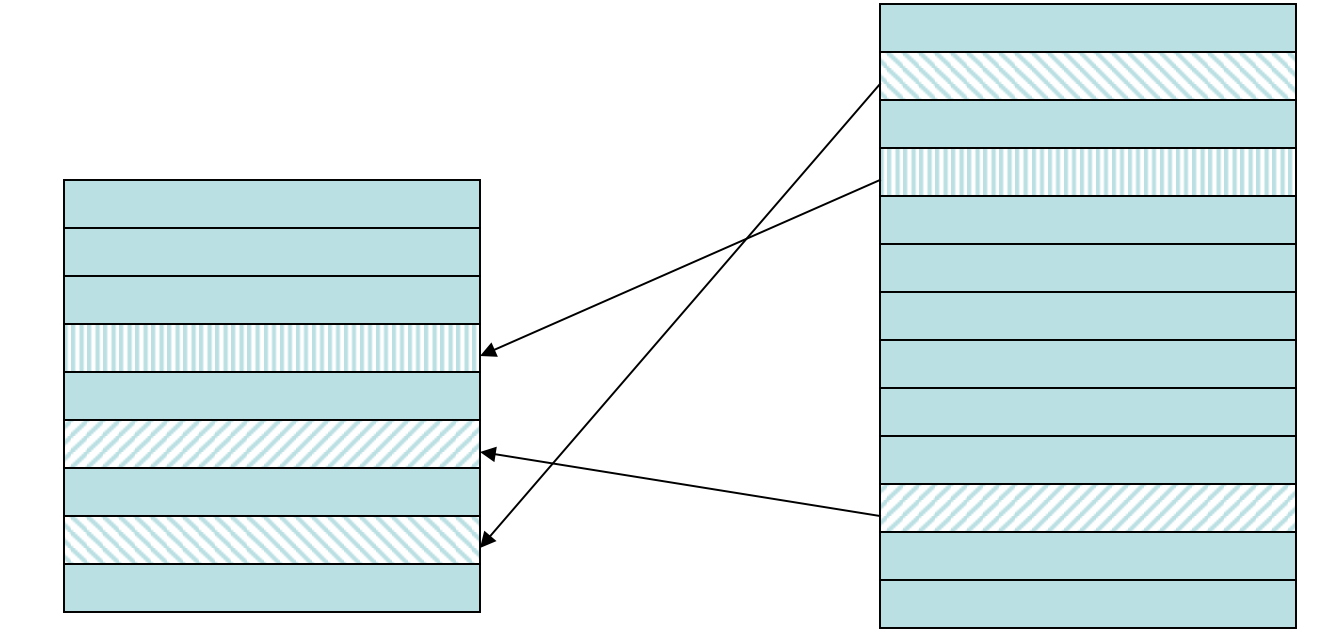
# IA32 Paging Scheme

Introduction to the Intel x86's support for "virtual" memory

# What is ‘paging’?

- It’s a scheme for dynamically remapping addresses for fixed-size memory-blocks

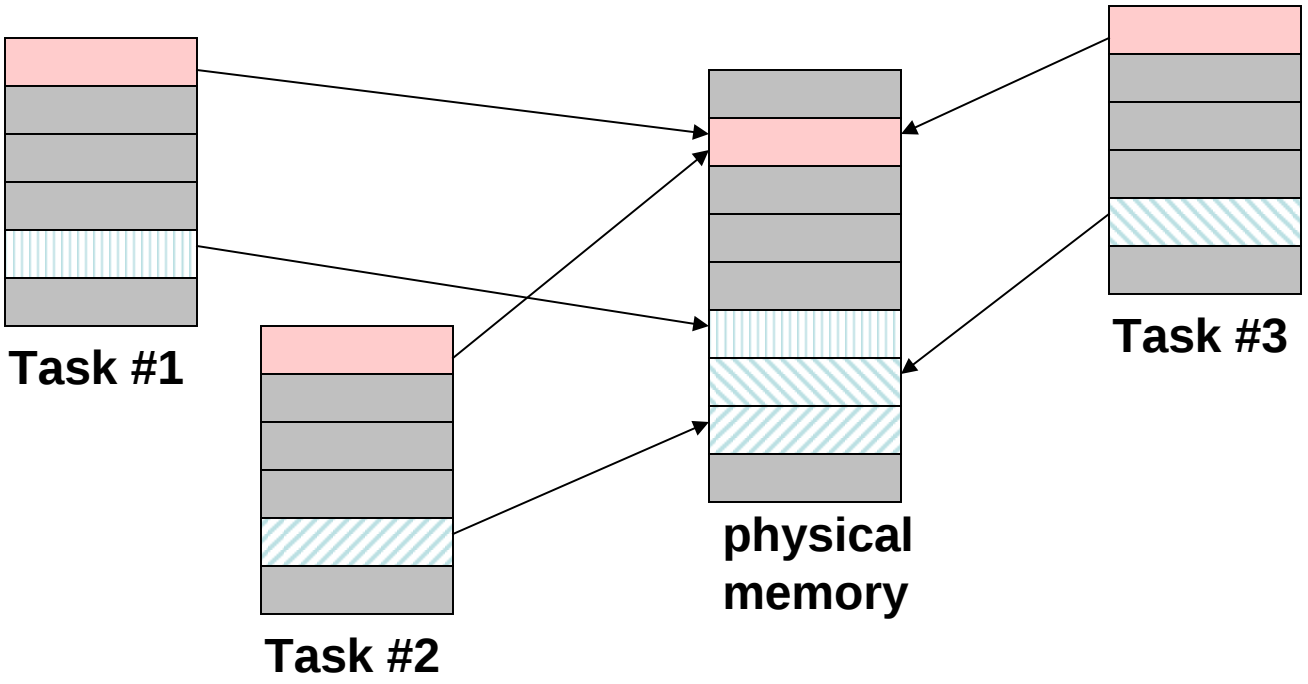


# What’s ‘paging’ good for?

- For efficient ‘time-sharing’ among multiple tasks, an operating system needs to have several programs residing in main memory at the same time
- To accomplish this using actual physical memory-addressing would require doing address-relocation calculations each time a program was loaded (to avoid conflicting with any addresses already being used)

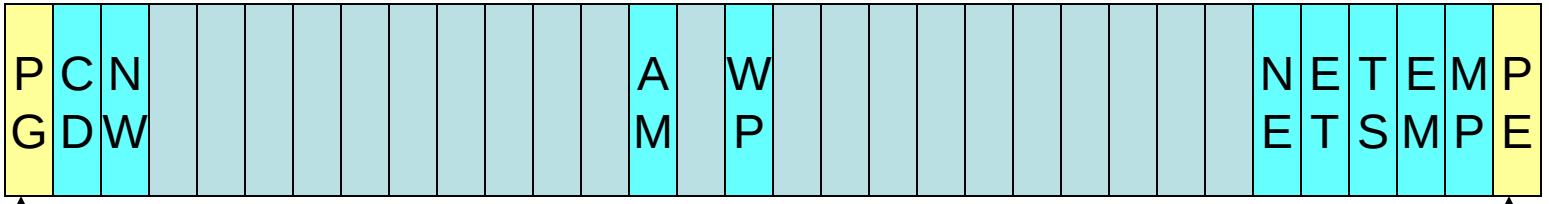
# Why use ‘paging’?

- Use of ‘paging’ allows ‘relocations’ to be done just once (by the linker), and every program can ‘reuse’ the same addresses



# How to enable paging

## Control Register CR0




**Protected-Mode must be enabled (PE=1)**

**Then ‘Paging’ can be enabled (set PG=1)**

```
# Here is how you can enable paging (if already in protected-mode)
        mov     %cr0, %eax      # get current machine status
        bts     $31, %eax       # turn on the PE-bit's image
        mov     %eax, %cr0      # put modified status in CR0
        jmp     pg              # now flush the prefetch queue
pg:
# but you had better prepare your ‘mapping-tables’ beforehand!
```

# Several ‘paging’ schemes

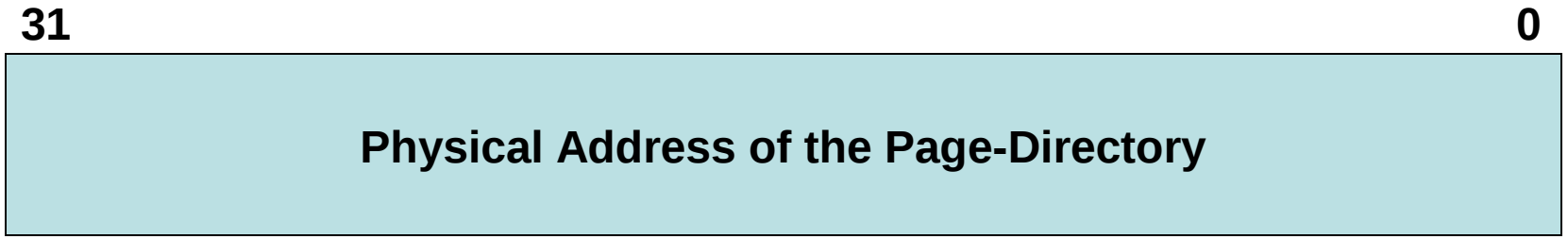
- Intel’s design for ‘paging’ has continued to evolve since its introduction in 80386 CPU
- New processors support the initial design, as well as several optional extensions
- We shall describe the initial design which is simplest and remains as the ‘default’
- It is based on subdividing the entire 4GB virtual address-space into 4KB blocks

# Terminology

- The 4KB memory-blocks are called ‘page frames’ -- and they are non-overlapping
- Therefore each page-frame begins at a memory-address which is a multiple of 4K
- Remember: $4K = 4 \times 1024 = 4096 = 2^{12}$
- So the address of any page-frame will have its lowest 12-bits equal to zeros
- Example: page six begins at 0x00006000

# Control Register CR3

- Register CR3 is used by the CPU to find the tables in memory which will define the address-translation that it should employ

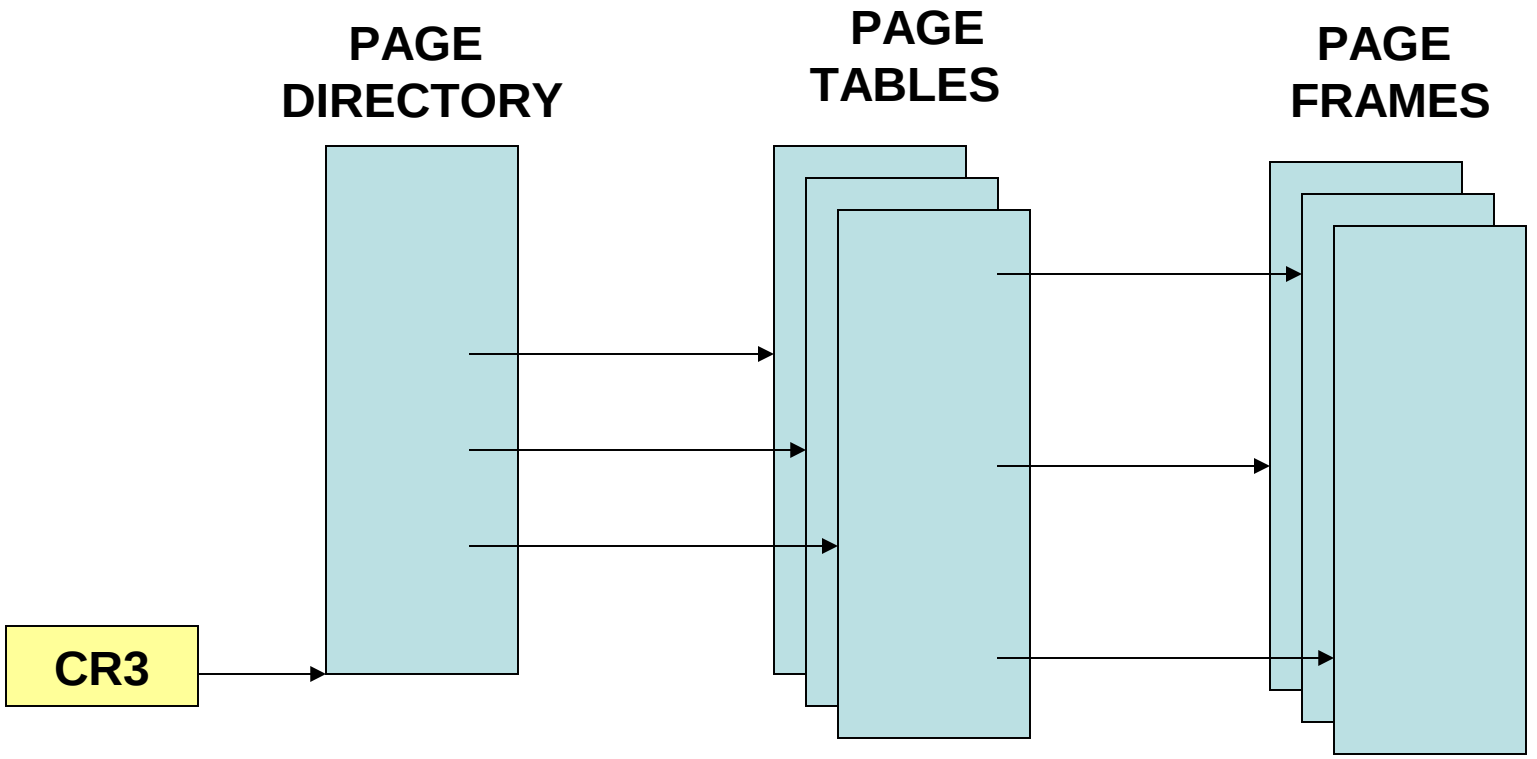



- This table is called the ‘Page Directory’ and its address must be ‘page-aligned’

# Page-Directory

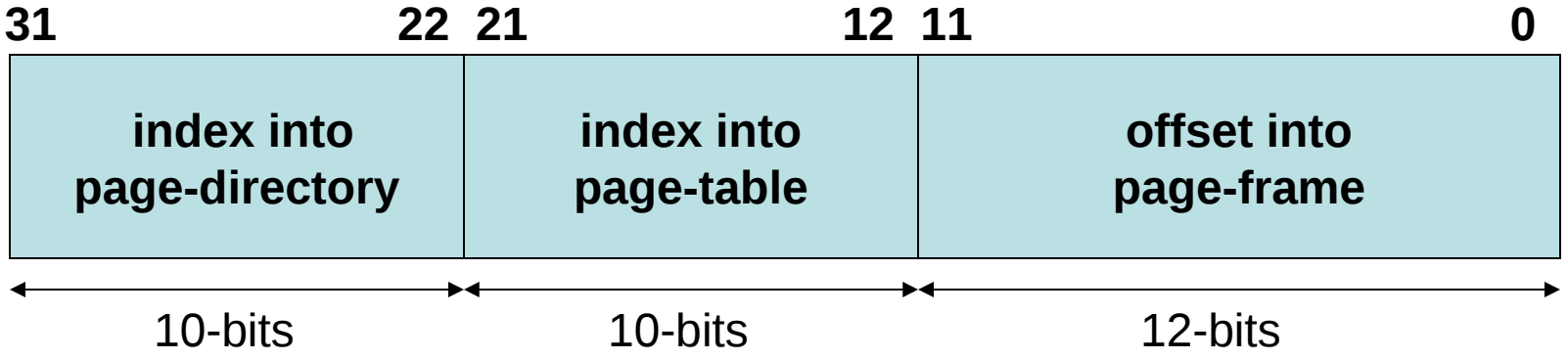
- The Page-Directory occupies one frame, so it has room for 1024 4-byte entries
- Each page-directory entry may contain a pointer to a further data-structure, called a Page-Table (also page-aligned 4KB size)
- Each Page-Table occupies one frame and has enough room for 1024 4-byte entries
- Page-Table entries may contain pointers

# Two-Level Translation Scheme



# Address-translation

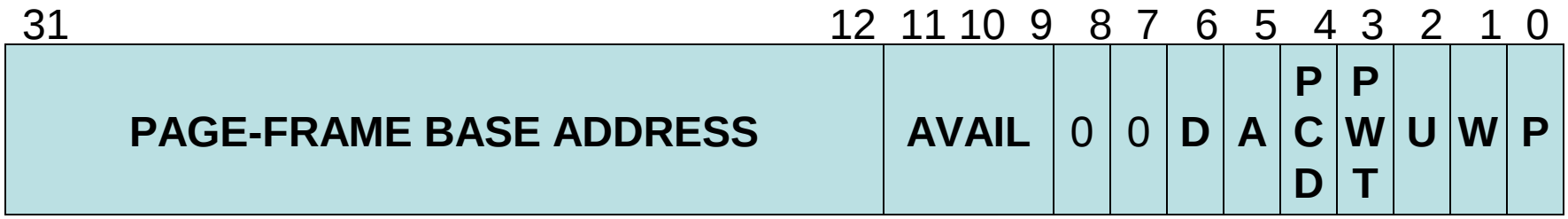
- The CPU examines any virtual address it encounters, subdividing it into three fields



# Page-Level ‘protection’

- Each entry in a Page-Table can assign a collection of ‘attributes’ to the Page-Frame that it points to; for example:
  - The P-bit (page is ‘present’) can be used by the operating system to support its implementation of “demand paging”
  - The W/R-bit can be used to mark a page as ‘Writable’ or as ‘Read-Only’
  - The U/S-bit can be used to mark a page as ‘User accessible’ or as ‘Supervisor-Only’

# Format of a Page-Table entry

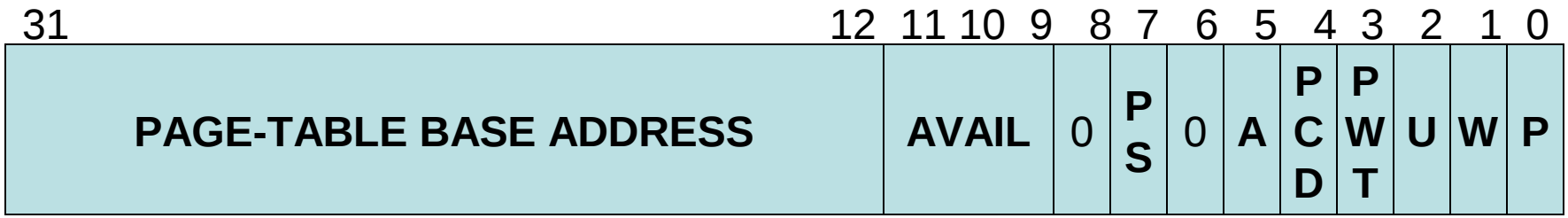



**LEGEND**

**P = Present (1=yes, 0=no)**
**W = Writable (1 = yes, 0 = no)**
**U = User (1 = yes, 0 = no)**
**A = Accessed (1 = yes, 0 = no)**
**D = Dirty (1 = yes, 0 = no)**

**PWT = Page Write-Through (1=yes, 0 = no)**
**PCD = Page Cache-Disable (1 = yes, 0 = no)**

# Format of a Page-Directory entry




**LEGEND**

**P = Present (1=yes, 0=no)**

**W = Writable (1 = yes, 0 = no)**

**U = User (1 = yes, 0 = no)**

**A = Accessed (1 = yes, 0 = no)**

**PS = Page-Size (0=4KB, 1 = 4MB)**

**PWT = Page Write-Through (1=yes, 0 = no)**

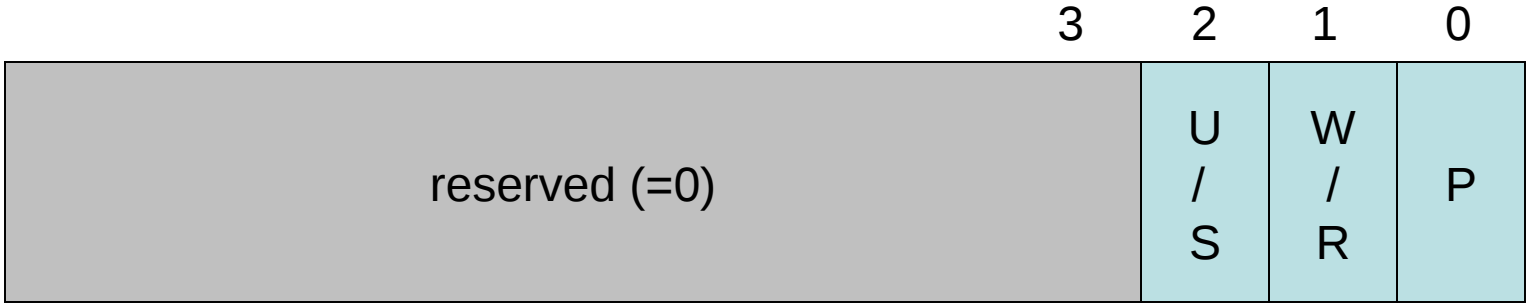
**PCD = Page Cache-Disable (1 = yes, 0 = no)**

# Violations

- When a task violates the page-attributes of any Page-Frame, the CPU will generate a ‘Page-Fault’ Exception (interrupt 0x0E)
- Then the operating system’s page-fault exception-handler gets control and can take whatever action it deems is suitable
- The CPU will provide help to the OS in determining why a Page-Fault occurred

# The Error-Code format

- The CPU will push an Error-Code onto the operating system's stack




**Legend:**

P (Present): 1=attempted to access a 'not-present' page
W/R (Write/Read): 1=attempted to write to a 'read-only' page
U/S (User/Supervisor): 1=user attempted to access a 'supervisor' page

'User' means that CPL = 3; 'Supervisor' means that CPL = 0, 1, or 2

# Control Register CR2

- Whenever a ‘Page-Fault’ is encountered, the CPU will save the virtual-address that caused that fault into the CR2 register
  - If the CPU was trying to modify the value of an operand in a ‘read-only’ page, then that operand’s virtual address is written into CR2
  - If the CPU was trying to read the value of an operand in a supervisor-only page (or was trying to fetch-and-execute an instruction) while CPL=3, the relevant virtual address will be written into CR2

# Identity-mapping

- When the CPU first turns on the ‘paging’ capability, it must be executing code from an ‘identity-mapped’ page (or it crashes!)
- We have created a demo-program that shows how to create the Page-Directory and Page-Tables for an identity-mapping of the lowest 24 page-frames of RAM
- But it maps the 25th page-frame to VRAM

# Using a ‘repeat-macro’

- Our assembler’s macro-capability can be used to build an ‘identity-map’ page-table

```
# Our linker-script offers us an advantage if we use the ‘.data’ section here
            .section  .data
            .align    0x1000
pgtbl:      entry = 0                           # the initial physical address
            .rept     24                        # we define 24 table-entries
            .long     entry + 0x003             # ‘present’ and ‘writable’
            entry = entry + 0x1000              # advance physical address
            .endr                               # conclusion of this macro

            .align    0x1000
pgdir:      .long     pgtbl + 0x10000 + 0x003   # first page-directory entry
            .align    0x1000
```

# Our ‘fileview’ utility

- One advantage of constructing paging-tables at assembly-time is that we can then examine the resulting table-entries with our ‘fileview’ utility

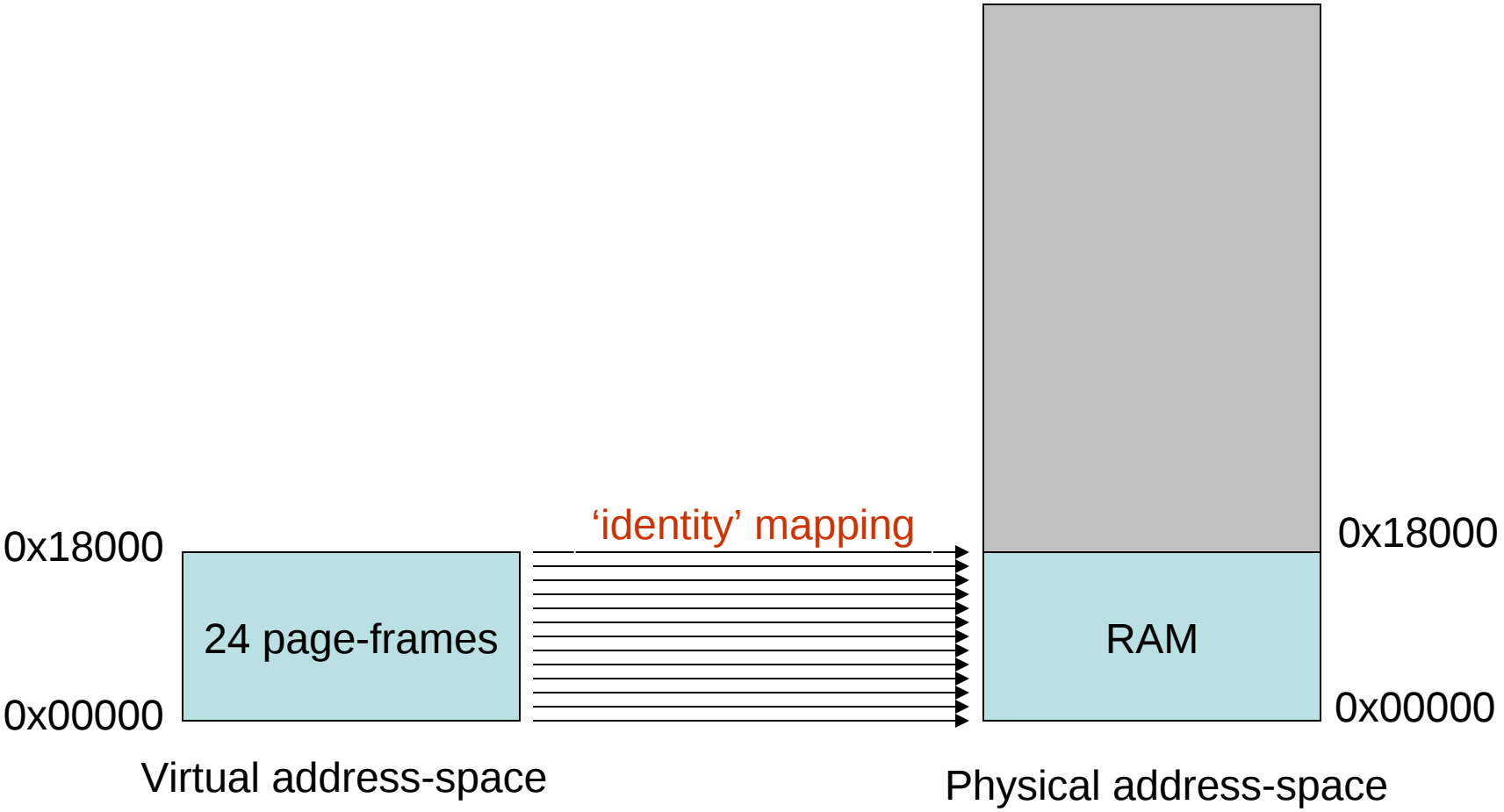
- For a demo-program named ‘pagedemo.s’:

```
$ as  pagedemo.s  –o pagedemo.o
$ ld  pagedemo.o  -T ldscript  -o pagedemo.b
```

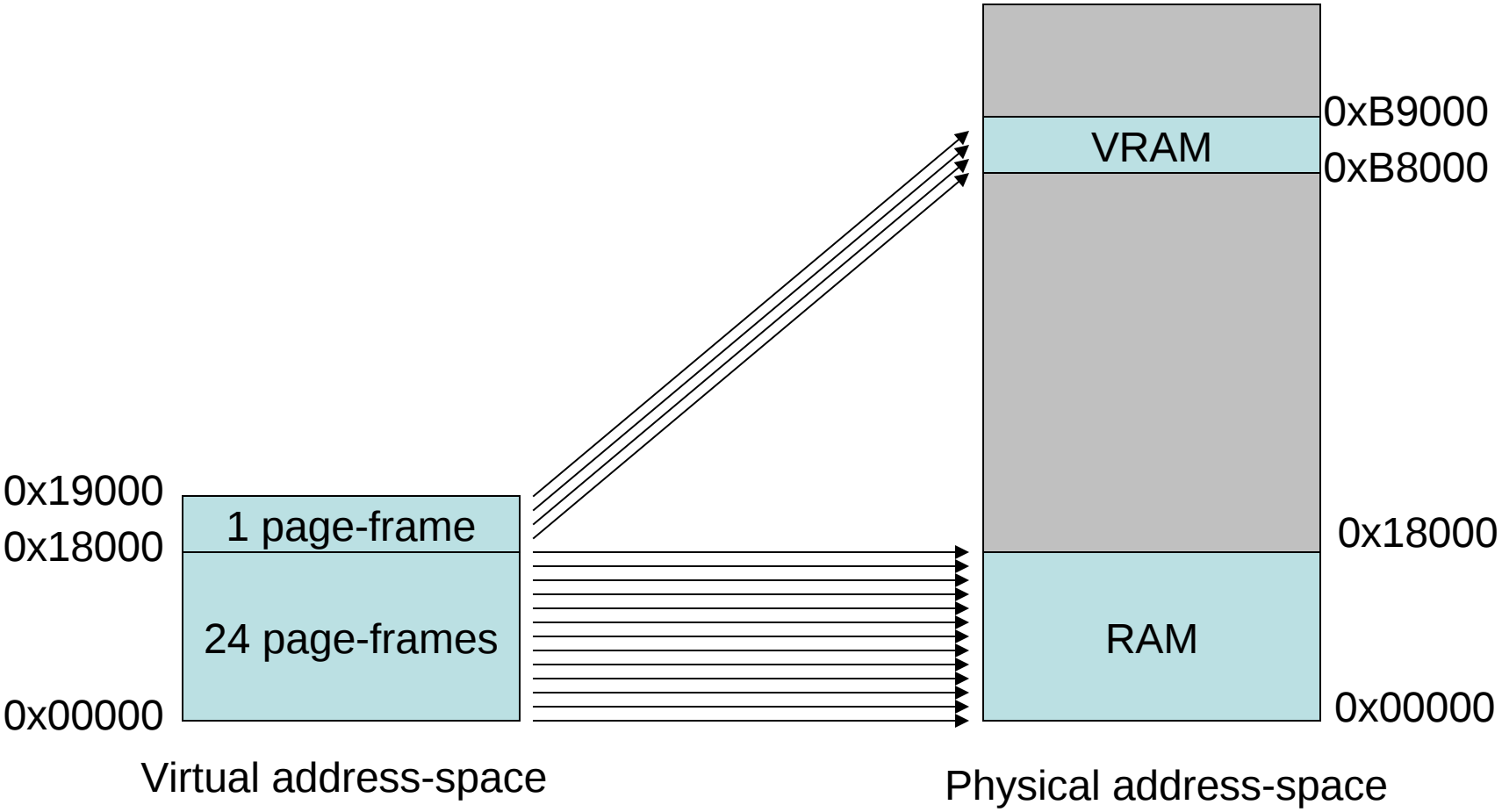
- Examine the executable ‘pagedemo.b’ file:

```
$ ./fileview  pagedemo.b
```

# Depiction of identity-mapping

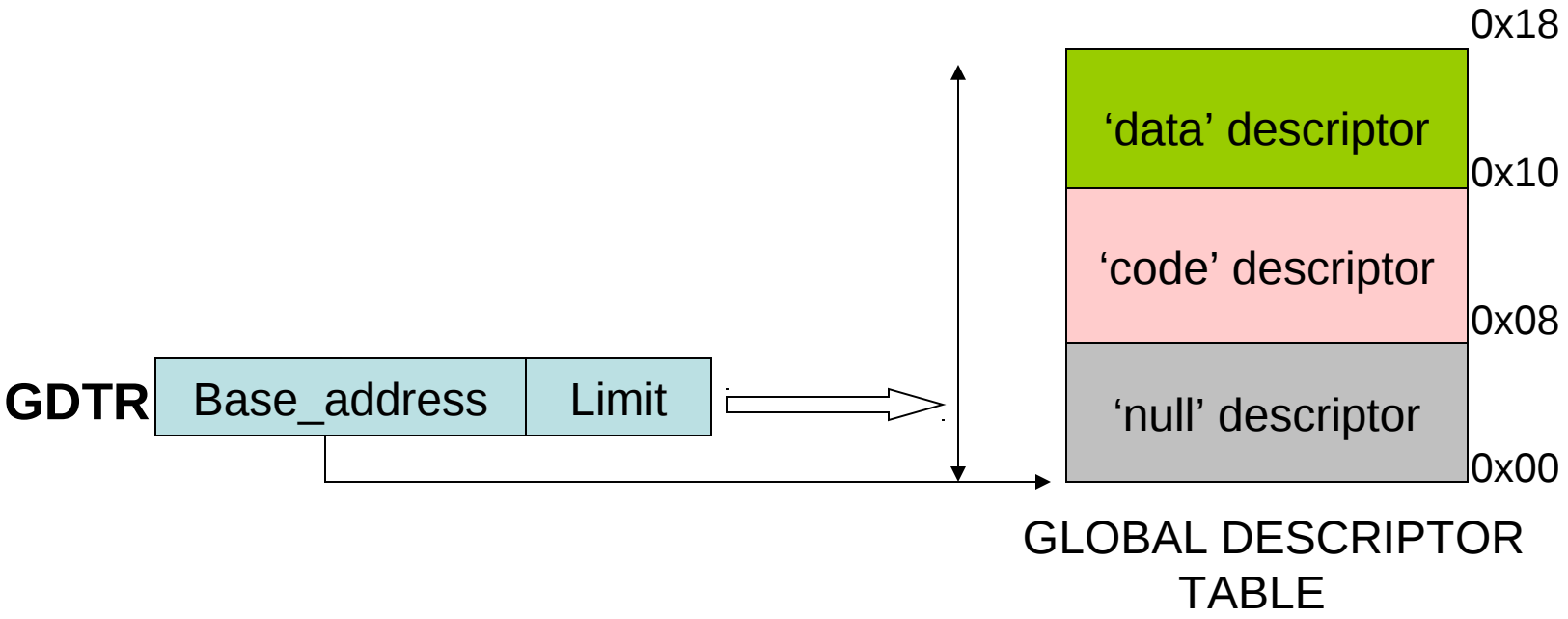


# Mapping used in ‘pagedemo.s’

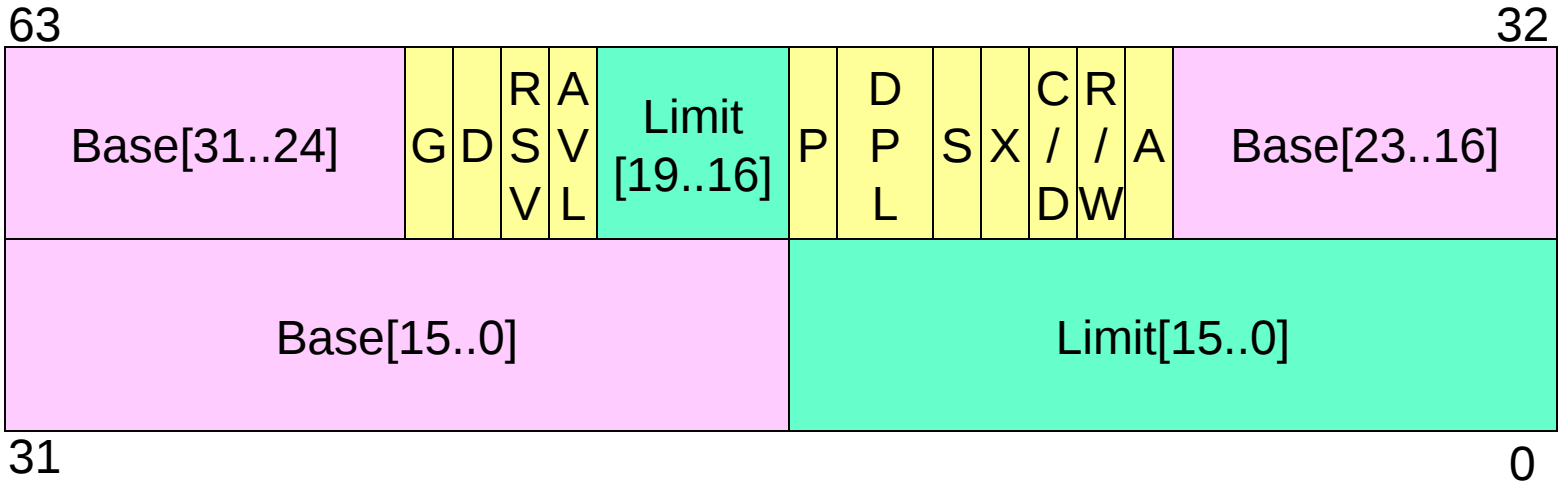


# ‘paging’ requires protected-mode

- Memory-addressing in protected-mode is performed using ‘segment-descriptors’
- Register GDTR holds a pseudo-descriptor



# Segment-Descriptor Format




**Legend:**

G = Granularity (0 = byte, 1 = 4KB-page)
D = Default size (0 = 16-bit, 1 = 32-bit)
X = eXecutable (0 = no, 1 = yes)

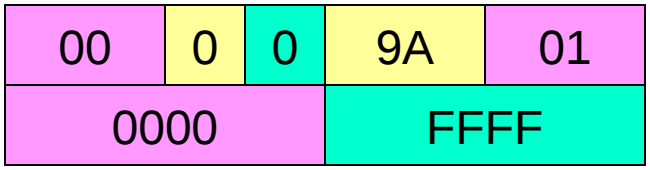
DPL = Descriptor Privilege Level (0..3)
P = Present (0 = no, 1 = yes)
S = System (0 = yes, 1 = no)
A = Accessed (0 = no, 1 = yes)

*code-segments:* R = Readable (0 = no, 1 = yes) C = Conforming (0=no, 1=yes)
*data-segments:* W = Writable (0 = no, 1 = yes) D = expands-Down (0=no, 1=yes)

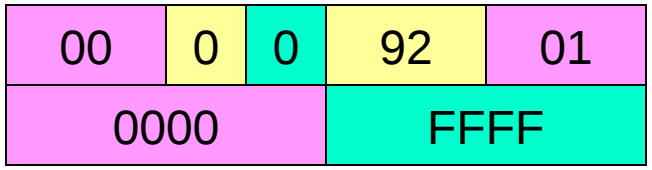
RSV = Reserved for future use by Intel

AVL = Available for user's purposes

# Descriptor Implementations




‘code’ descriptor

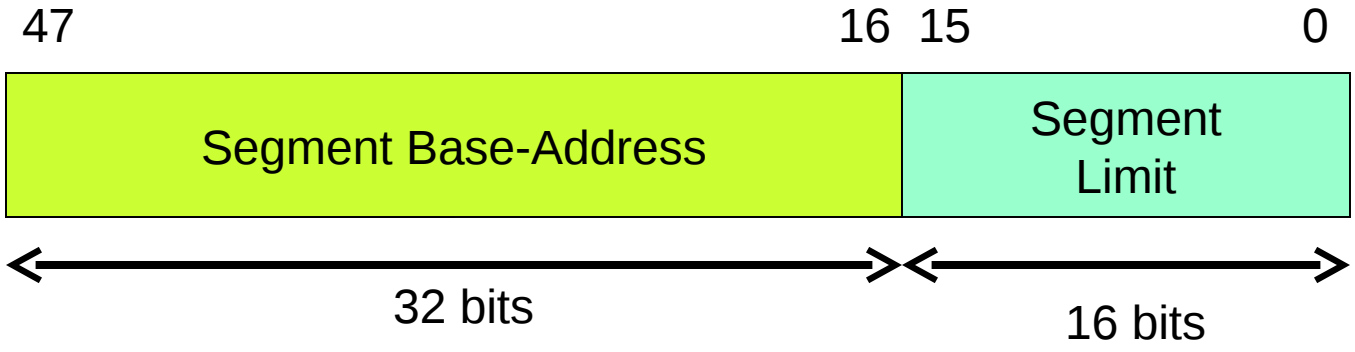



‘data’ descriptor

```
# segment-descriptor for ‘executable’ 64KB segment
.word      0xFFFF, 0x0000, 0x9A01, 0x0000


# segment-descriptor for ‘writable’ 64KB segment
.word      0xFFFF, 0x0000, 0x9201, 0x0000
```

# GDTR register-format




The register-image (48-bits) is prepared in a memory-location…

```
regGDT: .word     0x0017, theGDT, 0x0001   # register-image for GDTR
```

… then the register gets loaded from memory via a special instruction

```
        lgdt      regGDT                   # initializes register GDTR
```

# In-class exercise #1

- Can you modify our ‘pagedemo.s’ program so that the first page-frame of the video display memory (physical address-range starts from 0xB8000) will appear to start from the virtual-address 0x19000 ?