

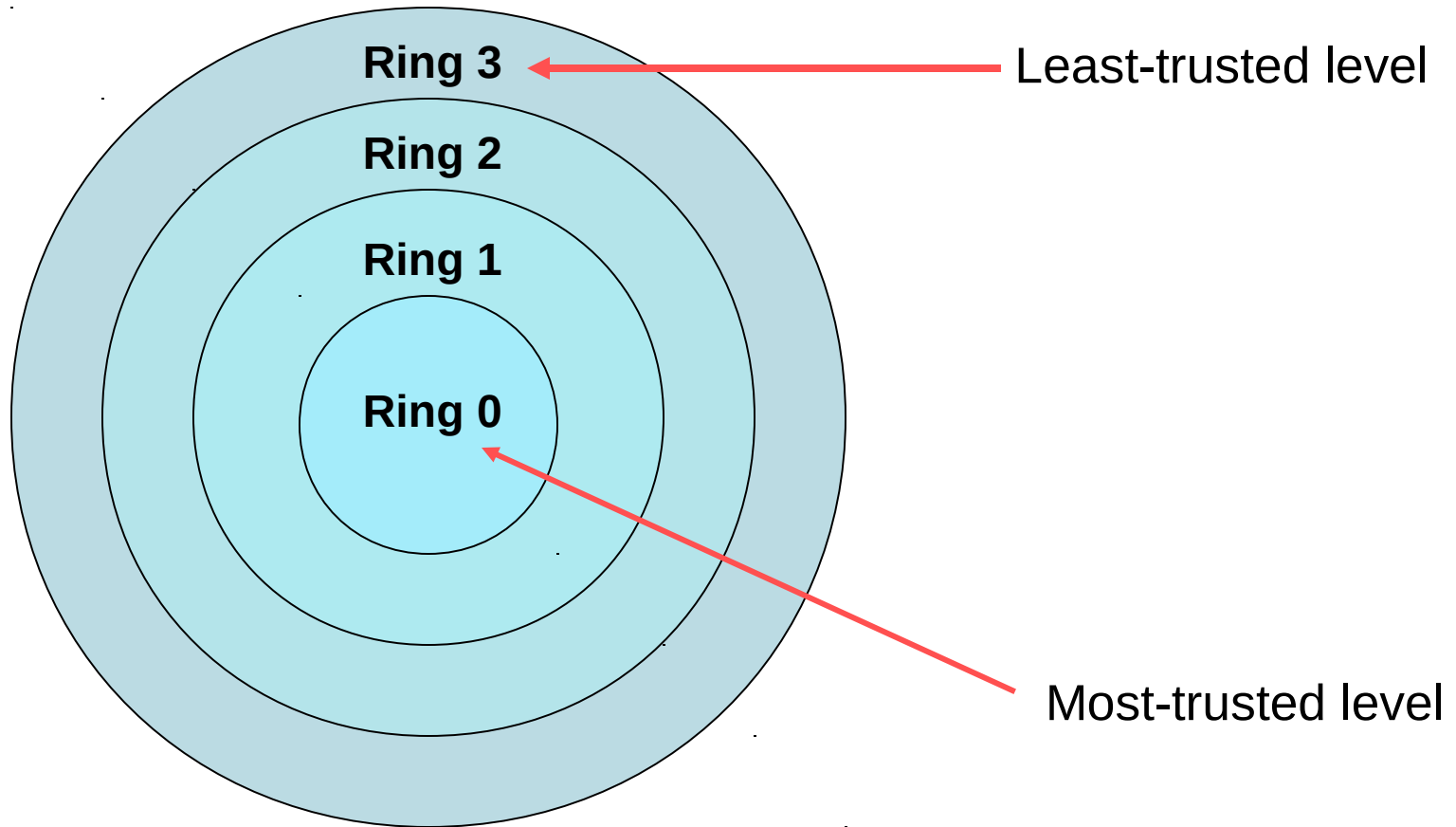
Ring-transitions for EM64T

How the CPU can accomplish transitions among its differing privilege-levels in 64-bit mode

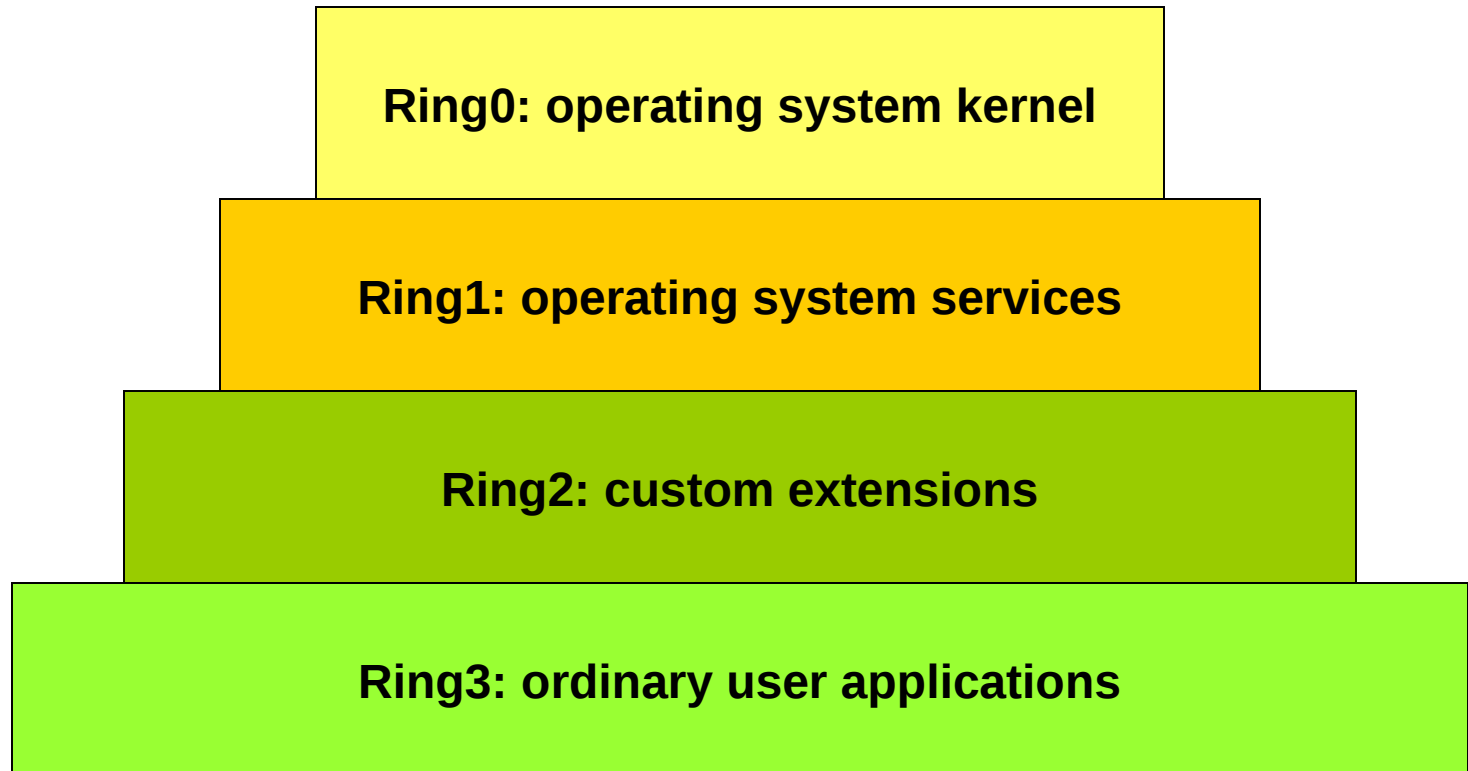
Rationale

- The usefulness of protected-mode derives from its ability to enforce restrictions upon software's freedom to take certain actions
- Four distinct privilege-levels are supported
- Organizing concept is “concentric rings”
- Innermost ring has greatest privileges, and privileges diminish as rings move outward

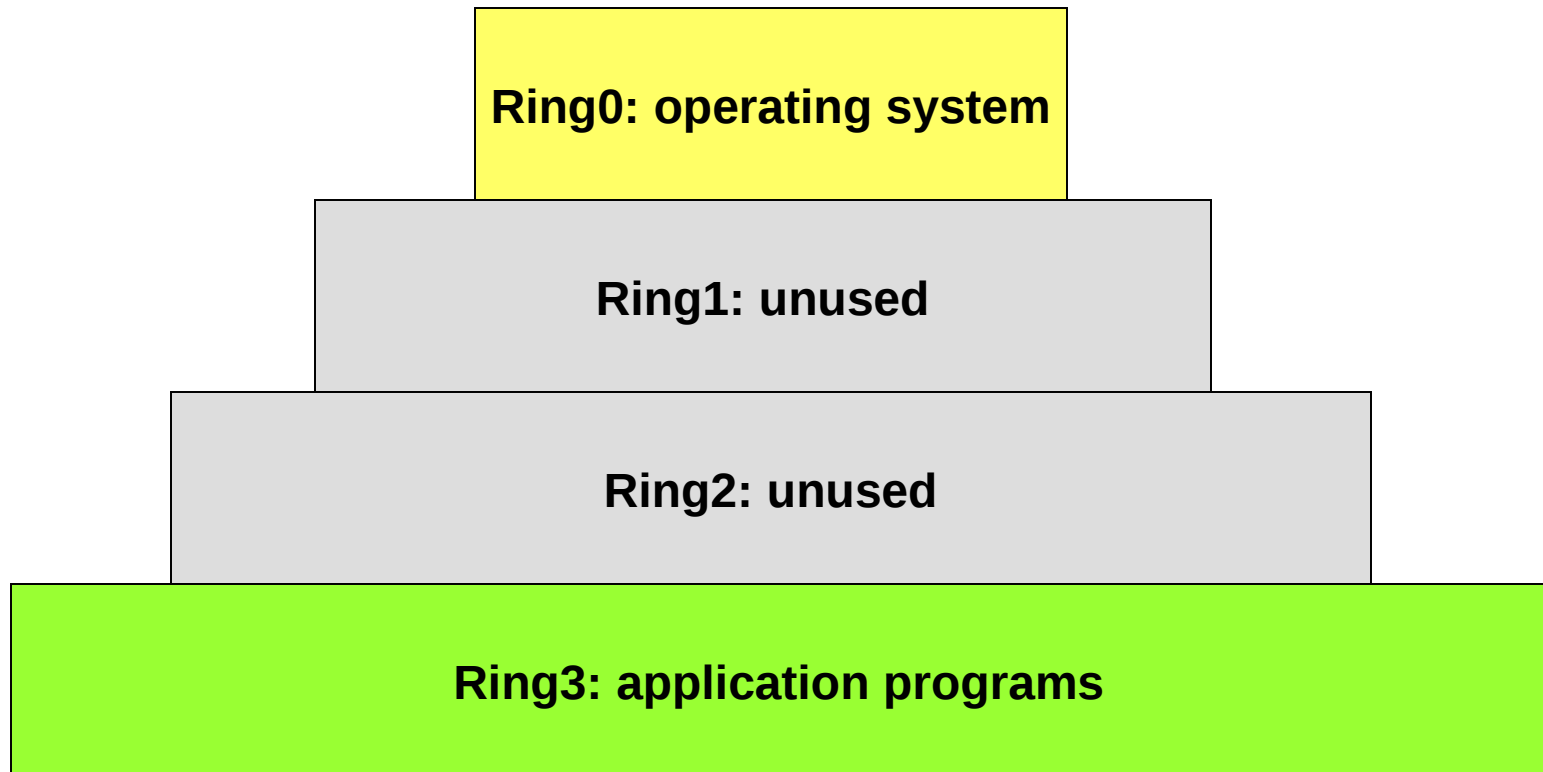
Four Privilege Rings



Suggested purposes



Unix/Linux and Windows



Legal Ring-Transitions

- A transition from an outer ring to an inner ring is made possible by using a special control-structure (known as a 'call gate')
- The 'gate' is defined via a data-structure located in a 'system' memory-segment normally not accessible for modifications
- A transition from an inner ring to an outer ring is not nearly so strictly controlled

Data-sharing

- Function-calls typically require that two separate routines share some data-values (e.g., parameter-values get passed from the calling routine to the called routine)
- To support reentrancy and recursion, the processor's stack-segment is frequently used as a 'shared-access' storage-area
- But among routines with different levels of privilege this could create a "security hole"

An example senario

- Say a procedure that executes in ring 3 calls a procedure that executes in ring 2
- The ring 2 procedure uses a portion of its stack-area to create 'automatic' variables that it uses for temporary workspace
- Upon return, the ring 3 procedure would be able to examine whatever values are left behind in this ring 2 workspace

Data Isolation

- To guard against unintentional sharing of privileged information, different stacks are provided at each distinct privilege-level
- Accordingly, any transition from one ring to another must necessarily be accompanied by an mandatory 'stack-switch' operation
- The CPU provides for automatic switching of stacks and copying of parameter-values

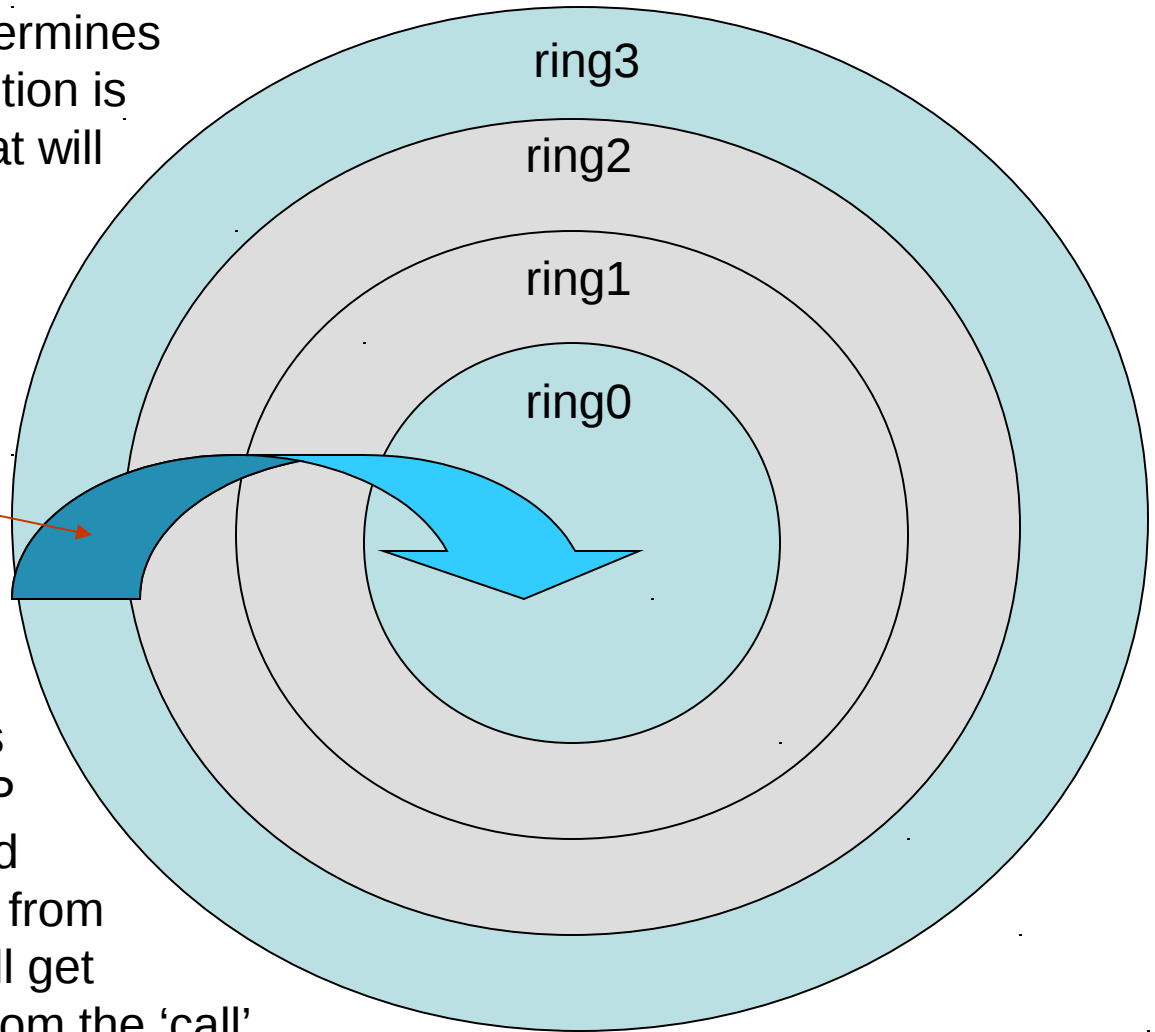
Inward ring-transitions

- Transfers from a ring with lesser privileges to a ring with greater privileges (e.g., from ring3 to ring0) are controlled by a system data-structure known as a ‘call gate’ and normally would be accomplished using an ‘lcall’ instruction (i.e., a ‘long’ call), either “direct” (the target is specified by data in the instruction) or “indirect” (the target is specified by data at a memory-location)

requires 'gate' and 'TSS'

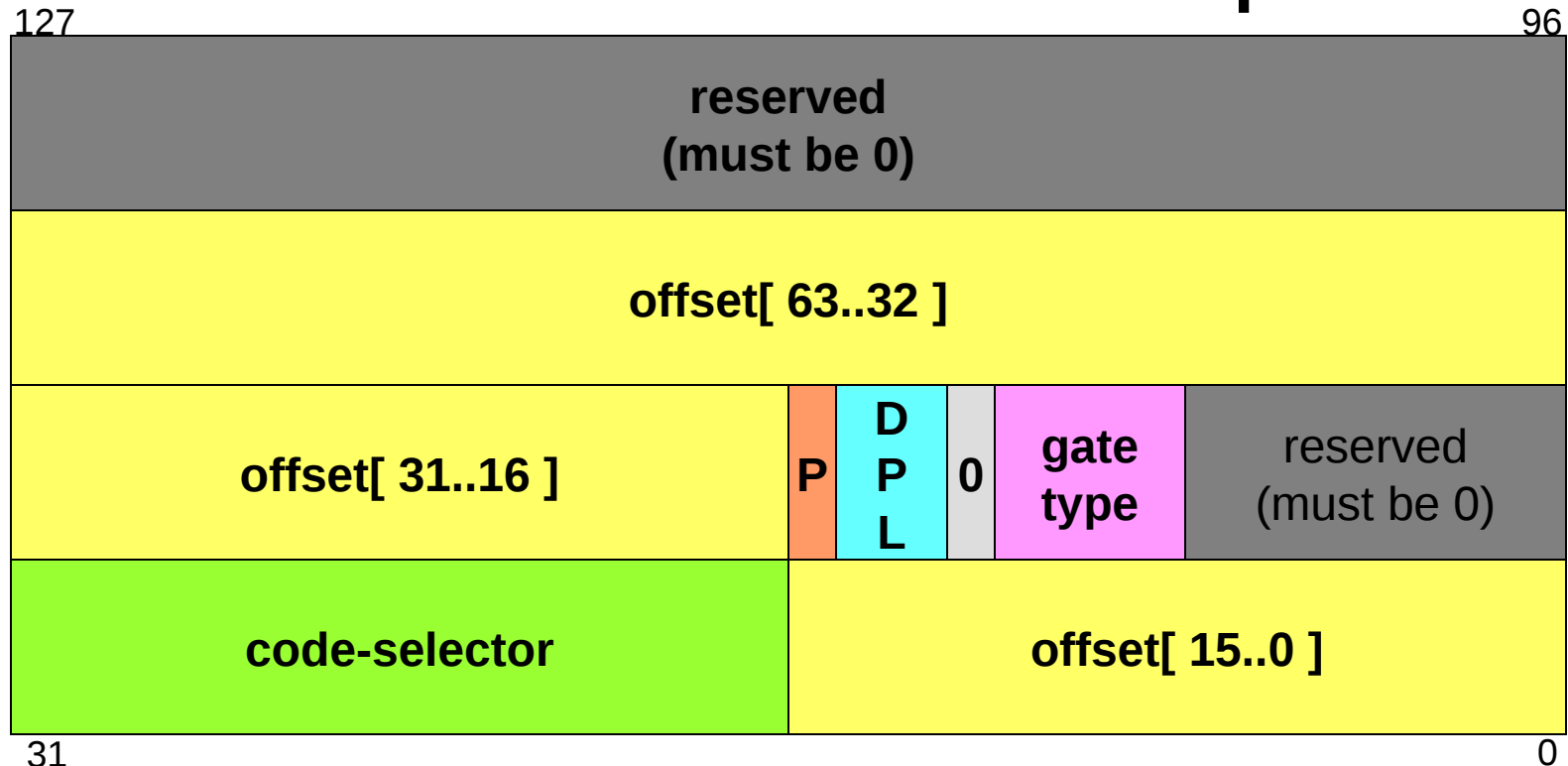
'gate' structure's DPL determines whether the inward transition is permitted, and if it is, what will be the new CS and RIP register-values

lcall instruction



'TSS' structure determines what the new SS and RSP register-values will be, and thus where the old values from SS, RSP, CS, and RIP will get saved for a later 'return' from the 'call'

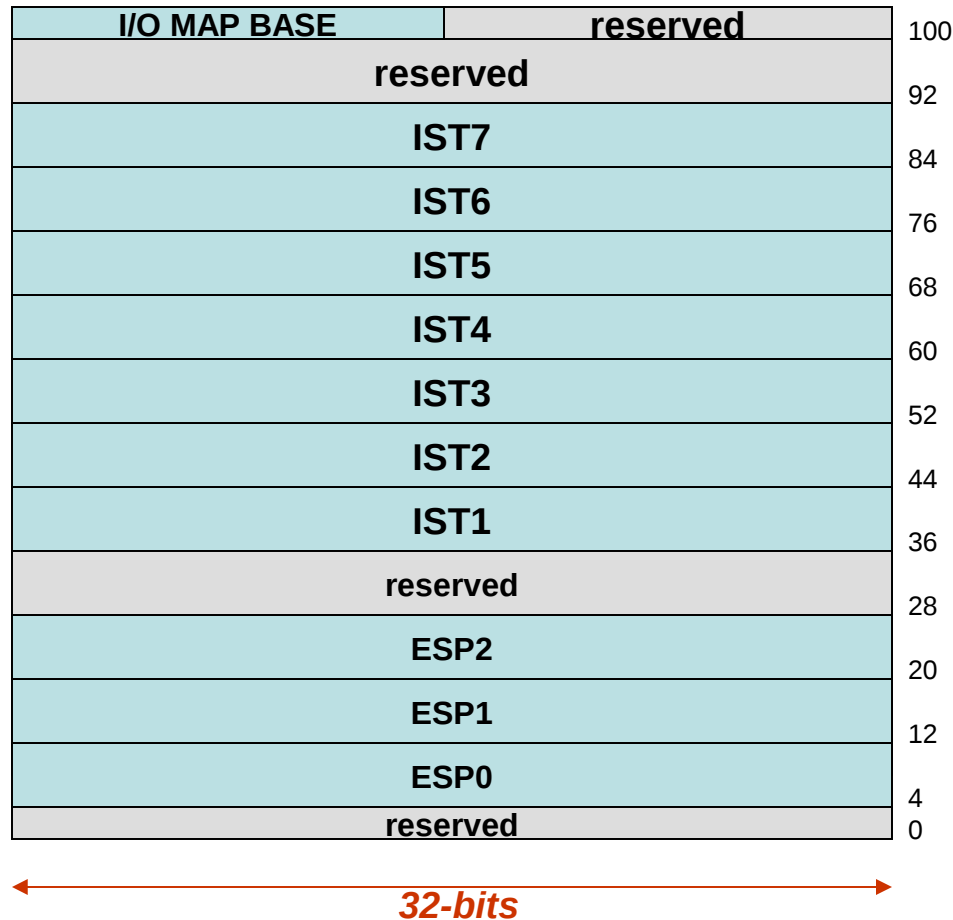
64-bit Call-Gate Descriptors




Legend:

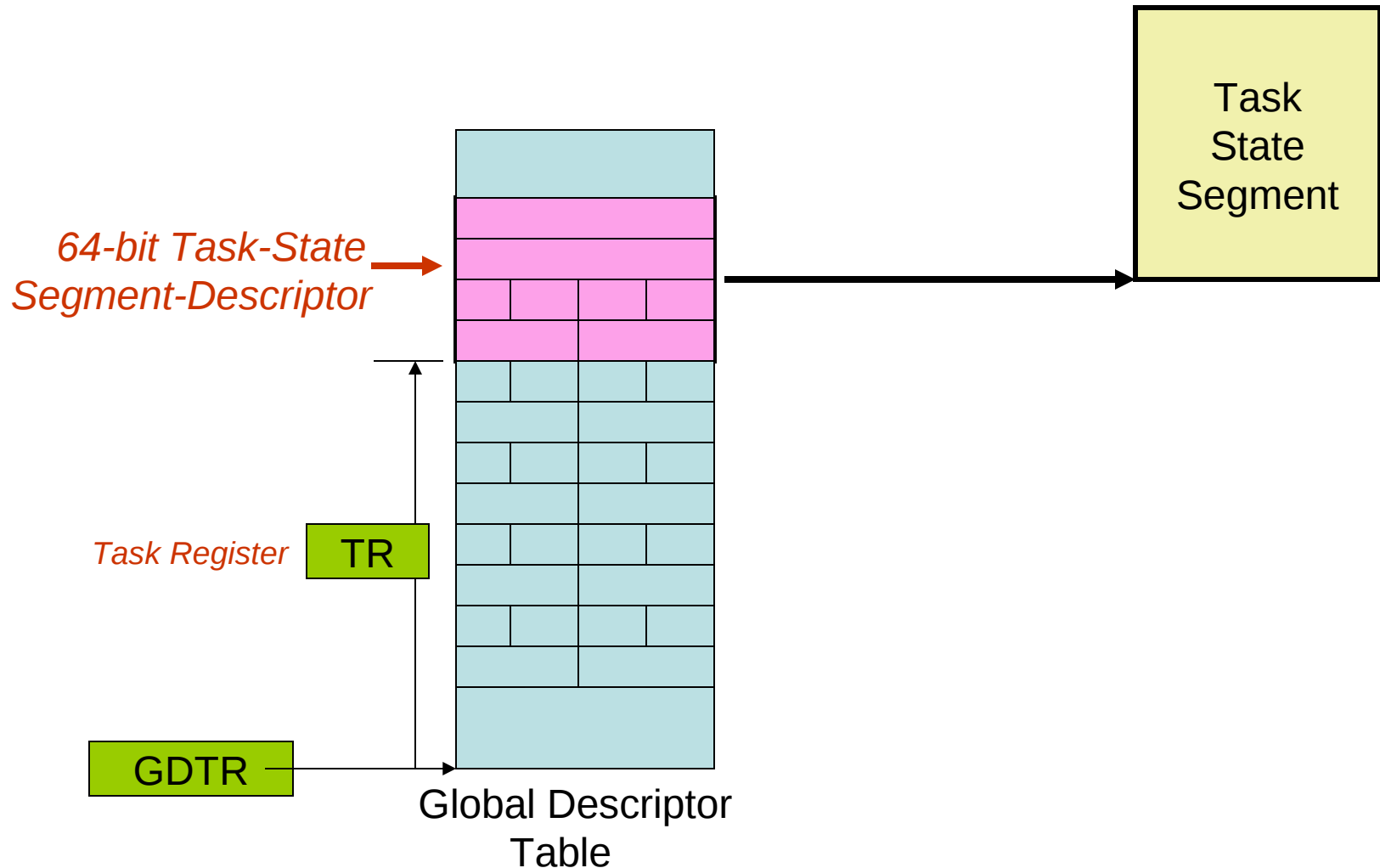
P=present (1=yes, 0=no) DPL=Descriptor Privilege Level (0,1,2,3)
code-selector (specifies memory-segment containing procedure code)
offset (specifies the procedure's entry-point within its code-segment)
gate-type: ('0xC' signifies a 64-bit call-gate when EFER.LMA=1)

64-bit Task-State Segment



 Reserved bits)must be set to zero)

How CPU finds the TSS



Outward ring-transitions

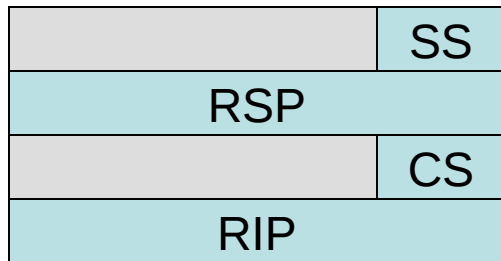
- Transfers from a ring with greater privilege to a ring having lesser privilege (e.g., from ring0 to ring3) are normally accomplished by using an 'lret' instruction (i.e., a 'long' return) and refer to values on the current stack to specify the changes in contents for registers CS and RIP (for the code transfer) and registers SS and RSP (for the mandatory stack-switch)

'returns' are less restrictive

Iret instruction

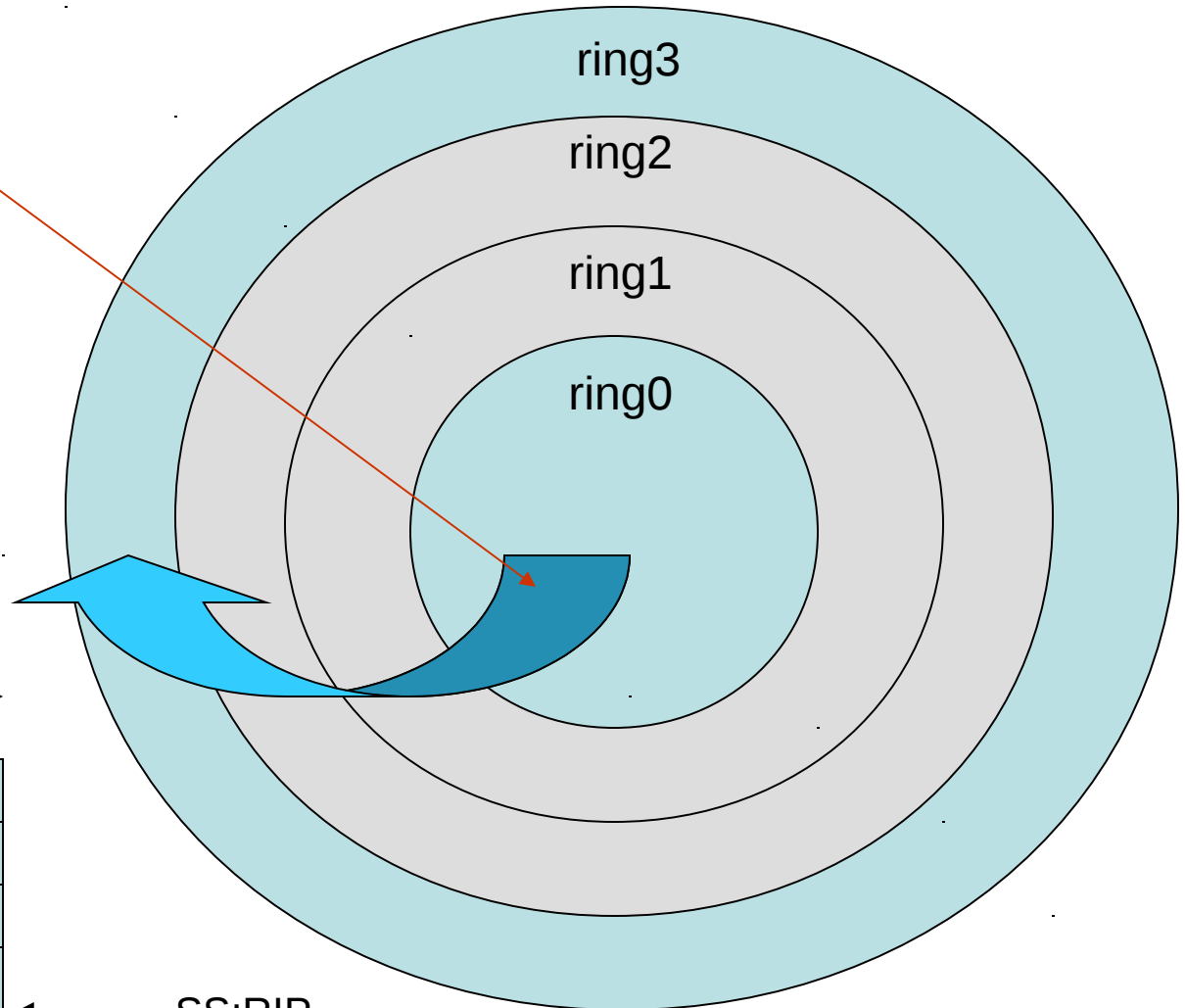
The new values for the CS, RIP, SS, and RSP registers will be taken from the current stack

64-bits



ring0 stack

SS:RIP

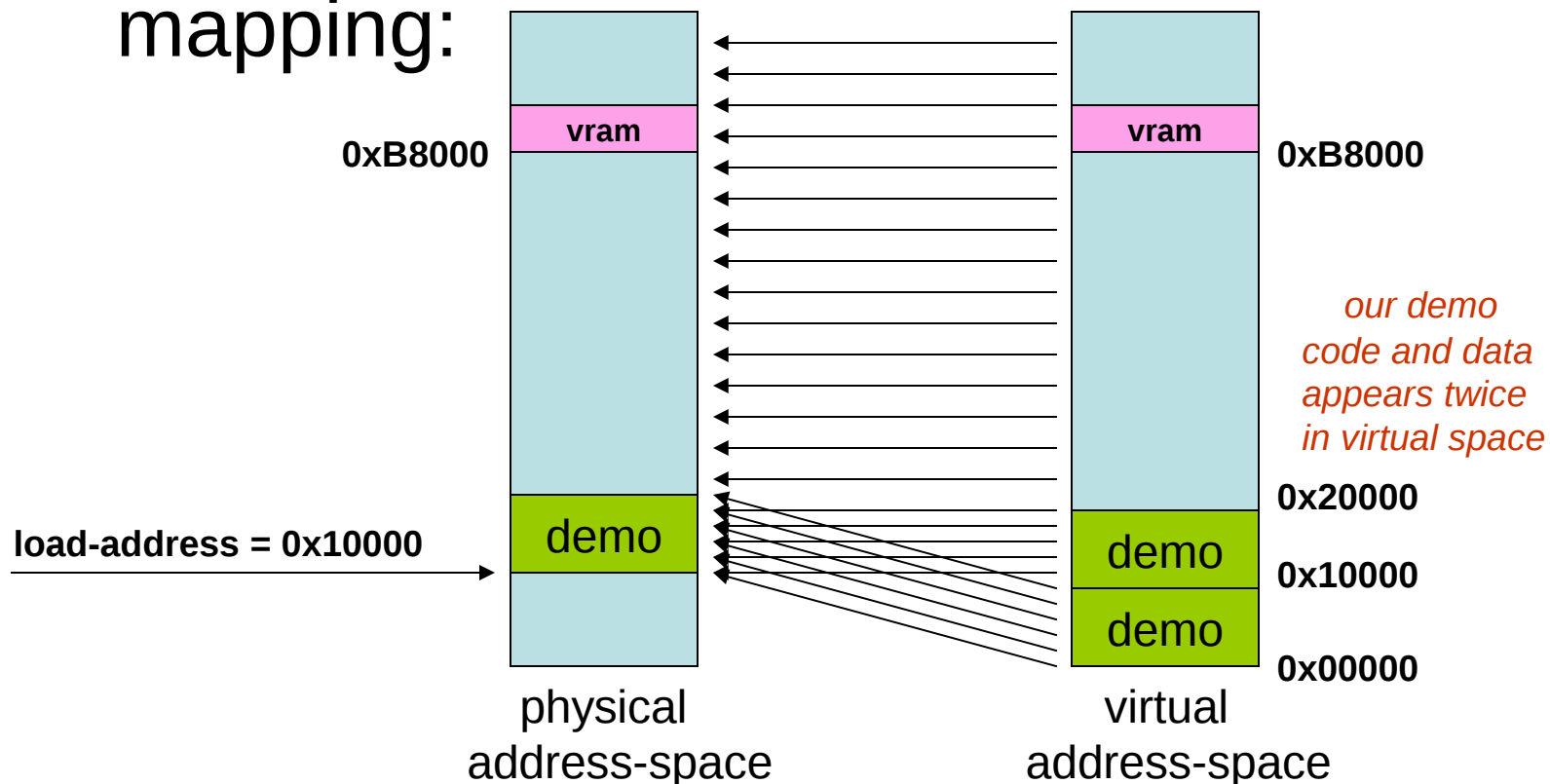


64-bit memory-addressing

- Recall that memory-addressing in 64-bit mode uses a ‘flat’ address-space (i.e., no segmentation: all addresses are offsets from zero, and no limit-checking is done)
- However, page-mapping is in effect, using the 4-level page-table scheme
- At least one page needs to be “identity-mapped” for the activation or deactivation of the processor’s ‘long’ mode (IA-32e)

New 'page-mapping' idea

- We can simplify our program addressing in 64-bit mode with a 'non-identity' mapping:

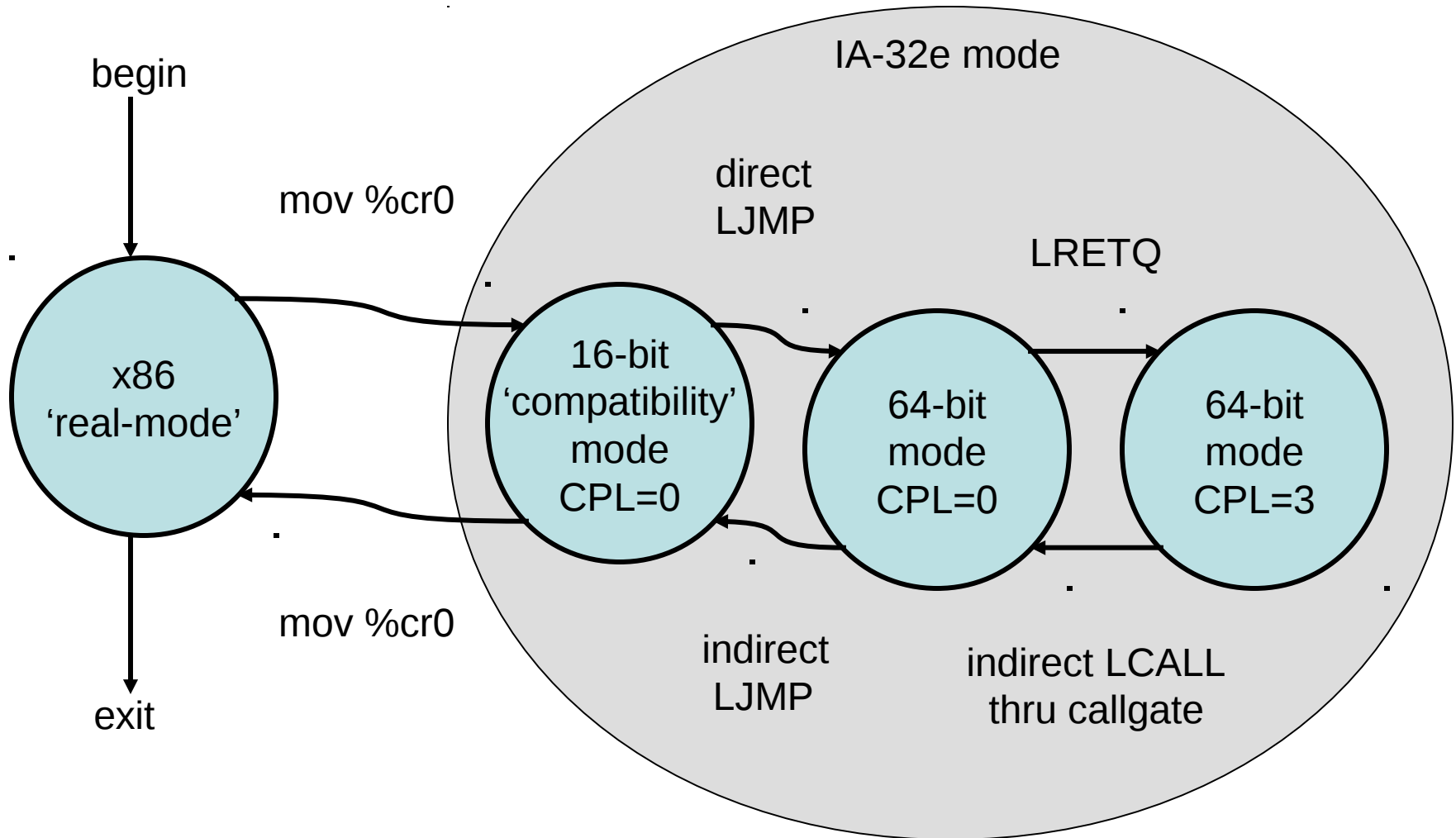


How we build the map-tables

```

        .section  .data
        .align    0x1000
level1:  entry = 0x10000
        .rept     16
        .quad     entry + 7
        entry = entry + 0x1000
        .endr
        entry     = 0x10000
        .rept     240
        entry = entry + 0x1000
        .quad     entry + 7
        .endr
        .align    0x1000
level2:  .quad     level1 + 0x10000 + 7
        .align    0x1000
level3:  .quad     level2 + 0x10000 + 7
        .align    0x1000
level4:  .quad     level3 + 0x10000 + 7
        .align    0x1000
```

Our 'trying3.s' demo



From 16-bit to 64-bit

- Once we arrive in IA-32e mode, our first transfer is from 16-bit ‘compatibility’ mode to 64-bit ‘long’ mode
- For this we can use a **long direct jump** with a default (i.e., 16-bit) operand-size (thanks to our special page-mapping)

```
ljmp    $sel_CS0, $prog64
```

selector for 64-bit code-segment
at privilege-level zero (i.e., ring0)

16-bit offset to our ‘prog64’ label

From 64-bit ring0 to 64-bit ring3

- Once we arrive in 64-bit mode, our second transfer is from ring0 to ring3
- For this we use a 'long return' instruction after setting up our current ring0 stack with the new values for SS, RSP, CS, and RIP, and we specify a quadword operand-size

```
pushq $sel_SS3
```

```
pushq $tos3
```

```
pushq $sel_CS3
```

```
pushq $showmsg
```

```
lretq
```

selector for writable data-segment
at privilege-level three (i.e., ring3)

selector for 64-bit code-segment
at privilege-level three (i.e., ring3)

From 64-bit ring3 to 64-bit ring0

- When we've finished our ring3 procedure, we get back to ring0 by using an **indirect** 'long call' through a 64-bit call-gate
- Our 64-bit TSS must be set up in advance for the accompanying stack-switch

lcall	*supervisor	# indirect long-call
supervisor: .long	0, sel_ret	# target of the call

selector for 64-bit call-gate descriptor accessible at ring3, specifying new register-values for CS and RIP (the new RSP-value comes from the TSS, and the new SS-value will be NULL)

a "dummy" operand, required by the syntax for 'lcall', but not used by the CPU (since all the needed info is in the call-gate and the Task-State Segment)

From 64-bit code to 16-bit code

- Finally, for returning to 'real-mode', we need to transfer from the 64-bit code in 'long mode' to 16-bit code in 'compatibility mode' (both ring0, so no privilege-change)
- For this we use an **indirect** long jump (as there's no direct long jump in 64-bit mode)

<code>ljmp</code>	<code>*departure</code>	<code># indirect long-jump</code>
<code>departure: .long</code>	<code>prog16, sel_cs0</code>	<code># target of this jump</code>

32-bit offset to our 'prog16' label

selector for 16-bit code-segment at privilege-level zero (i.e., ring0)

In-class exercise #1

- Try some alternative transfer-instructions:
 - Use `LJMPL $sel_C0, $prog64 + 0x10000`
in place of `LJMP $sel_C0, $prog64`
 - Use `LJMP *supervisor`
in place of `LCALL *supervisor`
 - Use `LRETW` in place of `LRETQ`
after setting up your stack with word-size
values instead of quadword-sized values

In-class exercises #2 and #3

- Can you adjust all the virtual addresses in your 64-bit code so that an 'identity-map' could be used in this 'tryring3.s' demo?
- Could you adjust all the privilege-levels so that 'ring2' gets used instead of 'ring3'?