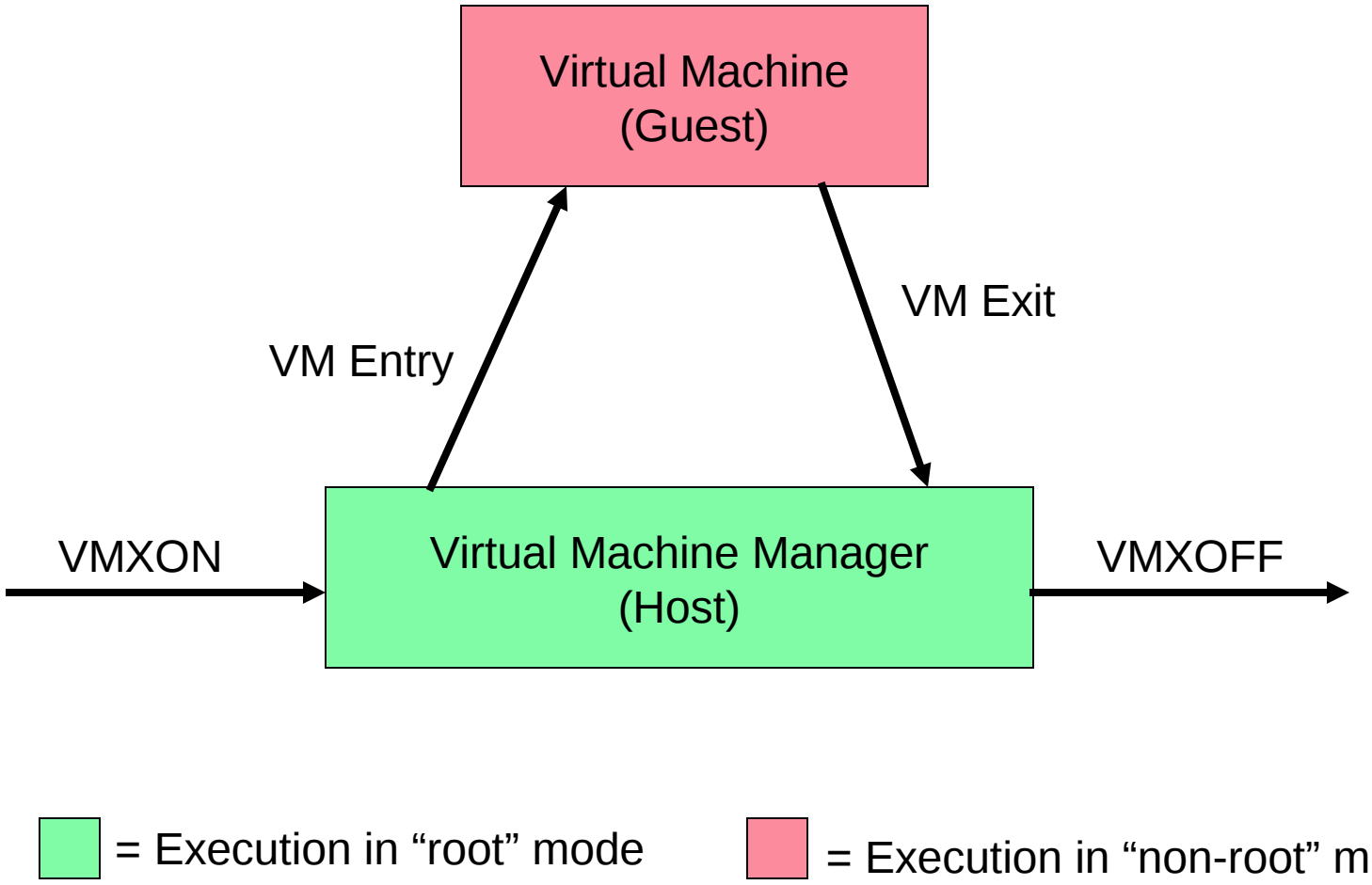
Constructing a VMX Demo

A “hands-on” exploration of Intel’s Virtual Machine Extensions

Guest and Host



The ten VMX instructions

- These nine execute in “root” mode:
  - VMXON and VMXOFF
  - VMPTRLD and VMPTRST
  - VMCLEAR
  - VMWRITE and VMREAD
  - VMLAUNCH and VMRESUME
- This one executes in “non-root” mode:
  - VMCALL

Step-by-step

- To become familiar with x86 Virtualization Technology, we propose to build a simple Guest example, with accompanying Host as well as appropriate VMX controls
- Each of these elements (Guest, Host, and Controls) will require us to construct some supporting data-structures ahead of time
- We can proceed in a step-by-step manner

Our ‘guest’ and ‘host’ modes

- Let’s arrange for our guest to operate as if it were a ‘virtual’ 8086 processor executing ‘real-mode’ code in a 1-MB address-space
- Let’s have our host execute 64-bit code in Intel’s advanced IA-32e protected-mode
- This plan should serve to demonstrate a useful aspect of VT -- since otherwise we couldn’t run real-mode code under IA-32e

‘Virtual-8086’

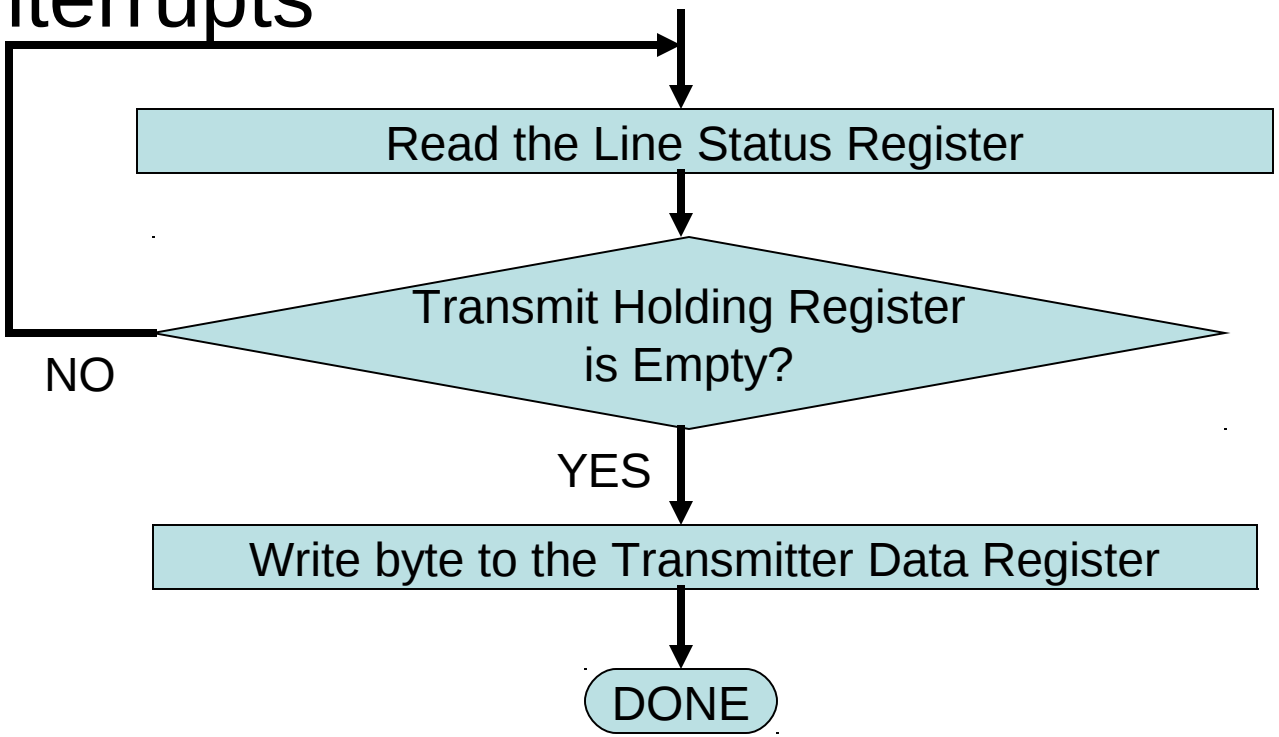
- We can’t execute ‘real-mode’ code in its ‘native’ processor-mode, but we can use the so-called virtual-8086 emulation-mode (it’s available as a sub-mode under 32-bit legacy protected-mode) if we use ‘paging’
- We will need supporting data-structures:
  - Page-mapping tables and Descriptor Tables
  - A 32-bit Task-State Segment
  - Stack-areas for ring-3 and for ring-0

What will our ‘guest’ do?

- We want to keep things simple, but we do need for our guest task to do something that has a perceptible effect – so we will know that it did, in fact, work as intended
- Drawing a message onscreen won’t work because we can’t see the actual display using our ‘remote’ Core-2 Duo machines
- Idea: transmit a message via the UART

The UART issues

- To keep things simple, let's have our guest employ 'polling' rather than use 'interrupts'



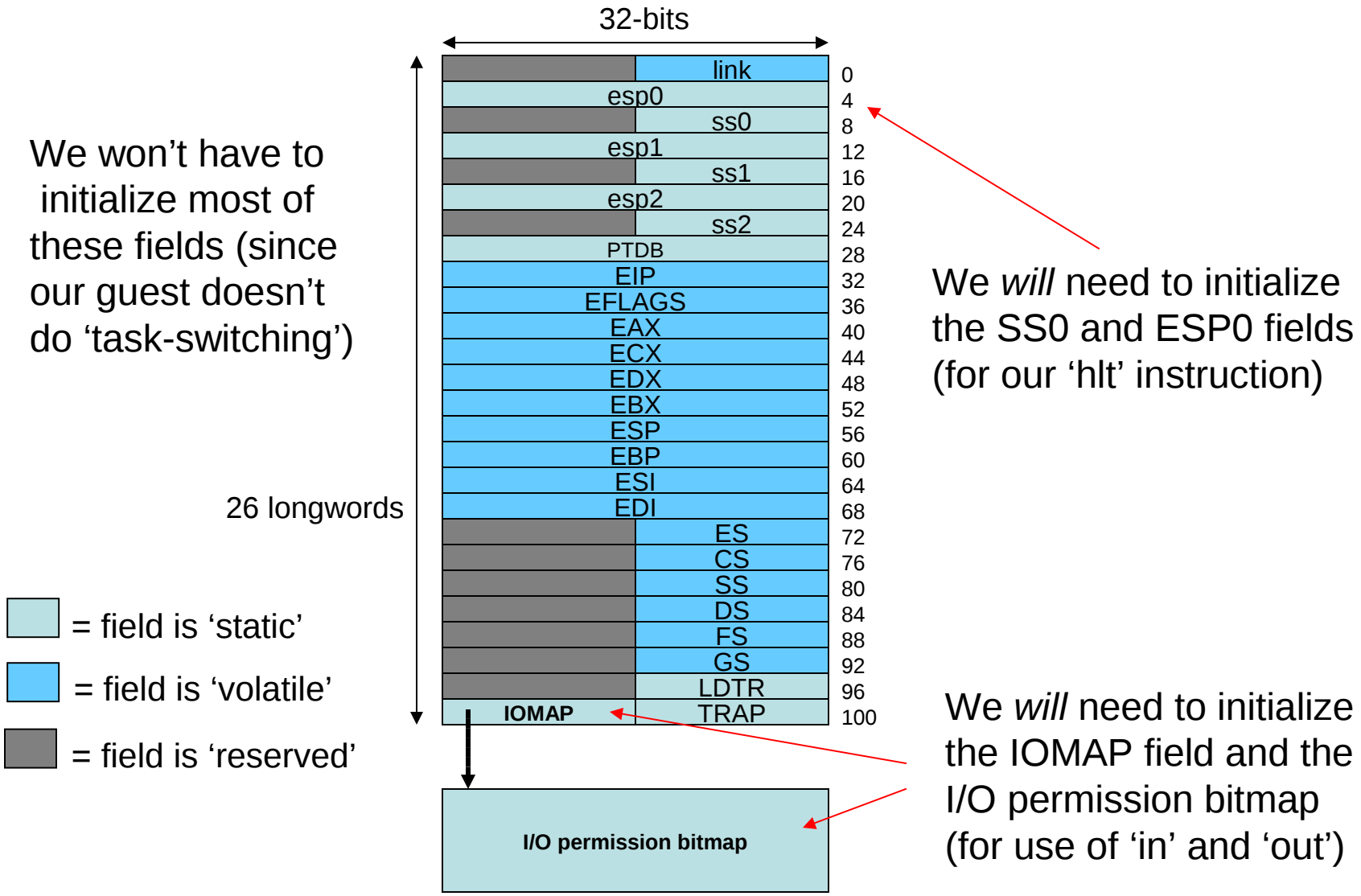
Our guest's code

```
# This loop uses 'polling' to transmit a message via the serial UART
            .code16                         # for x86 'real-mode' instructions
            mov     $0x1000, %ax            # 'real-mode' segment-address
            mov     %ax, %ds                #    loaded into the DS register
            xor     %si, %si                # initialize message array-index
nxbyte:     mov     $UART+5, %dx            # i/o port for UART's Line-Status
            in      %dx, %al # input the current line-status
            test    $0x20, %al              # Tx-Holding Register Empty?
            jz      nxbyte                  # not yet, check status again
            mov     msg(%si), %al           # else fetch the next character
            or      %al, %al                # is it the final null-byte?
            jz      done                    # yes, this loop is concluded
            mov     $UART+0, %dx            # else setup i/o port for Tx-Data
            out     %al, %dx # and transmit that character
            inc     %si                     # then advance the array-index
            jmp     nxbyte                  # and go back for another byte
done:       hlt                             # else exit from the guest task
#-------------------------------------------------------------------------------------------
msg:        .asciz      " Hello from our 8086 Virtual Machine guest "
```

Can ring3 code do I/O?

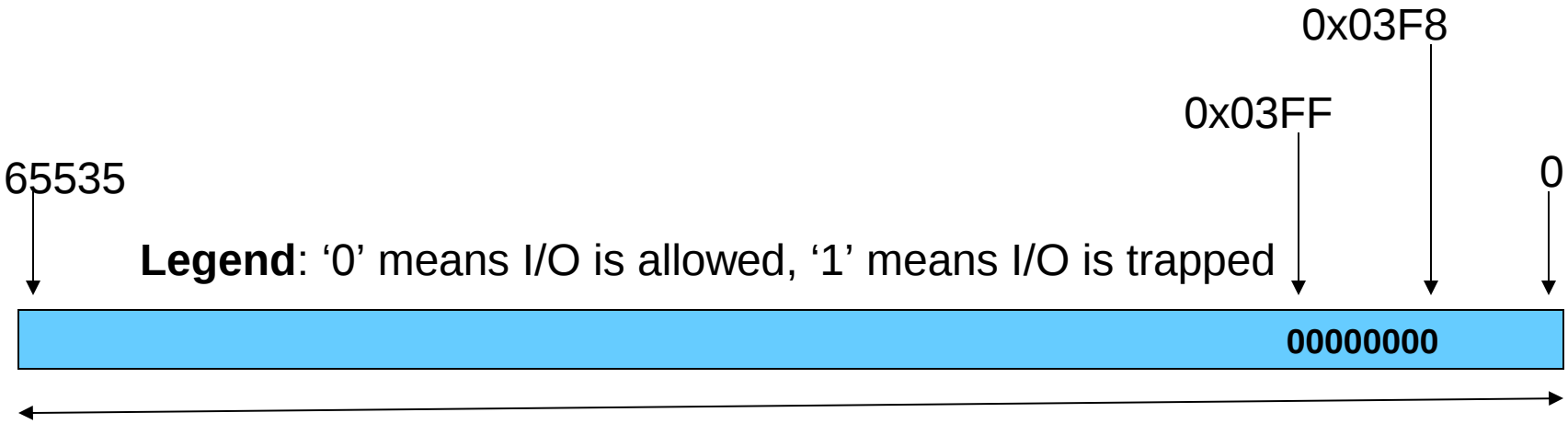
- Execution in ‘virtual-8086’ mode occurs at privilege-level 3, so input/output to devices is subject to ‘protected-mode’ restrictions
- It is ‘allowed’ on a port-by-port basis by a data-structure in the Task-State Segment known as the “I/O Permission Bitmap”
- We will need to build a TSS that includes that ‘bitmap’ data-structure

The 80386 TSS format



I/O permission bitmap

- There is potentially one bit for every I/O port-address (thus, up to 65536 bits!)
- Our ‘guest’ only needs to use UART ports



To encompass **all** the i/o ports, the bitmap needs 8K bytes!

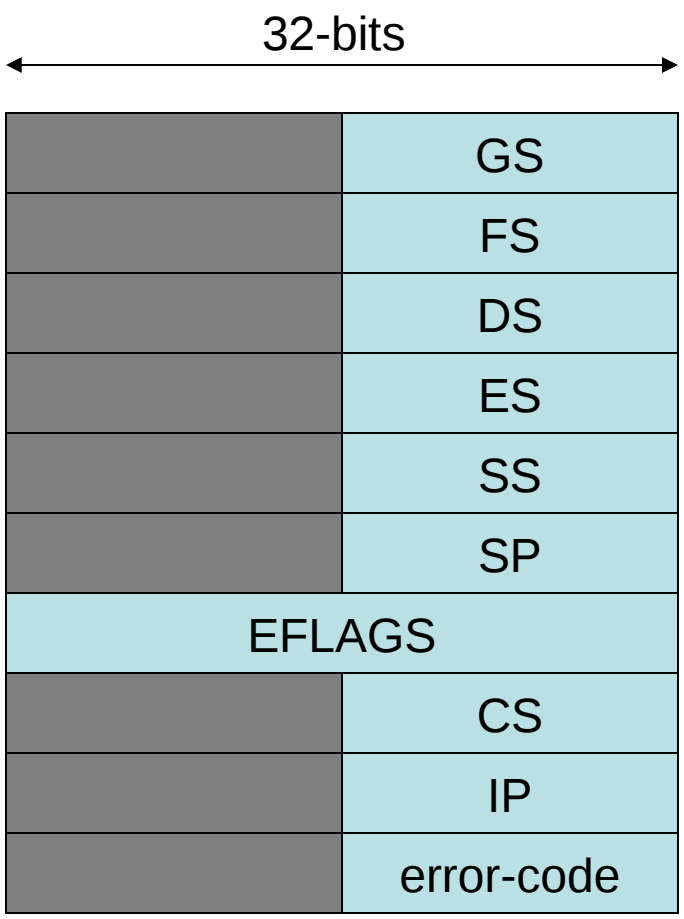
The ‘hlt’ instruction

- In ‘native’ real-mode the ‘hlt’ instruction is used to halt the CPU’s fetch-execute cycle
- But in a multiuser/multitasking system, it wouldn’t be appropriate to allow one task to stop the processor from doing any work
- So in protected-mode the ‘hlt’ instruction is ‘privileged’ – it will trigger an exception if the CPU isn’t executing at ring-0

Our exception-handler

- When our guest-task encounters the ‘hlt’ instruction, it won’t be executed – instead the CPU will switch from ring3 to ring 0 to execute an exception-handling procedure
- We need to write that handler’s code, we need to install an interrupt-gate that will direct the CPU to that code, and we need to put a ring0 stack-address in the TSS

The stack-frame layout



ring0 stackframe

When a general protection exception occurs in Virtual-8086 mode, the CPU automatically switches stacks, then it pushes nine register-values, plus an ‘error-code’, onto the new ring0 stack and transfers control to a procedure whose address it finds in the IDT’s ‘gate’ descriptor for Interrupt-0x0D

We can write an exception-handler that will perform a ‘VM Exit’ from our Guest-task to our Host VM Manager (for example, by using ‘VMCALL’)

Error-Code's format

- Normally the 'error-code' contains useful information about what caused the 'fault'

- But when a privileged instruction was the fault's cause, this error-code will be zero (as the instruction's address will be there)

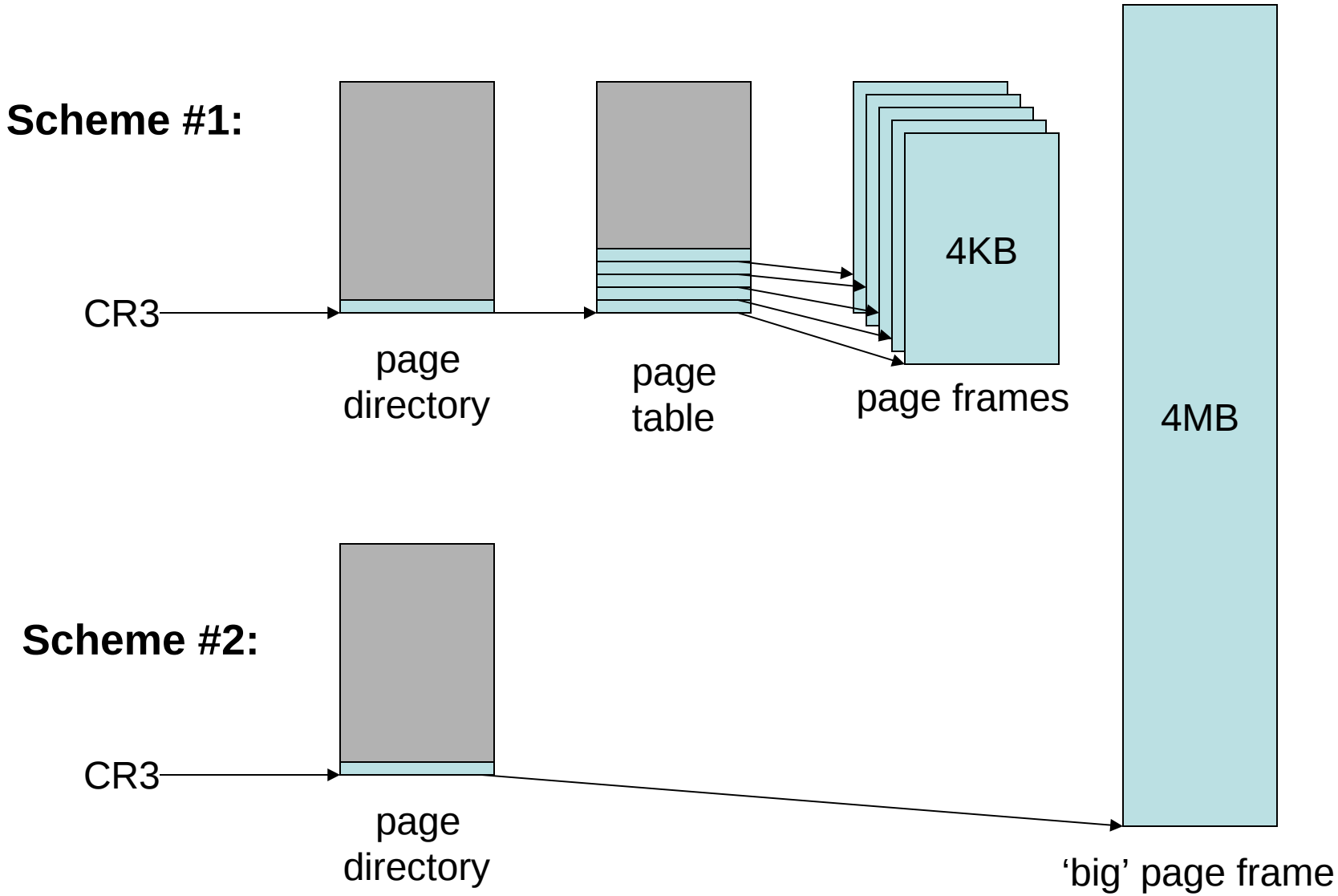
Segment-descriptors

- Our guest-task need a Global Descriptor for its mandatory Task-State Segment
- It may optionally need a Global Descriptor for its Local Descriptor Table (if used)
- It will definitely require code-segment and data-segment descriptors (for ring0 use)
- It will NOT need descriptors its code and data in ring3 (‘real-mode’ addresses used)

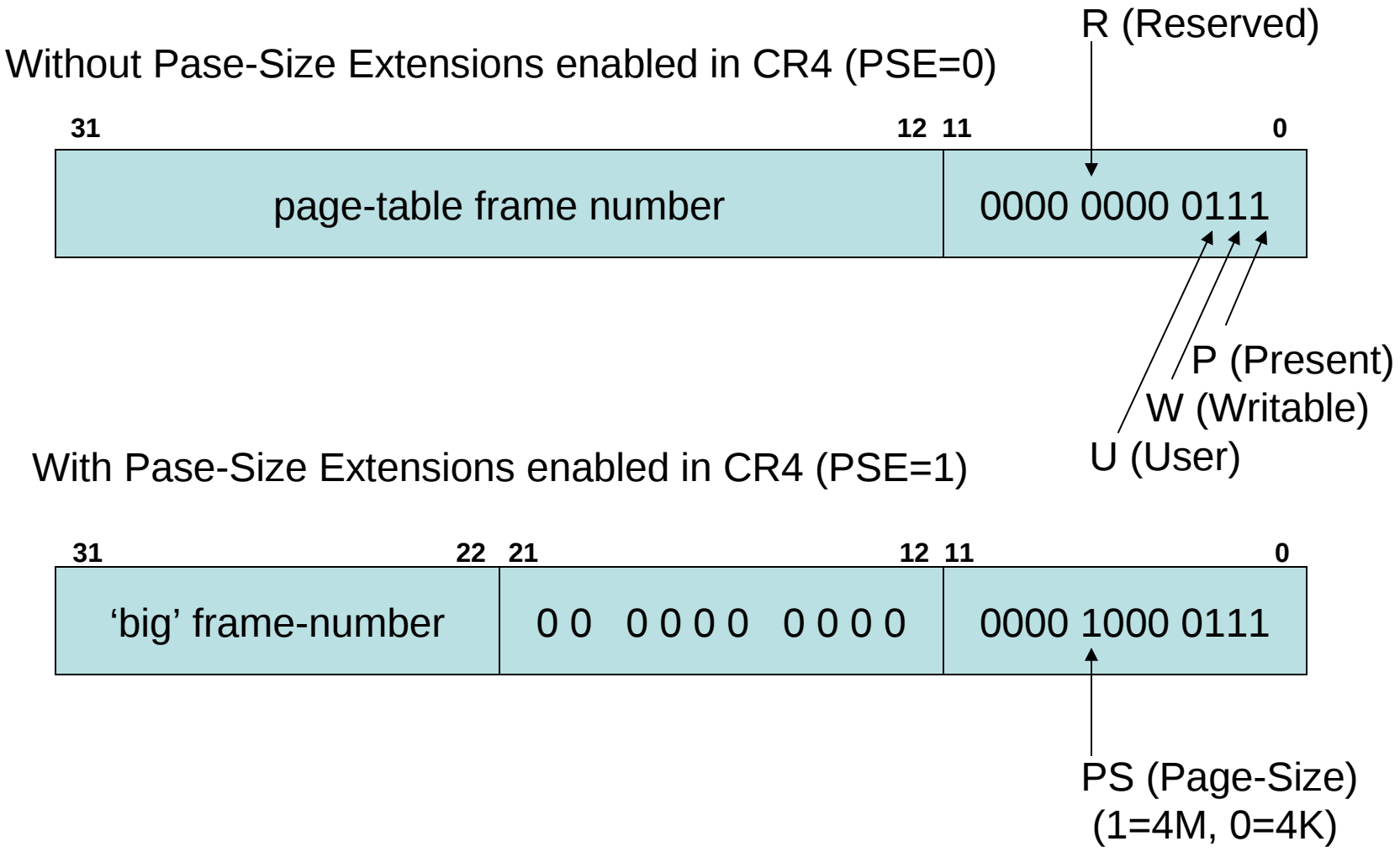
Page-mapping Tables

- Our guest-task needs to use page-tables that the processor understands in 32-bit protected-mode (and Virtual-8086 mode)
- For this we have several options
  - Strict emulation of the 8086's one-megabyte address-space will require us to construct a Page-Table and a Page-Directory
  - Simpler solution, using Page-Size Extensions, would require only a Page-Directory table

Page-mapping alternatives



Directory-entry formats



Summary

- Step-by-step approach focuses first on the Virtual Machine guest-task
- We need these guest data-structures:
  - A Page-Directory table
  - A Task-State Segment
  - A GDT and an IDT (and maybe an LDT)
  - A ring-3 stack-area and a ring-0 stack-area
- We need these executable procedures:
  - The guest-task’s ring-3 routine (.code16)
  - The guest-task’s fault-handler (.code32)

In-class exercise

- Modify our ‘vm86demo.s’ program so that it includes the code needed to transmit a message-string via the serial null-modem cable. (You will also need to enlarge the TSS to include an I/O permission bitmap.)
- You can use the ‘rxrender.cpp’ program to test your changes on our classroom PCs

- Next: Design our ‘Host’ and VMX Controls