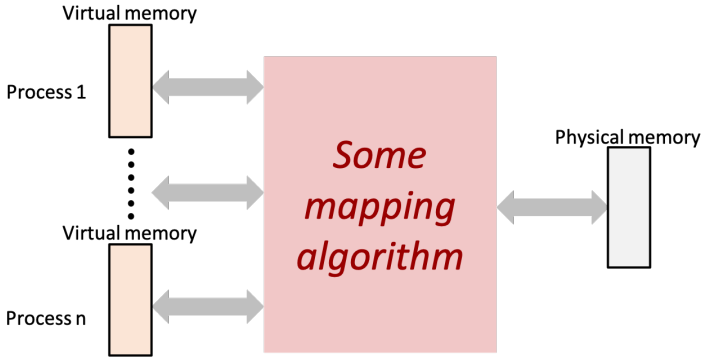
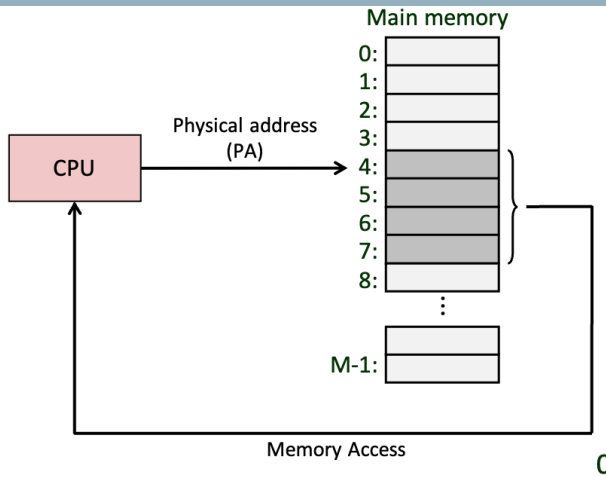
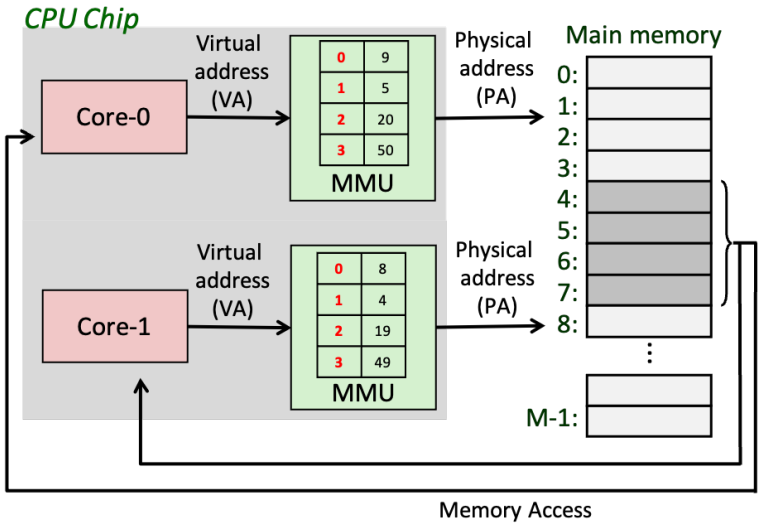
Lecture 16: Segmentation

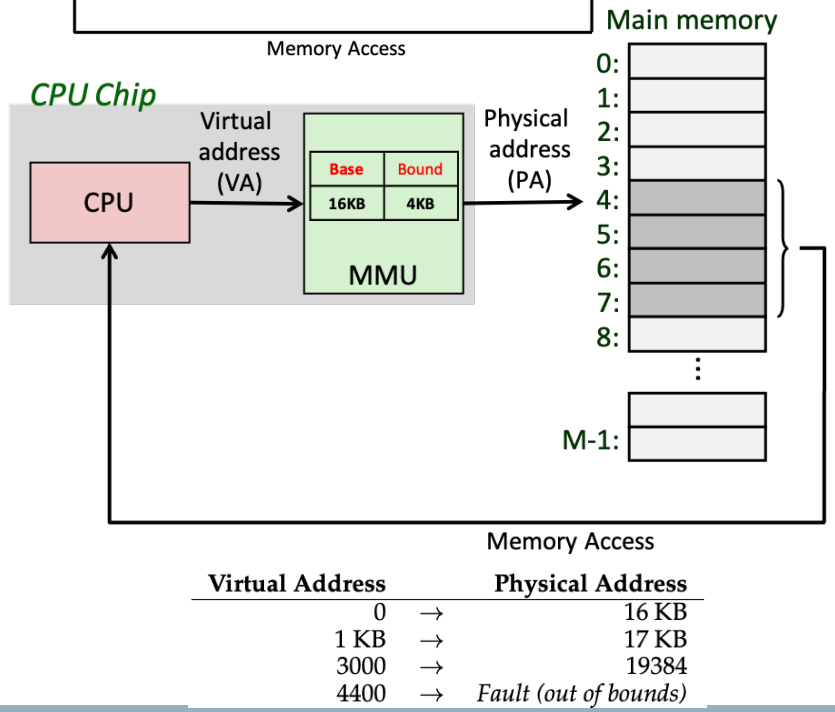
Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture









Virtual Address		Physical Address
0	→	16 KB
1 KB	→	17 KB
3000	→	19384
4400	→	*Fault (out of bounds)*

- We need some kind of mapper between VM-PM as VM (virtual memory) too big than available PM (physical memory)
  - One-to-one mapping
  - Using MMU table
  - Dynamic relocation using base/bound registers
    - Prone to fragmentation

Today's Class

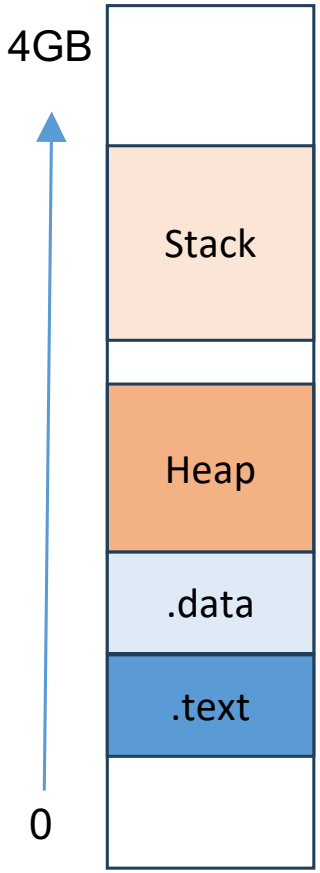
- Segmentation
- Segment registers
- Global Descriptor Table (GDT)

Loader View

```
vivek@possum:~/os23$ readelf -l fib

Elf file type is DYN (Position-Independent Executable file)
Entry point 0x1080
There are 11 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x00000034 0x00000034 0x00160 0x00160 R   0x4
  INTERP         0x000194 0x00000194 0x00000194 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x00000000 0x00000000 0x003f8 0x003f8 R   0x1000
  LOAD           0x001000 0x00001000 0x00001000 0x00278 0x00278 R E 0x1000
  LOAD           0x002000 0x00002000 0x00002000 0x00134 0x00134 R   0x1000
  LOAD           0x002ed4 0x00003ed4 0x00003ed4 0x00134 0x0013c RW  0x1000
  DYNAMIC        0x002edc 0x00003edc 0x00003edc 0x000f8 0x000f8 RW  0x4
  NOTE           0x0001a8 0x000001a8 0x000001a8 0x00044 0x00044 R   0x4
  GNU_EH_FRAME   0x002020 0x00002020 0x00002020 0x0003c 0x0003c R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10
  GNU_RELRO      0x002ed4 0x00003ed4 0x00003ed4 0x0012c 0x0012c R   0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.gnu.build-id .note.ABI-tag .gnu.hash .dynsym .dynst
   03     .init .plt .plt.got .text .fini
   04     .rodata .eh_frame_hdr .eh_frame
   05     .init_array .fini_array .dynamic .got .data .bss
   06     .dynamic
   07     .note.gnu.build-id .note.ABI-tag
   08     .eh_frame_hdr
   09
   10     .init_array .fini_array .dynamic .got
```

- PHDR index 3
  - Starting virtual address is 4096 and total size 632
- PHDR index 4
  - Starting virtual address is 8192 and total size 308
- PHDR index 5
  - Starting virtual address is 16084
- Non-contiguous memory layout across segments (inter-segment)
- Contiguous memory within segments (intra-segment)

ELF Segment Allocations

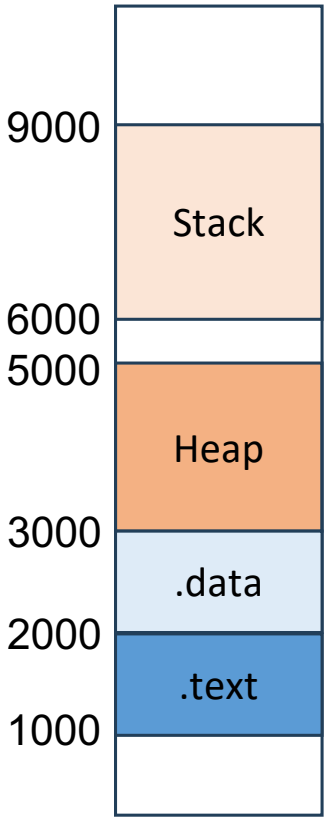


- Exact size for stack and heap size cannot be known upfront
  - Hello world program
    - Minimal sized stack and heap
  - Recursive Fibonacci program
    - Large stack but minimal sized heap
  - Recursive merge sort program
    - Large stack and large heap
- Observations
  - Each segment (stack, heap, data and text) has different memory requirements

Mapping Virtual to Physical Address

- Approaches
  1. Virtual address is same as the physical address
  2. Virtual address is not the same as physical address
     a) Using base & bound for VA to PA mapping
     b) Using segmentation for VA to PA mapping

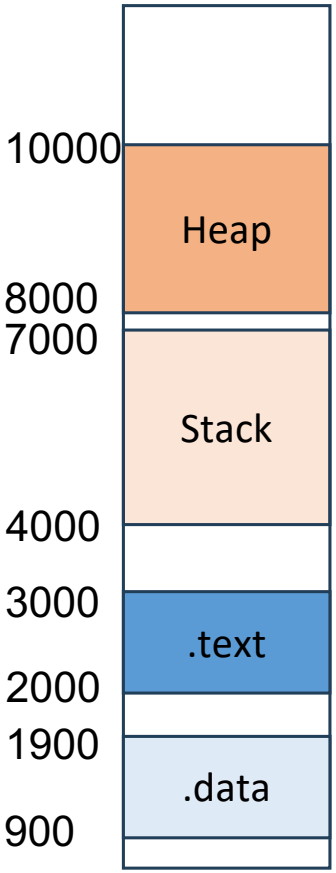
Approach-2: Segment Wise Base/Bound



- Each segment can have its own ("personal") base and bound values
  - **Segmentation**
- Each virtual address can have few bits to specify the segment they belong (e.g., 2 bits to specify 4 different segments), and remaining bits to specify the offset into that segment

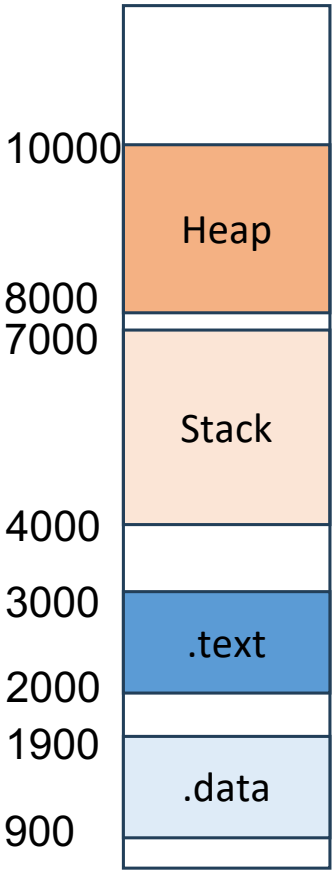
- Example: How to access virtual address 250 (binary: **11111010**)?
  1. Find the segment from the segment selector (e.g., data for first two bits)
  2. Find the offset (111010) = 58
  3. Find the **base** for this segment (Base=2000)
  4. Check if offset within **bounds** (Bound=1000)
  5. Physical address = 2058

Non-Contiguous Inter-Segment Memory



- As the base and bound is separate for each segment, now it is not a requirement to have a contiguous chunk of physical address space for the process's address space
  - Better space management and reduction in fragmentation
- Example: How to access virtual address 250 (binary: **11111010**)?
  1. Find the segment from the segment selector (e.g., data for first two bits)
  2. Find the offset (111010) = 58
  3. Find the **base** for this segment (Base=900)
  4. Check if offset within **bounds** (Bound=1000)
  5. Physical address = **958**

Accessing Segment's Base/Bound

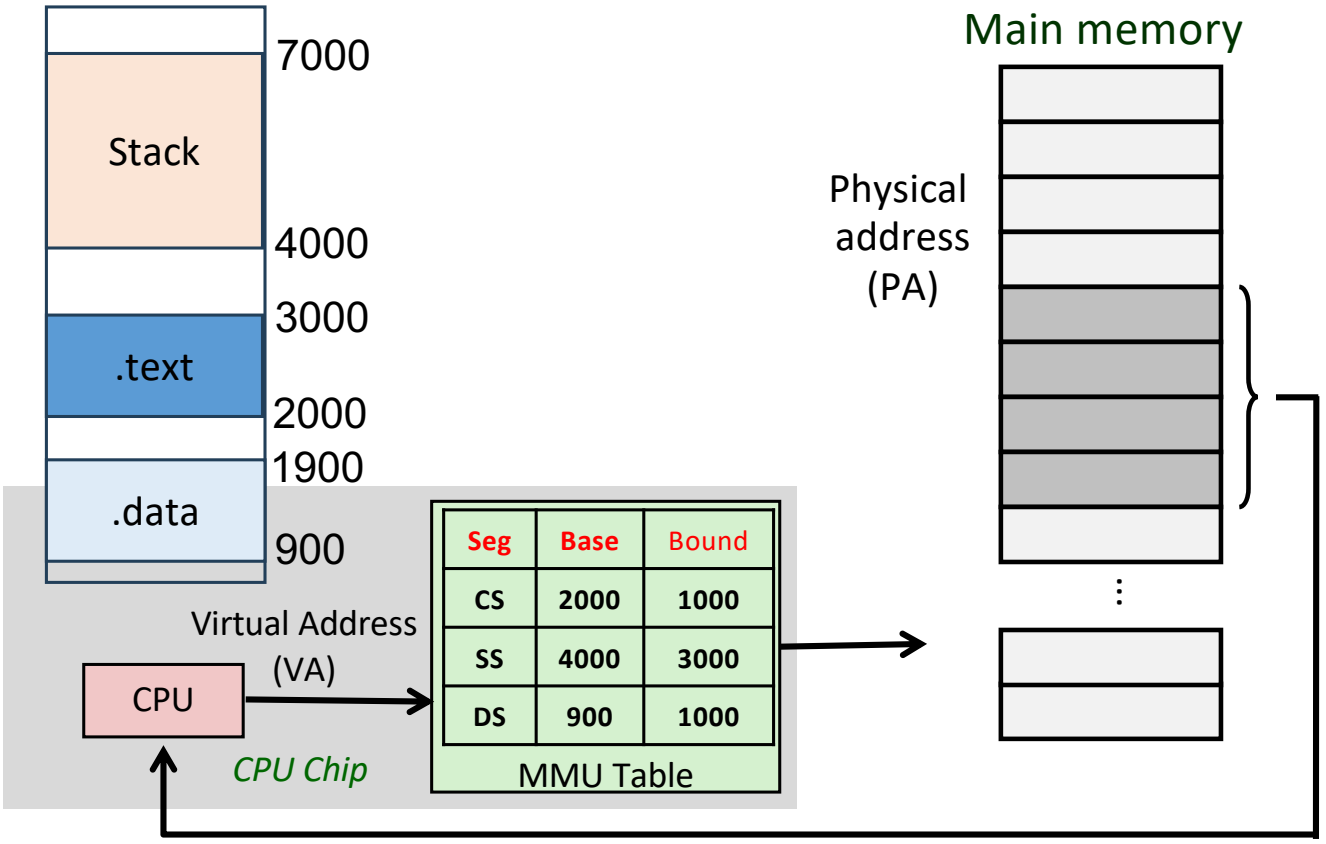


- How to access base and bounds of individual segments?
- Can we store the segments and their base/bound values in a MMU table?
  - Yes, but it would require **iterating** through the MMU table entries [O(n)]
  - Any simple technique?
    - Hardware support by providing dedicated registers on each core

Hardware Support for Segmentation

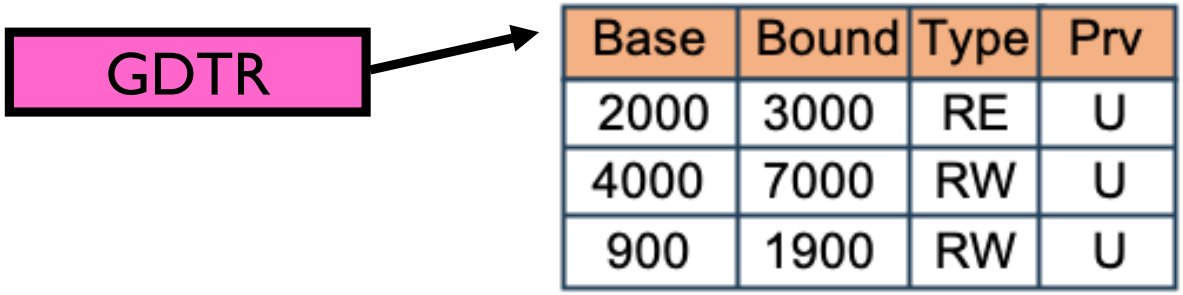
- x86 platform supports six segment registers per CPU
  - **CS (Code Segment)**
  - **SS (Stack Segment)**
  - **DS (Data Segment)**
  - ES, FG, and GS
    - General purpose
- Reduced address translation time as single CPU cycle required to access registers
- A program could have thousands of segments, but at any given time it can only use a maximum of 6 segment registers on x86

Segment Registers



- MMU table can now have entries for each segment registers and their corresponding base and bound values
- **Hardware provides special support for "MMU table"**

Global Descriptor Table (GDT)



- Similar to IDT, there is also a GDT stored in memory (DRAM)
- Usually one GDT per processor
- Each CPU has a dedicated register (GDTR) that holds the starting address of this GDT
- Each entries in GDT has a base-bound pair, privilege mode for this physical memory segment (kernel vs user space), memory access type (read/write/executable), etc.
- Still, too many operations for each virtual memory accesses
  1. Load segment selector from the virtual address
  2. Load GDT **offset** by looking into segment register
  3. Load GDTR and add the above **offset** to get index
  4. Load base and bound values stored at that index
- How to directly access segment's base/bound?
  - Add more hardware support!

Note: There is also another table Local Descriptor Table (LDT) in x86, which is per-process unlike the GDT. GDT can store base/bound for the LDTs

Caching of Base/Bound Per Register

Visible Part	Hidden Part	
GDT index	Base Address, Limit, Access Information	CS
		SS
		DS
		ES
		FS
		GS

- Each segment register has a visible and a hidden part
  - Visible part is programable and it is an index into the GDT for that particular segment register
  - The hidden part is non-programable and it stores the base, bound (a.k.a limit) and access information for each segment register (cache)
    - It is cached automatically the first time a GDT index is accessed
    - No need to access GDT until a context switch is done (Why?)
    - No extra CPU cycles wasted for accessing GDT in VA to PA translation

A System Using Segmentation

```
void scheduler() {
    while(true) {
        lock(process_table);
        foreach(Process p: scheduling_algorithm(process_table)) {
            if(p->state != READY) {
                continue;
            }
            p->state = RUNNING;
            unlock(process_table);
            swtch_segment_registers(scheduler_process, p);
            swtch(scheduler_process, p);
            // p is done for now..
            lock(process_table);
        }
        unlock(process_table);
    }
}
```

- Index into the GDT for all the segment registers are reloaded during context switch
- It also automatically updates the values stored inside the hidden part of each segment register (see previous slide)

Points to Ponder

- Can user process update GDTR to point to a new GDT?
  - No, it can only be done at the boot time by the OS running in privileged mode
- As GDT is stored in memory, can some rouge process can easily change the values stored inside the GDT?
  - No, GDT is stored in physical memory location that is accessible only to the OS
  - Updates to GDT entries are only carried out by OS in privileged mode (e.g., allocating process address space during exec)
- As GDT is stored in memory, can some rouge process access memory segments related to the OS code?
  - No, recall that each entry inside the GDT also has a privilege information stored
  - OS specific memory is stored in the kernel space whereas user application specific memory is stored in user space
- Can a user process update its segment register?
  - Yes, there are only 6 segment registers but a process could have thousands of segments, and it might want to switch to a different segment for which it will update its segment register (except CS, why?)

Segmentation in Linux

- Linux uses segmentation in a very limited form on modern x86 64-bit processors
  - Simplifies the memory management
  - Portability across non-x86 architectures
- Each segment register has exact same base and bound (effectively disabling segmentation)
  - Base is 0 and bound is 2^{32} - 1
  - Privilege is set as per kernel or user accessible segment
- How is then VA to PA mapping carried out?
  - Paging!

Segmentation Summary

- Advantages
  - Limited hardware support (GDTR, 6 segments, etc.)
  - Reduces internal fragmentation (exact amount of memory allocated)
- Disadvantages
  - Difficult to increase the memory size once allocated if there is no contiguous chunk of unused physical memory of the new size
  - Favors external fragmentation due to allocation of variable sized memory chunk for each segments
  - Swapping out to disk has huge overheads, as an entire segment's memory would be swapped out if it is unused

Reference

- Chapter 16 in OSTEP book (“Segmentation”)

Next Lecture

- Introduction to paging
  - If you miss this lecture then you would have hard time understanding Lectures 18-20