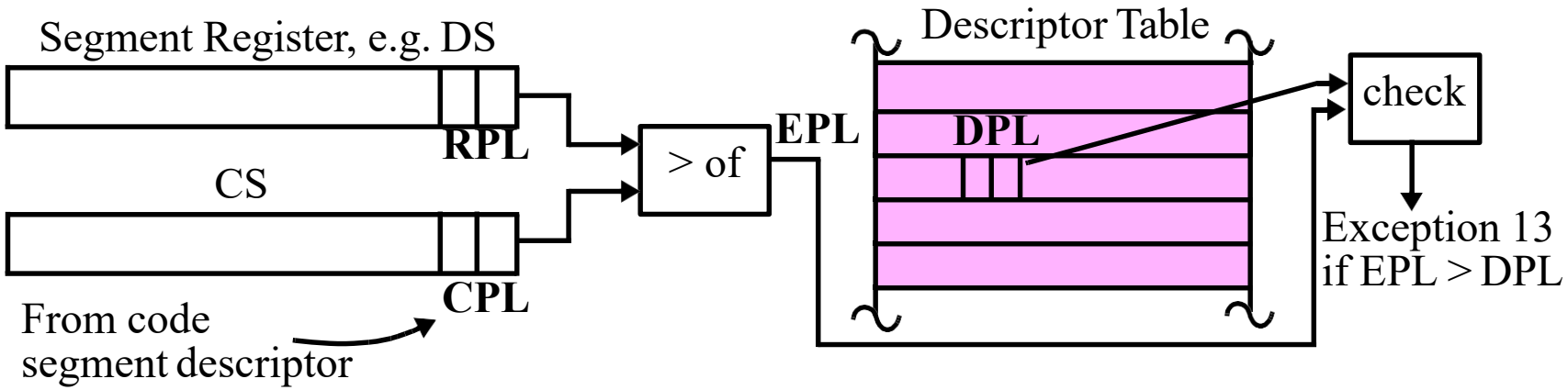
Privilege Levels

Privilege levels are *assigned to segments*, as we have seen, using the **DPL** (DescriptorPrivilege Level) field (bits 45 and 46).

Define **CPL** as the **Code Privilege Level** of the process, which is the **DPL** of its*code segment*!

Define **RPL** as the **Requestor's Privilege Level.**

Privilege Level Definitions:



When data selectors are loaded, the corresponding data segment's DPL is compared to the *larger* of your CPL or the selector's RPL.

Therefore, you can use RPL to *weaken* your current privilege level, if you want.

Privilege Levels

CPL is defined by the descriptors, so access to them must be restricted.

Privileged Instructions:

- Those that affect the segmentation and protection mechanisms (CPL=0 only).
  For example, LGDT, LTR, HLT.
- Those that alter the Interrupt flag (CPL <= IOPL field in EFLAGS).
  For example, CLI, STI (Note: only DPL 0 code can modify the **IOPL** fields.)
- Those that perform peripheral I/O (CPL <= IOPL field in EFLAGS).
  For example, IN, OUT.

Privileged Data References:

Two checks are made in this case:

- **Trying to load** the DS, ES, FS or GS register with a selector whose DPL is > the DPL of the code segment descriptor generates a *general protection fault*.
- **Trying to use** a data descriptor that has the proper privilege level can also be illegal, e.g. trying to write to a read-only segment.

For **SS**, the rules are even more restrictive.

Privilege Levels

Privileged Code References:

Transferring control to code in another segment is performed using the FAR forms of JMP, CALL and RET.

These differ from intra-segment (NEAR) transfers in that they change **both** CS and EIP.

The following checks are performed:

- The new selector must be a code segment (e.g. with execute attribute).
- CPL is set to the DPL (RPL is of no use here).
- The segment is present.
- The EIP is within the limits defined by the segment descriptor.

The RPL field is always set to the CPL of the process, independent of what was actually loaded.

You can examine the RPL field of CS to determine your CPL.

Changing CPL

There are two ways to **change** your CPL:

- *Conforming Code segments*.

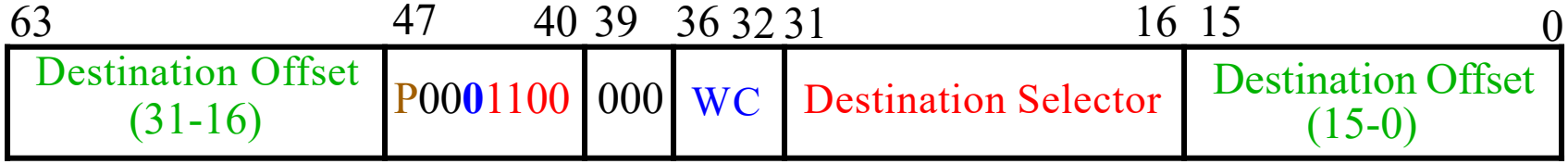
  Remember Types 6 and 7 defined in the AR byte of descriptor?

  Segments defined this way have no privilege level -- they conform to the level of the calling program.

  This mechanism is well suited to handle programs that share code but run at different privilege levels, e.g., shared libraries.

- Through special segment descriptors called **Call Gates**.

Call Gate descriptor:



Call gates act as an interface layer between code segments at different privilege levels.

They define *entry points* in more privileged code to which control can be transferred.

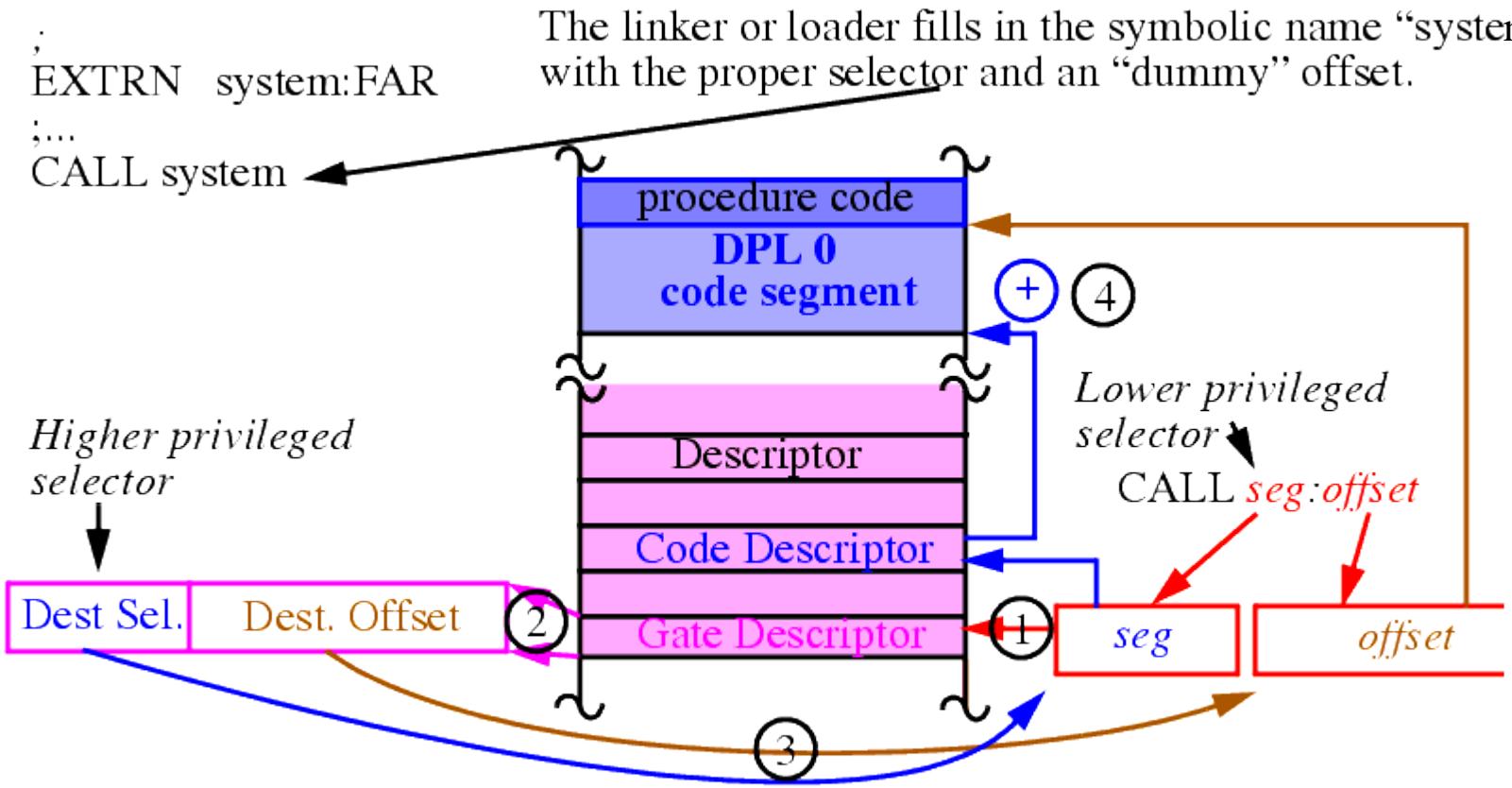
Call Gates

They must be referred to using FAR CALL instructions (no JMPs are allowed).

Note, references to call gates are *indistinguishable* from other FAR CALLs in the program -- a segment and offset are still both given.

However, in this case, both are ignored and the call gate data is used instead.

Call Gate Mechanism:



Call Gates

Note that both the **selector** and **offset** are given in the call gate preventing lower privileged programs from jumping into the middle of higher privileged code.

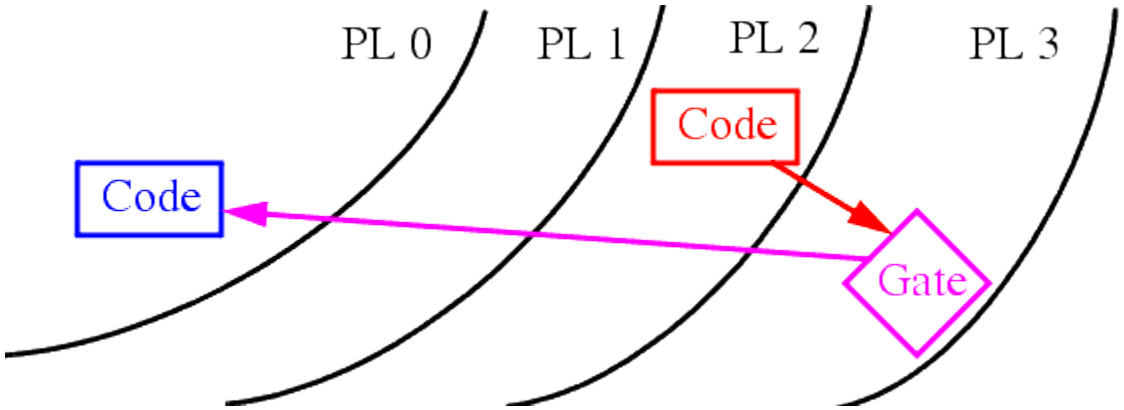
This mechanism makes the higher privileged code *invisible* to the caller.

Call Gates have 'tolls' as well, making some or all of them inaccessible to lower privileged processes.

The rule is that the Call Gate's DPL field (bits 45-46) MUST be >= (lower in privilege) than the process's CPL before the call.

Moreover, the privileged code segment's DPL field MUST be <= the process's CPL before the call.

Privileged Code DPL ≤ Max(RPL, CPL) ≤ Call Gate DPL



Call Gates

Changing privilege levels requires a *change in the stack* as well (otherwise, the protection mechanism would be sabotaged).

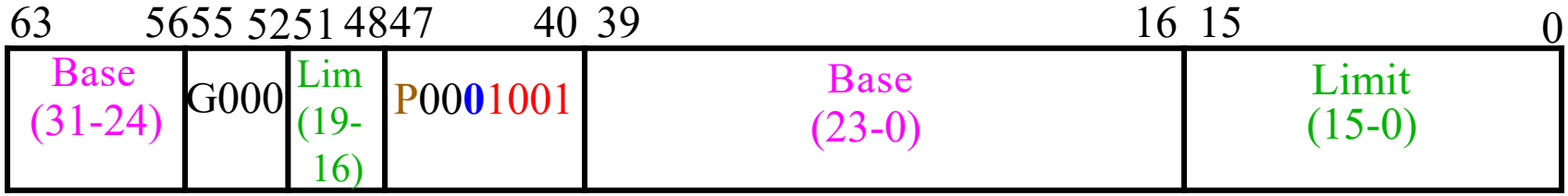
Stack segment DPLs MUST match the CPL of the process.

This happens transparently to the program code on both sides of the call gate!

Where does the new stack come from?

From yet another descriptor, Task State Segment (**TSS**) descriptor, and a special segment, the TSS.

The TSS stores the state of all tasks in the system and is described using a TSS descriptor.



The processor saves all the information it needs to know about a task in the TSS.