# *ECE391*
# *Computer System Engineering*
### *Lecture 14*

Dr. Zbigniew Kalbarczyk

University of Illinois at Urbana- Champaign

Spring 2021

# *Lecture Topics*

- Virtual memory motivation
- x86 support for VM
  - protection model
  - segmentation
  - paging

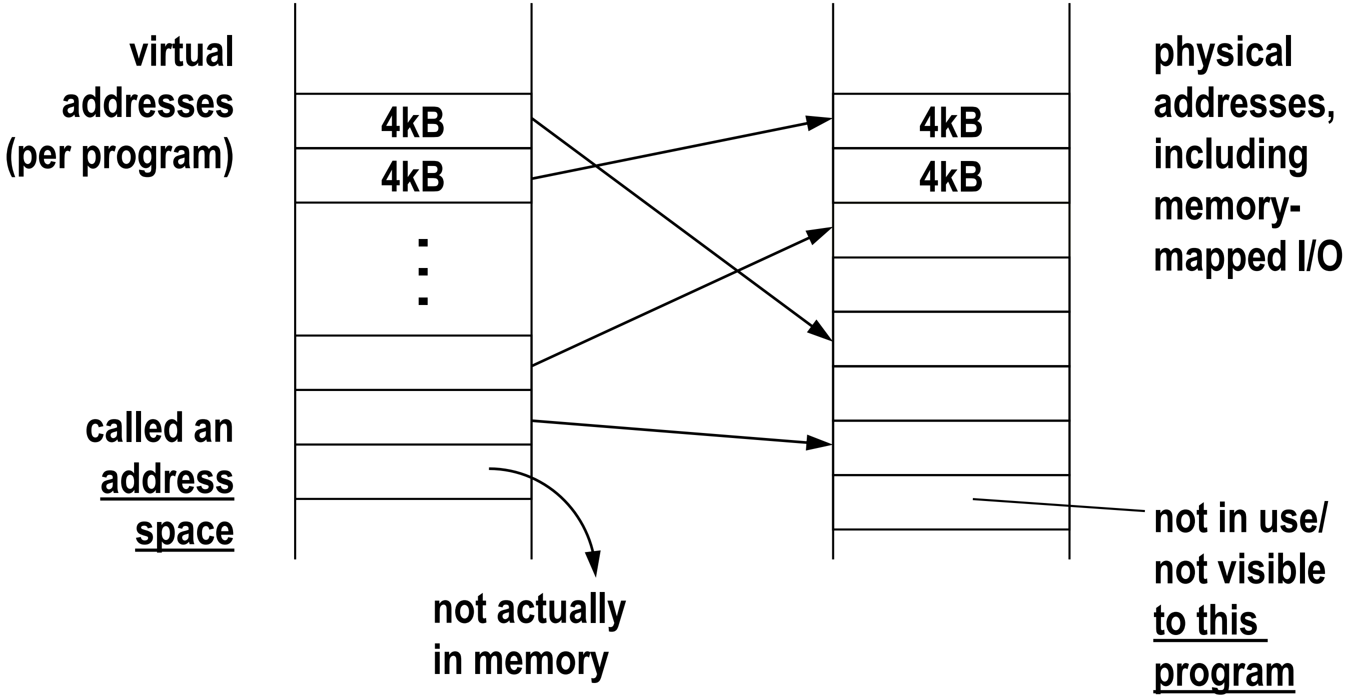
# *Aministrivia*

- **MP2 Checkpoint 2**
  - **Due by 5:59pm Monday, March 15**

- **MP3 Teams**
  - Please submit a google form: https://forms.gle/FfeSuDFjo49VNjtd7 before Thursday 3/11 at 11:59PM CT
  - Every team must have 4 members

# *Virtual Memory Definition*

- What is virtual memory?
  - <u>indirection</u> between memory addresses seen by software and those used by hardware
  - typically done with large blocks, e.g., 4kB  or 4MB in x86



virtual addresses (per program)

called an <u>address space</u>

not actually in memory

physical addresses, including memory-mapped I/O

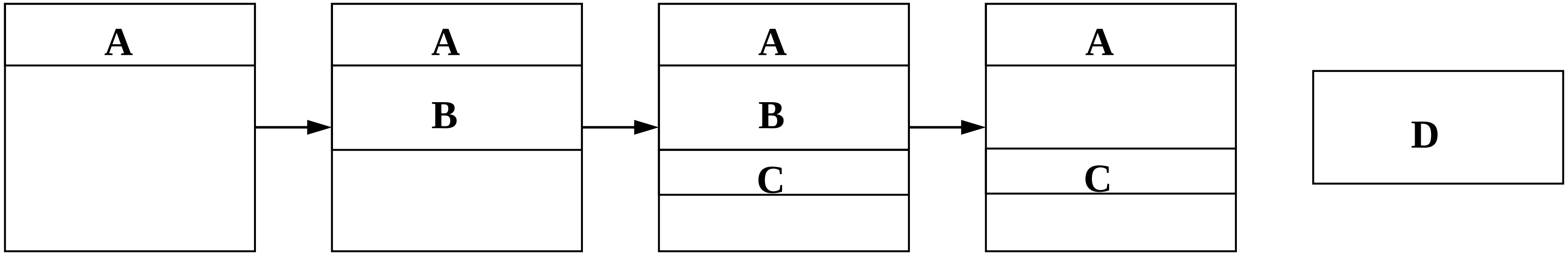
not in use/ not visible <u>to this program</u>

# *Virtual Memory Motivation*

- Why use virtual memory? (in decreasing order of importance)
  - protection
    - one program cannot accidentally or deliberately destroy another’s data
    - the memory is simply not accessible
  - more effective sharing
    - two (or more) programs that share library code can share a single copy of the code in physical memory
    - code and data not actively used by a program
      - can be pushed out to disk
      - to make room for other programs’ active data
      - provides the illusion of a much larger physical memory

# *Virtual Memory Motivation (cont.)*

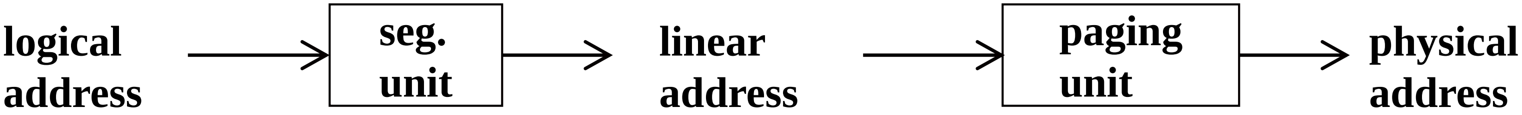
– no fragmentation (little to none)
  - systems without virtual memory suffer fragmentation when try to multitask
  - e.g., if we run A followed by B followed by C, and B finishes, we can't give D a contiguous block of memory, even though it fits in the absolute sense



– simplifies program loading and execution:
  no relocation of code, rewriting stored pointer values, etc.

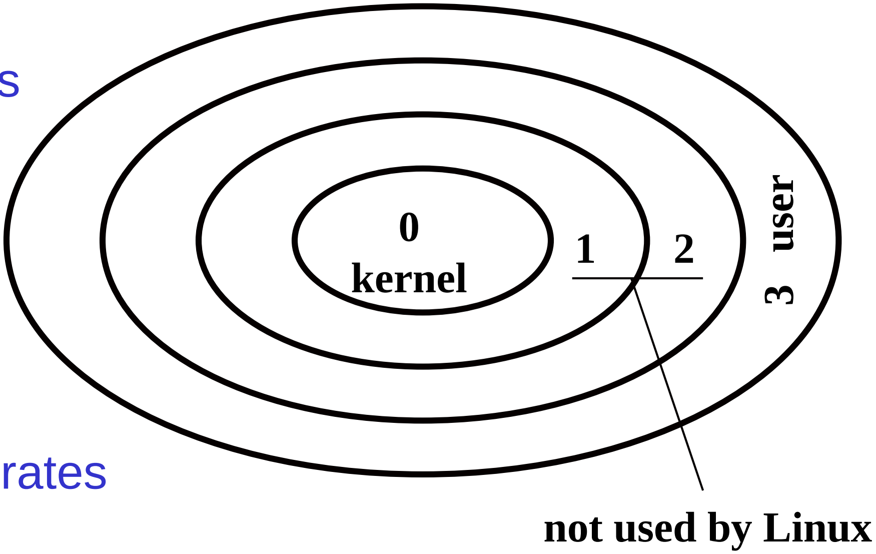
# *x86 Support for VM*

- protection model
- segmentation
- paging



# *x86 Protection Model*

- Four rings: kernel (ring 0) through user (ring 3)
  - lower numbers are more privileged
  - lower numbers never call/trust higher numbers
  - higher numbers call lower numbers only through narrow interfaces (e.g., system calls)
- **CPL** – current privilege level (of executing code)
- **RPL** – requestor's privilege level; when code at high privilege level executes on behalf of code at lower level, some accesses may voluntarily lower privilege to that of caller/beneficiary
- **DPL** – descriptor privilege level; level necessary to execute code/data
- if MAX(CPL,RPL) > DPL, processor generates an exception (general protection fault)



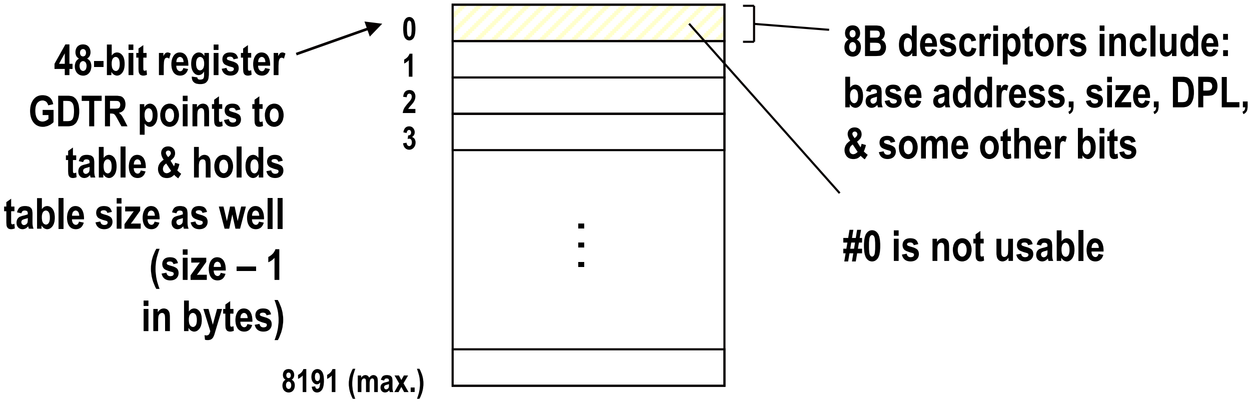
# *x86 Segmentation*

- x86 actually has two levels of indirection
- One is mostly unused…(this one!) => Segmentation
- A segment is a contiguous portion of a linear address space
  - such as the 32-bit space of physical addresses
- x86 in protected mode always uses segmentation

| | | |
|---|---|---|
| code segment | CS | each segment register has 16 bits visible + ~64 bits shadow (not accessible via ISA) that cache the description of the segment # referenced by the visible 16 bits |
| data segment | DS | |
| extra data segment | ES | |
| still more extras (floating point + another) | FS | |
| | GS | |
| stack segment | SS | |

# *x86 Segmentation (cont.)*

- Global descriptor table (GDT)

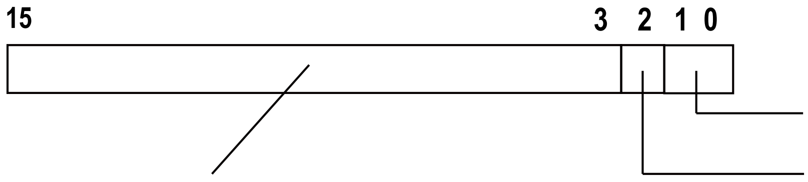


48-bit register GDTR points to table & holds table size as well (size – 1 in bytes)

8B descriptors include: base address, size, DPL, & some other bits

#0 is not usable

# Segment Register Meaning

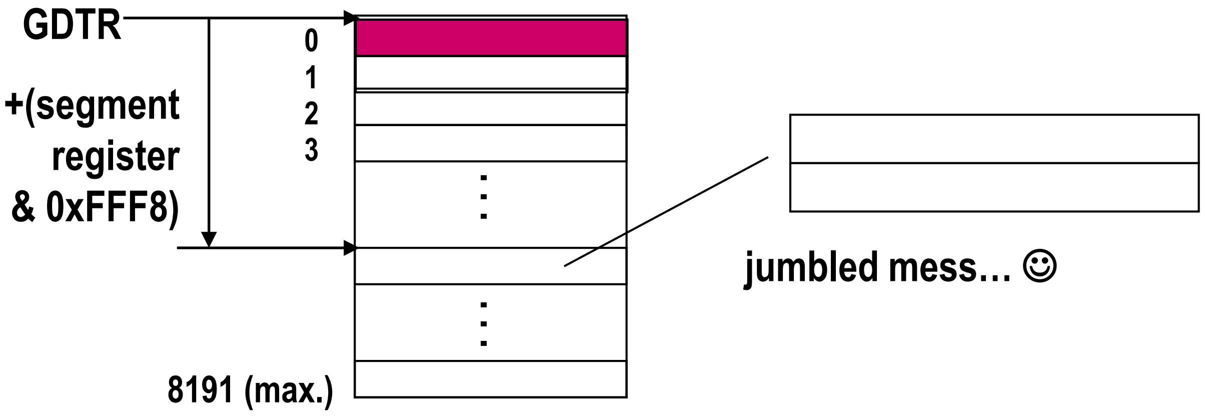


RPL

0 for GDT
1 for Local Descriptor Table (not mentioned yet, & essentially not used by Linux)

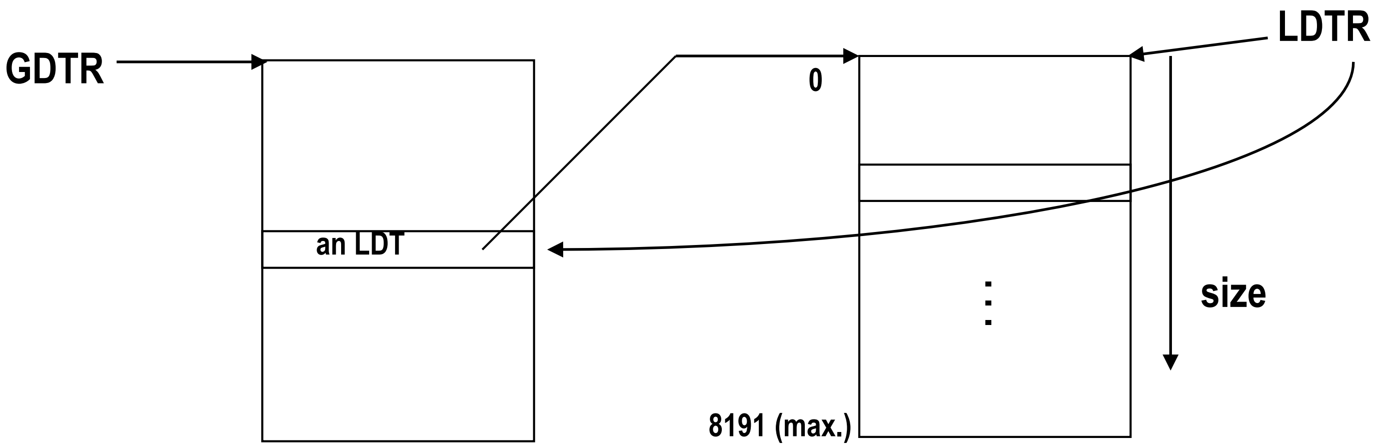
index in table
(0…8191)
or, since table entries are 8B,
offset to find entry

# *Segment Register Meaning (cont.)*



# *Segment Register Meaning (cont.)*

- GDT entries can also describe local descriptor tables (LDTs)
  - LDT originally meant to be per-task segment tables
  - LDTR points to current LDT
    (includes base, size, and index of LDT in GDT)



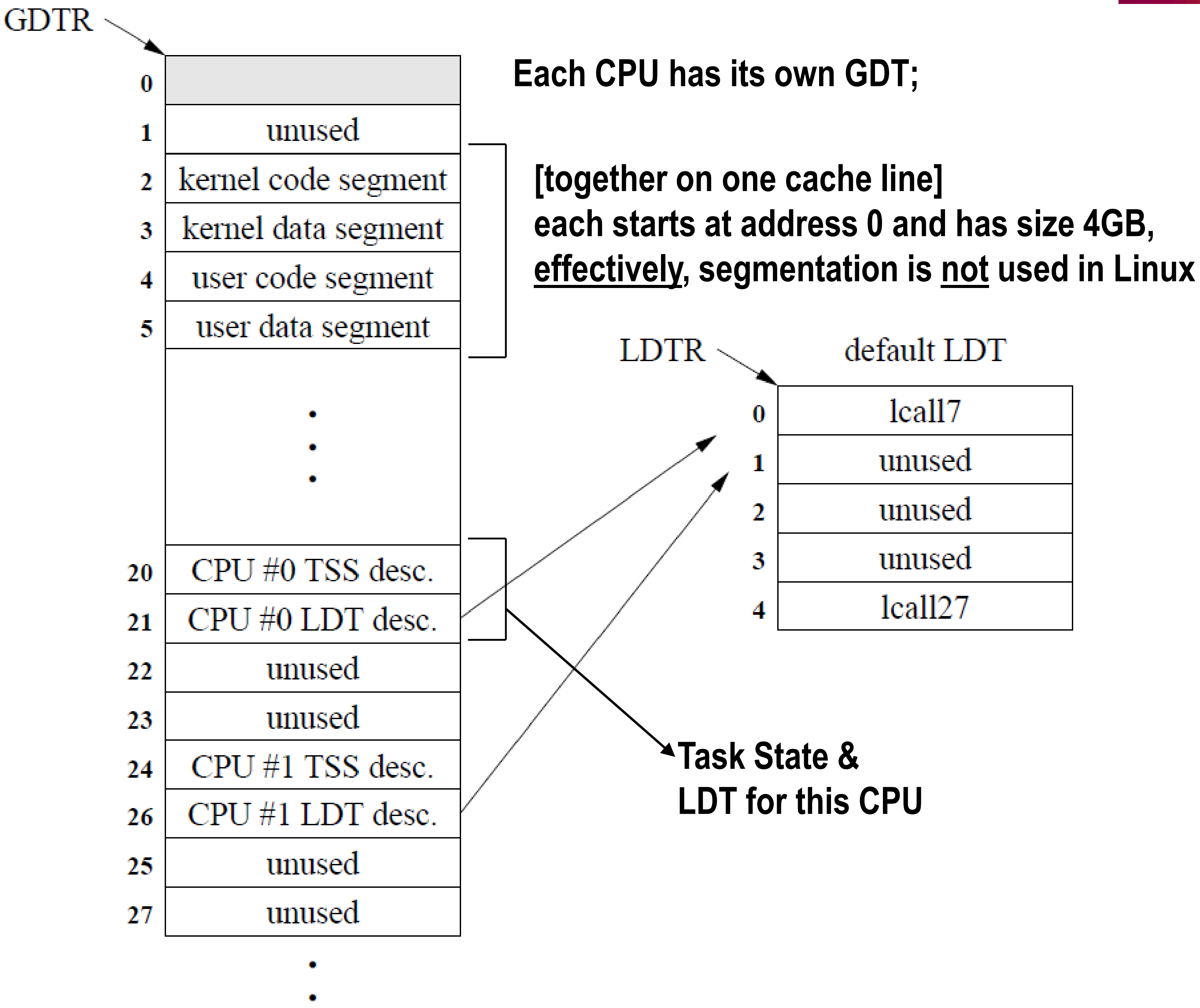
# *Segment Register Meaning (cont.)*

- Descriptors can also differentiate
  - code (executable and possible readable) from data (readable and possibly writable)
  - other useful things

- Also descriptors in the GDT can describe certain aspects of program state (e.g., the task state segment, or TSS)

# *Linux Use of Segmentation*

## *(details in asm/segment.h)*



Each CPU has its own GDT;

[together on one cache line]
each starts at address 0 and has size 4GB,
effectively, segmentation is not used in Linux

Task State &
LDT for this CPU