PROJECT 3:
PER-CPU VARIABLES

Minimum Requirements: 3 public tests

You ONLY need to modify ~13 files for this project, don’t be scared.

TEST DISTRIBUTION

Public tests – 3 tests | 32 points

Release tests – 0 tests | 0 points

Secret tests – 4 tests | 20 points

PER-CPU VARIABLES

- Data that is local to a processor can be useful.
- In this project, two variables are needed to be saved for each processor (CPU).
  - The current thread.
  - The index of the current CPU.

HOW TO GET PER-CPU VARIABLES

- Before (expensive!):
    - Disable preemption → Visit a shared region of memory in the APIC region → Re-enable preemption.
- After this project (more efficient):
    - Each processor will have its own memory area and can only be accessed by the processor alone. No need to disable interrupts.

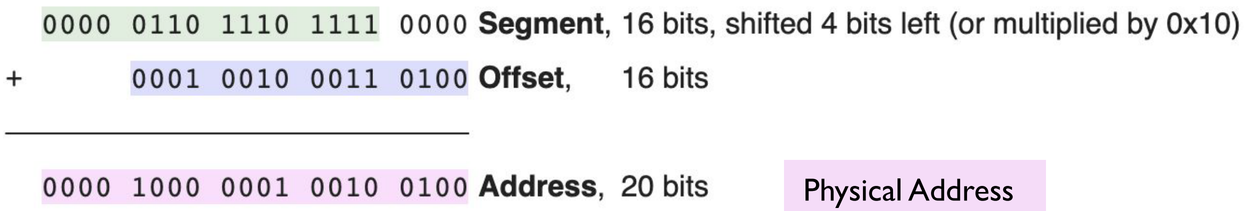
SEGMENTATION

- A memory segment specifies a region of memory and the "privilege level" that is required to access that memory.
- Each user program has its own memory segments - one for code, one for data, one for its stack, plus a couple extra for various purposes.
- If the operating system sets up the segments properly, a program will be limited to accessing only its own memory.
- Memory segments allow programs to use relative memory references. All memory references are interpreted by the processor to be relative to the base of the current memory segment.

X86 REAL MODE & PROTECTED MODE

- Real mode:
  - GeekOS enters this mode upon power up
  - Address translation in real mode:

Segment Register

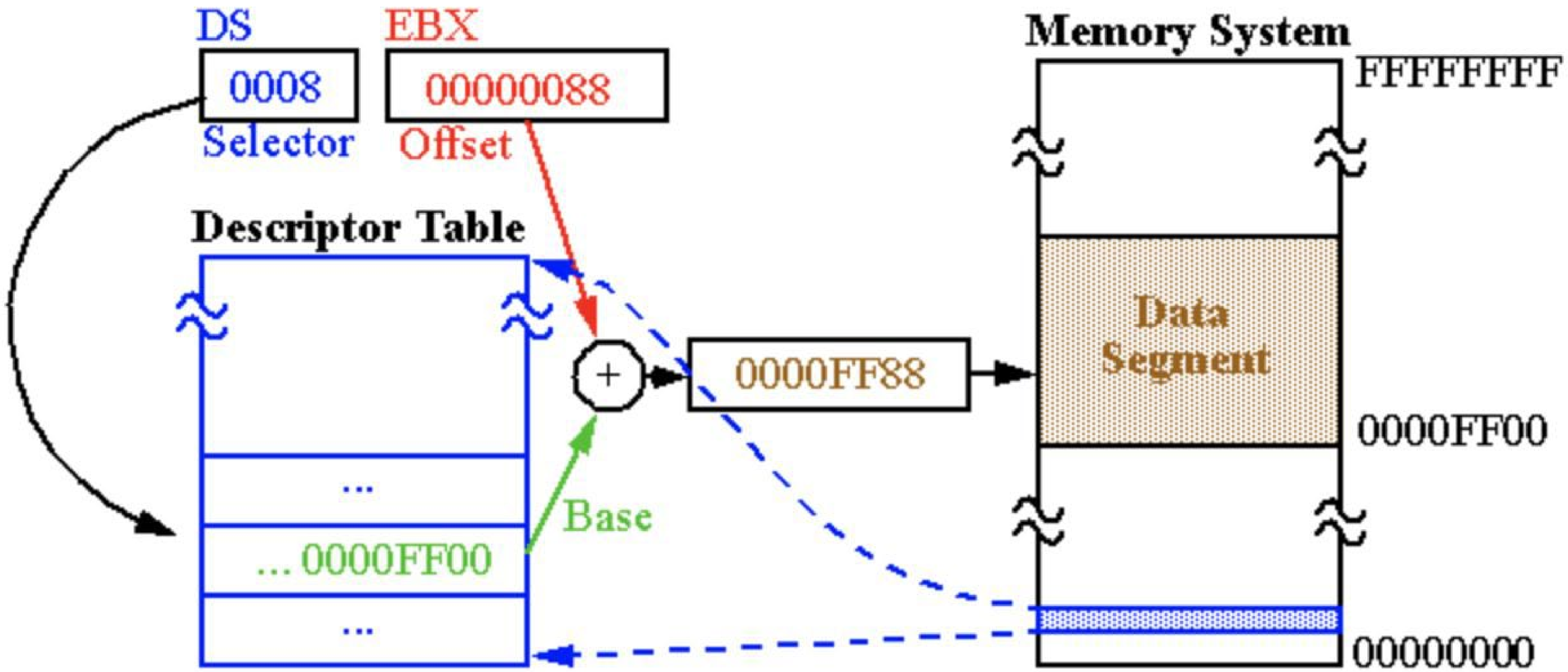


```
  0000 0110 1110 1111  0000 Segment, 16 bits, shifted 4 bits left (or multiplied by 0x10)
+      0001 0010 0011 0100 Offset,   16 bits
  ________________________
  0000 1000 0001 0010 0100 Address,  20 bits
```

Physical Address

- Protected mode
  - Enters protected mode setup.asm. GeekOS runs in this mode.
  - In protected mode, the processor is a 32-bit machine with many more features
  - Address translation: A segment register saves a 16-bit segment selector, which select a 64-bit segment descriptor from a segment descriptor table. The segment descriptor and the 32-bit offset together define a memory address.

Protected Mode Memory Addressing



A hardware memory management unit (MMU) is responsible for translating the segment and offset into a physical address

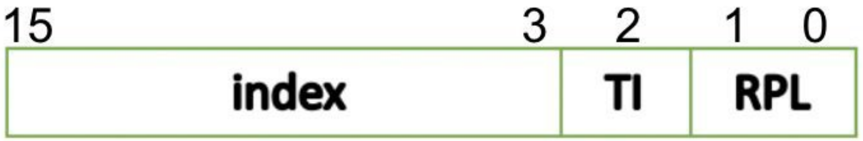
SEGMENT REGISTERS

CS	Code Segment
DS	Data Segment
SS	Stack Segment
ES	Extra Segment
FS	General purpose segments
GS	

- The purpose of segment registers is to hold segment selectors.
- We use GS for this project.

SEGMENT SELECTORS

Format of a segment selector:



15 … 3	2	1 … 0
index	TI	RPL

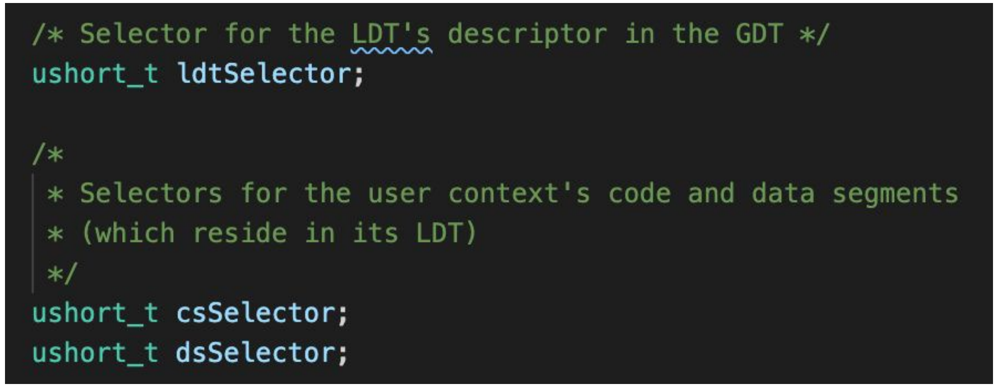
Index: index in GDT or LDT

TI: Table indicator (GDT/LDT)

RPL: Requested Privilege level (protection level of segment)

A segment selector uses its index as the entry index of the segment descriptor in a GDT/LDT.

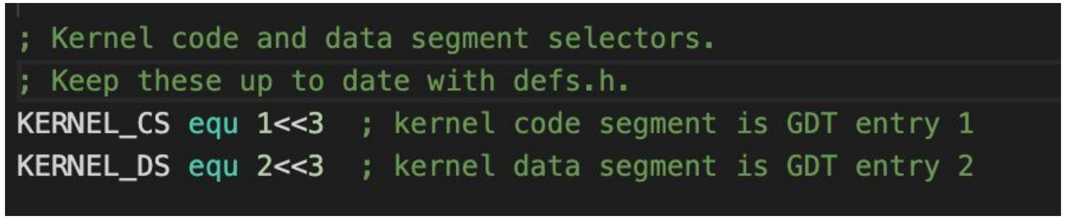
User's segment selectors (user.h)



```
/* Selector for the LDT's descriptor in the GDT */
ushort_t ldtSelector;

/*
 * Selectors for the user context's code and data segments
 * (which reside in its LDT)
 */
ushort_t csSelector;
ushort_t dsSelector;
```

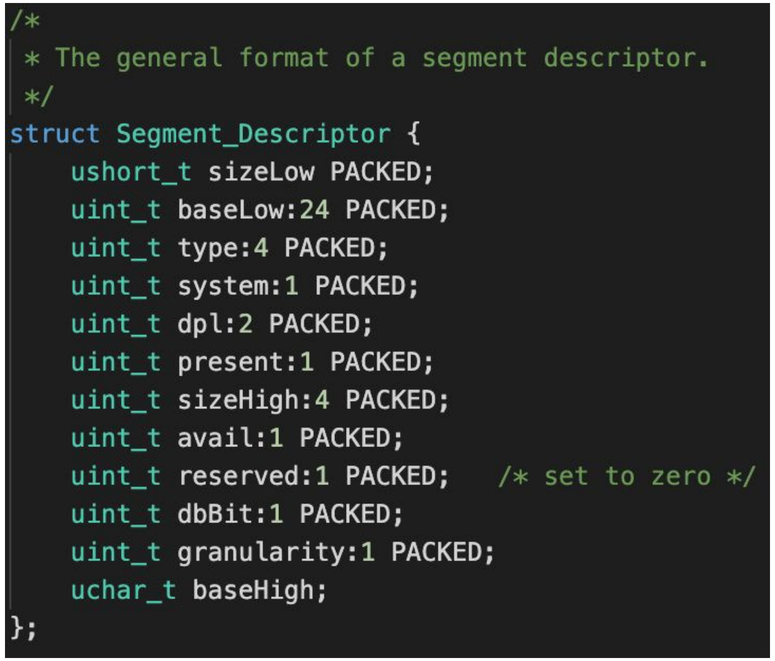
Kernel's segment selectors (defs.asm)



```
; Kernel code and data segment selectors.
; Keep these up to date with defs.h.
KERNEL_CS equ 1<<3  ; kernel code segment is GDT entry 1
KERNEL_DS equ 2<<3  ; kernel data segment is GDT entry 2
```

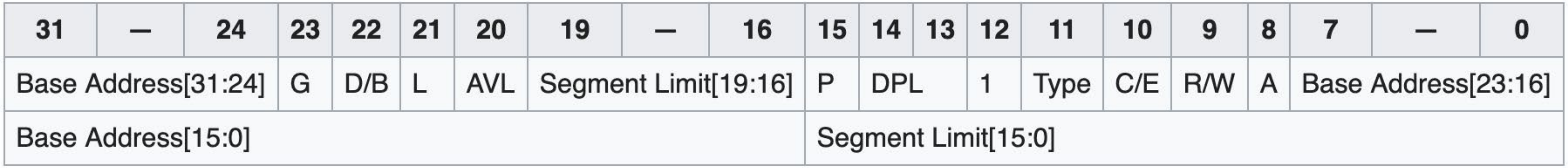
SEGMENT DESCRIPTORS

- Fully describe characteristics of memory segments.
- Include the base address, the limit (size of the memory segment), the privilege level and several other bits.
- Segment descriptors are stored in segment descriptor tables.



```
/*
 * The general format of a segment descriptor.
 */
struct Segment_Descriptor {
    ushort_t sizeLow PACKED;
    uint_t baseLow:24 PACKED;
    uint_t type:4 PACKED;
    uint_t system:1 PACKED;
    uint_t dpl:2 PACKED;
    uint_t present:1 PACKED;
    uint_t sizeHigh:4 PACKED;
    uint_t avail:1 PACKED;
    uint_t reserved:1 PACKED;   /* set to zero */
    uint_t dbBit:1 PACKED;
    uint_t granularity:1 PACKED;
    uchar_t baseHigh;
};
```

The x86 and x86-64 segment descriptor structure:



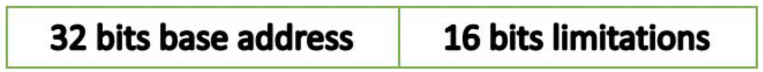
31	—	24	23	22	21	20	19	—	16	15	14	13	12	11	10	9	8	7	—	0
Base Address[31:24]			G	D/B	L	AVL	Segment Limit[19:16]			P	DPL		1	Type	C/E	R/W	A	Base Address[23:16]		
Base Address[15:0]										Segment Limit[15:0]										

DESCRIPTOR TABLE

- Two types of descriptor tables:
  - Global descriptor table (GDT)
    - Contains information for all of the processes and kernel.
    - There will be a GDT per processor.
    - Each user process has an entry in the GDT that refers to its LDT (in userContext)
  - Local descriptor table (LDT)
    - Stores the segment descriptors for each user process.
    - There is only one LDT per process.
- But how does the process find the GDT or its LDT?
  - By using the GDTR and LDTR registers.

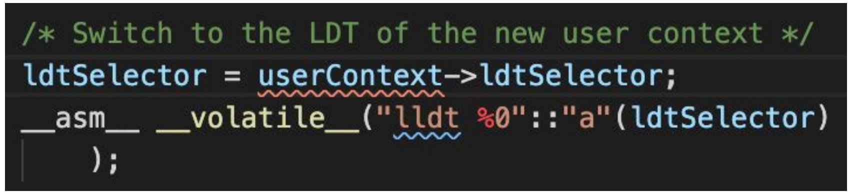
GDTR & LDTR REGISTERS

- GDTR stores the addresses of the GDT
- LDTR stores the selector that selects the LDT descriptor of the current process in GDT
- The memory segments for a process are activated by loading:
  - The LDT segment selector into the LDTR
  - The segment selectors into the various segment registers (CS, DS, GS, etc).

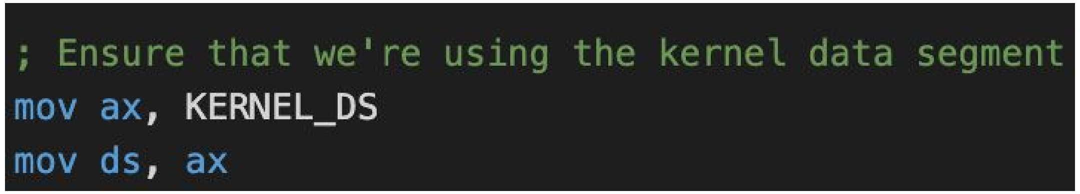


32 bits base address	16 bits limitations

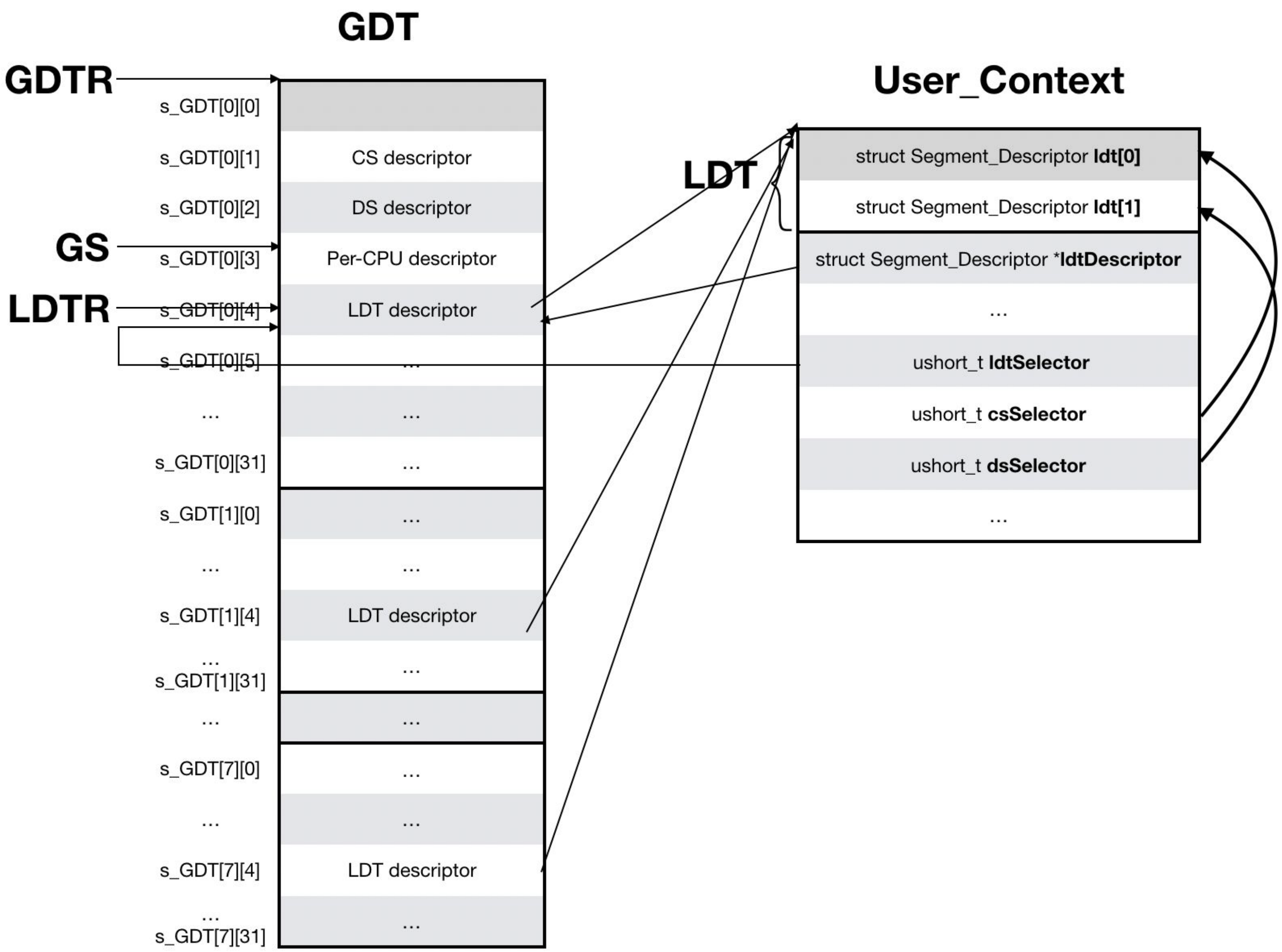
GDTR

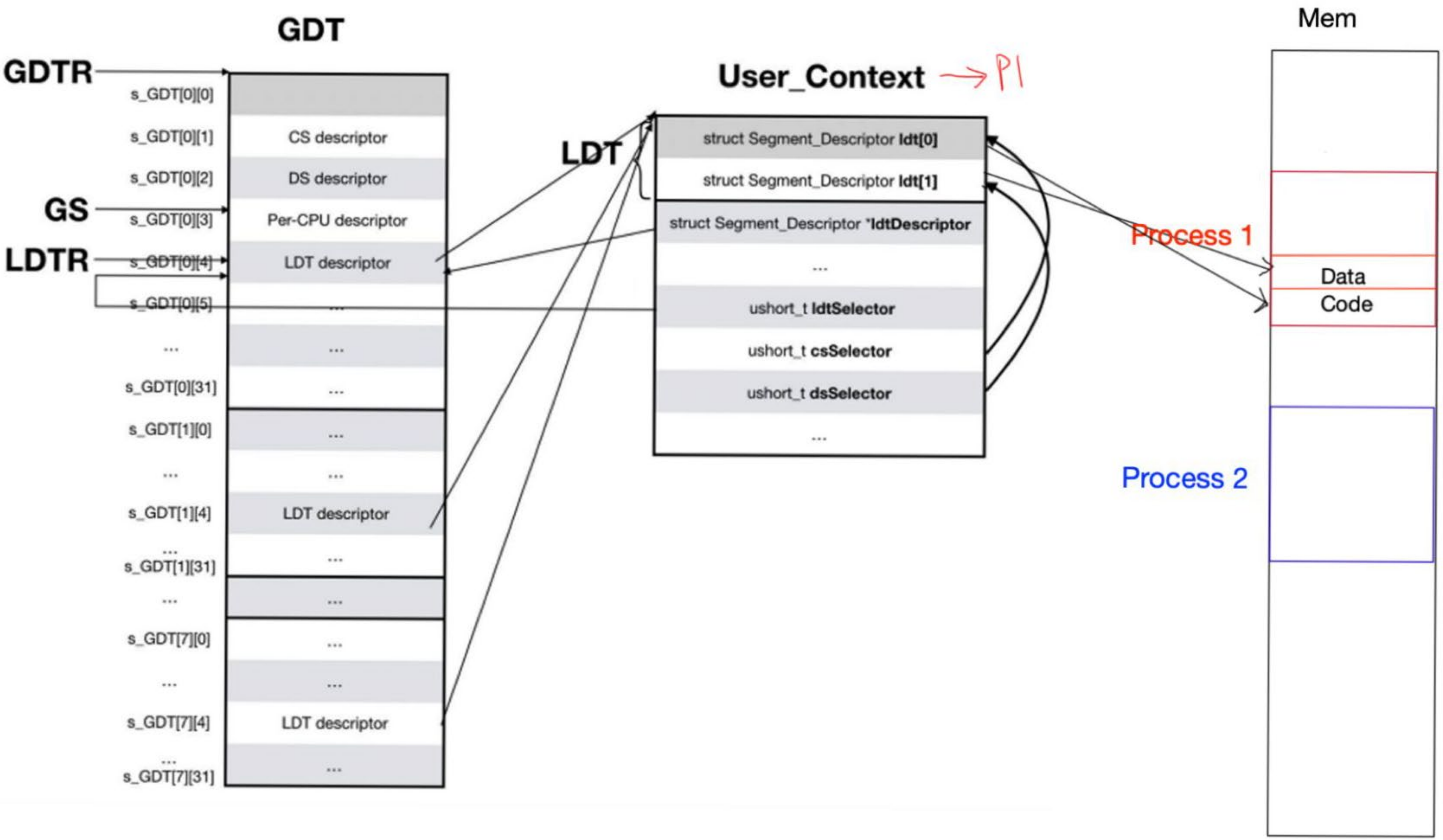


```
/* Switch to the LDT of the new user context */
ldtSelector = userContext->ldtSelector;
__asm__ __volatile__("lldt %0"::"a"(ldtSelector)
    );
```



```
; Ensure that we're using the kernel data segment
mov ax, KERNEL_DS
mov ds, ax
```


GDT
GDTR
GS
LDTR
s_GDT[0][0]
s_GDT[0][1]
CS descriptor
s_GDT[0][2]
DS descriptor
s_GDT[0][3]
Per-CPU descriptor
s_GDT[0][4]
LDT descriptor
s_GDT[0][5]
…
…
…
s_GDT[0][31]
…
s_GDT[1][0]
…
…
…
s_GDT[1][4]
LDT descriptor
…
s_GDT[1][31]
…
…
…
s_GDT[7][0]
…
…
…
s_GDT[7][4]
LDT descriptor
…
s_GDT[7][31]
…
User_Context
LDT
struct Segment_Descriptor ldt[0]
struct Segment_Descriptor ldt[1]
struct Segment_Descriptor *ldtDescriptor
…
ushort_t ldtSelector
ushort_t csSelector
ushort_t dsSelector
…



GDT

GDTR

GS

LDTR

s_GDT[0][0]

s_GDT[0][1] CS descriptor

s_GDT[0][2] DS descriptor

s_GDT[0][3] Per-CPU descriptor

s_GDT[0][4] LDT descriptor

s_GDT[0][5] ...

... ...

s_GDT[0][31] ...

s_GDT[1][0] ...

... ...

s_GDT[1][4] LDT descriptor

... s_GDT[1][31] ...

... ...

s_GDT[7][0] ...

... ...

s_GDT[7][4] LDT descriptor

... s_GDT[7][31] ...

LDT

User_Context → P1

struct Segment_Descriptor **ldt[0]**

struct Segment_Descriptor **ldt[1]**

struct Segment_Descriptor ***ldtDescriptor**

...

ushort_t **ldtSelector**

ushort_t **csSelector**

ushort_t **dsSelector**

...

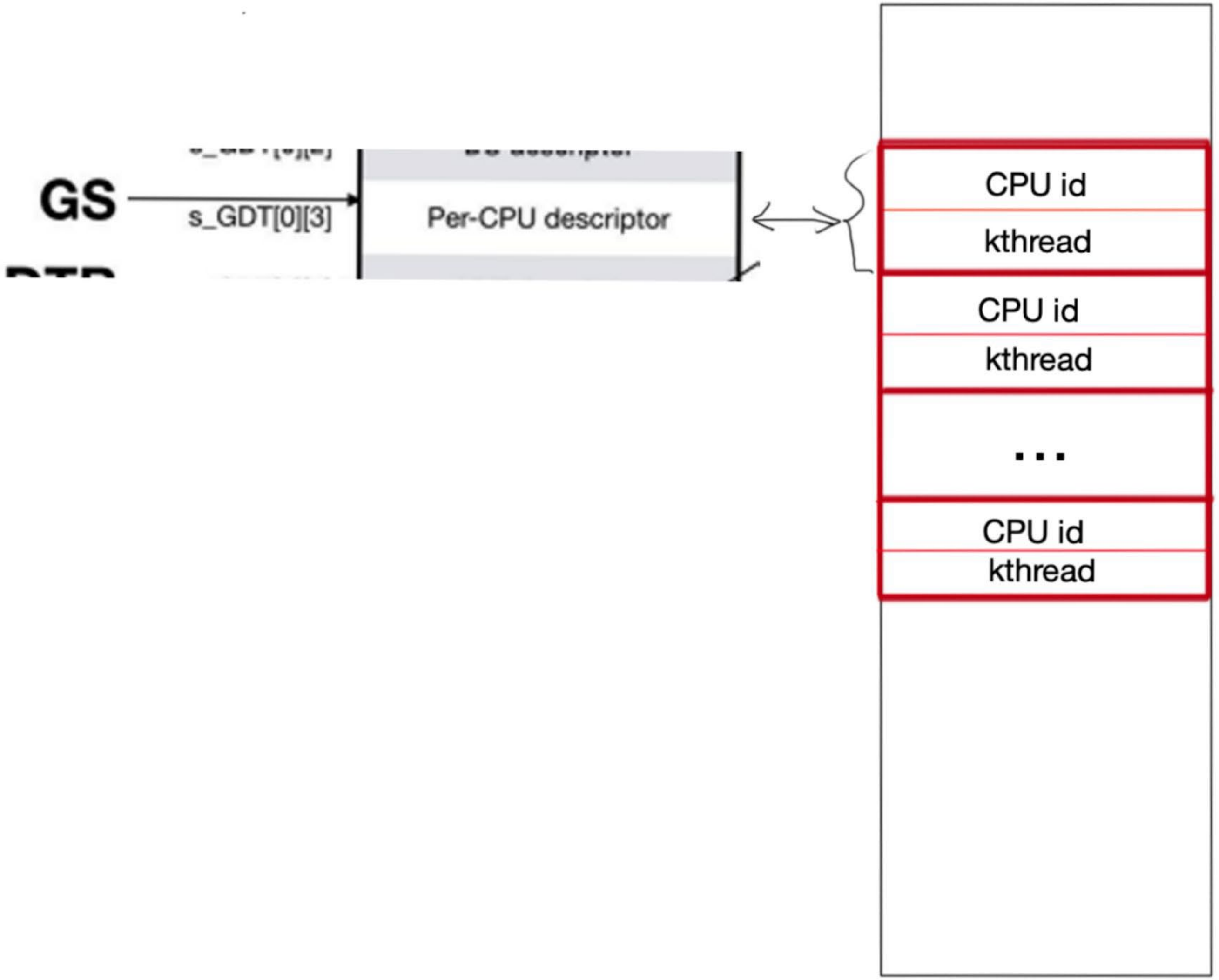
Mem

Process 1

Data

Code

Process 2



GS

s_GDT[0][3]

Per-CPU descriptor

CPU id

kthread

CPU id

kthread

...

CPU id

kthread

Per-cpu variables for cpu0

Per-cpu variables for cpu1

DATA STRUCTURE DEF: PERCPU.C AND PERCPU.H

- percpu.h
    - Define a struct to hold percpu data (e.g. cpu index and a pointer to current thread). Make an array to hold the percpu data struct instances for all CPUs.
- Percpu.c
    - Void Init_PerCPU(int cpu):
        - Init the related percpu data structure instance. The field related to current_thread can be initialized to null.
    - Int PerCPU_Get_CPU(void)
        - Get the CPU ID using inline assembly.
    - Struct Kernel_Thread *PerCPU_Get_Current(void)
        - Get the current thread using inline assembly.
- Use inline assembly: move the data stored at gs_segment + some_offset to the local var and return. (Later we will define the offset of gs segments as the addresses of the elements in the per-cpu data array.)

INLINE ASSEMBLY EXAMPLE:

AT&T syntax: mov src, dst

```
asm("movl %%gs:0, %0"
: "=r" (kthread) //output, %0
:
);
```

Description: Copy the value from gs segment offset by 0 into variable kthread, use any register.

ASSEMBLY GETTER SETTER: PERCPU.ASM

- Currently empty, but you need to redefine current_thread macros to use the per-cpu segment.
- Overwrite the macros defined in lowlevel.asm, you might want to take a look at their original definition
    - Get_Current_Thread_To_EAX
    - Set_Current_Thread_From_EBX
    - Push_Current_Thread_PTR
- Each of them should only be one line assembly within the macro in x86 assembly syntax.

INITIALIZATION: MAIN.C, SMP.C AND KTHREAD.C

- Call Init_PerCPU(). (Main.c)
    - Init the first cpu data (cpu 0)
- In Secondary_Start (Smp.c)
    - Call Init_PerCPU()
    - You know which cpu you are supposed to init since the CPU id is given to you.
- In Init_Scheduler (Kthread.c)
    - Put mainThread into the percpu data struct instance of the current CPU.

GETTER INVOCATION: SMP.C

- Modify get_current_thread(Smp.c)
  - Use the function you defined in percpu.c to get the current thread.

SET UP GDT: GDT.C

- s_GDT: kernel's global descriptor table (max number of CPUs allowed here is 8, but the actual number used is CPU_Count in smp.c)
- Allocate a segment descriptor for the cpu of index (int)cpu rather than just for the first cpu.

SET UP GDT: GDT.C

- In Init_GDT function
  - Go through this function and the functions used in it to see how the GDT entries (descriptors) for cs and ds are allocated and initialized.
  - It originally only initializes the GDT for the first CPU (only when if(cpuid == 0)), since in this project we are using multiple CPUs, the way of assigning the descriptors should be changed accordingly.
  - Allocate a descriptor for gs (percpu data segment), and initialize that descriptor. For the initialization part, you might want to define another function in segment.c to do that.

SET UP GDT: SEGMENT.C

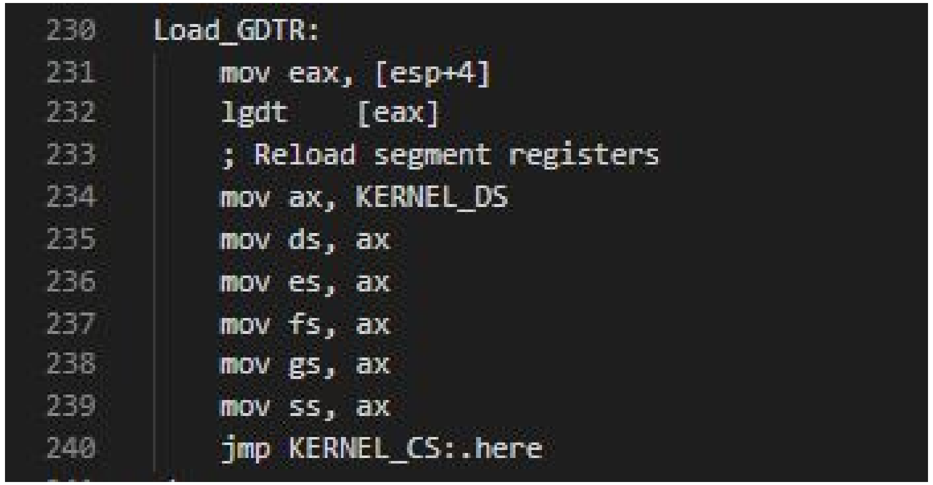
- Define a function (which you might use in gdt.c) to initialize the descriptor for the percpu data segment.
  - Refer to Init_Data_Segment_Descriptor() on how everything is initialized, but you may want to use Set_Size_And_Base_Bytes().

DEFINE SEGMENT SELECTOR

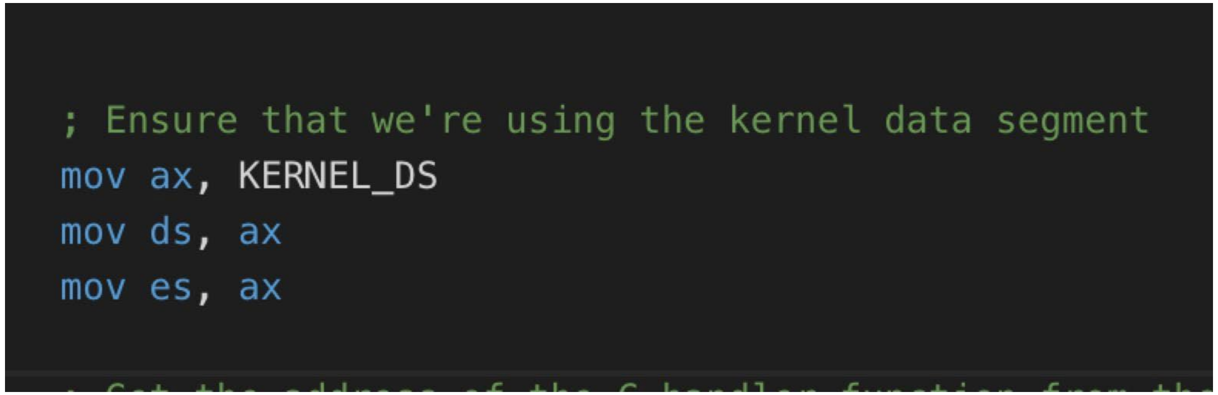
- defs.asm
    - You’ll need to define a GDT selector for the per-cpu data segment. Refer to how KERNEL_CS is defined.
- defs.h
    - You’ll need to define a Per-CPU variables selector. Refer to how KERNEL_CS is defined.

LOWLEVEL.ASM

- Load gs in Handle_Interrupt and Load_GDTR



```
Load_GDTR:
	mov eax, [esp+4]
	lgdt    [eax]
	; Reload segment registers
	mov ax, KERNEL_DS
	mov ds, ax
	mov es, ax
	mov fs, ax
	mov gs, ax
	mov ss, ax
	jmp KERNEL_CS:.here
```



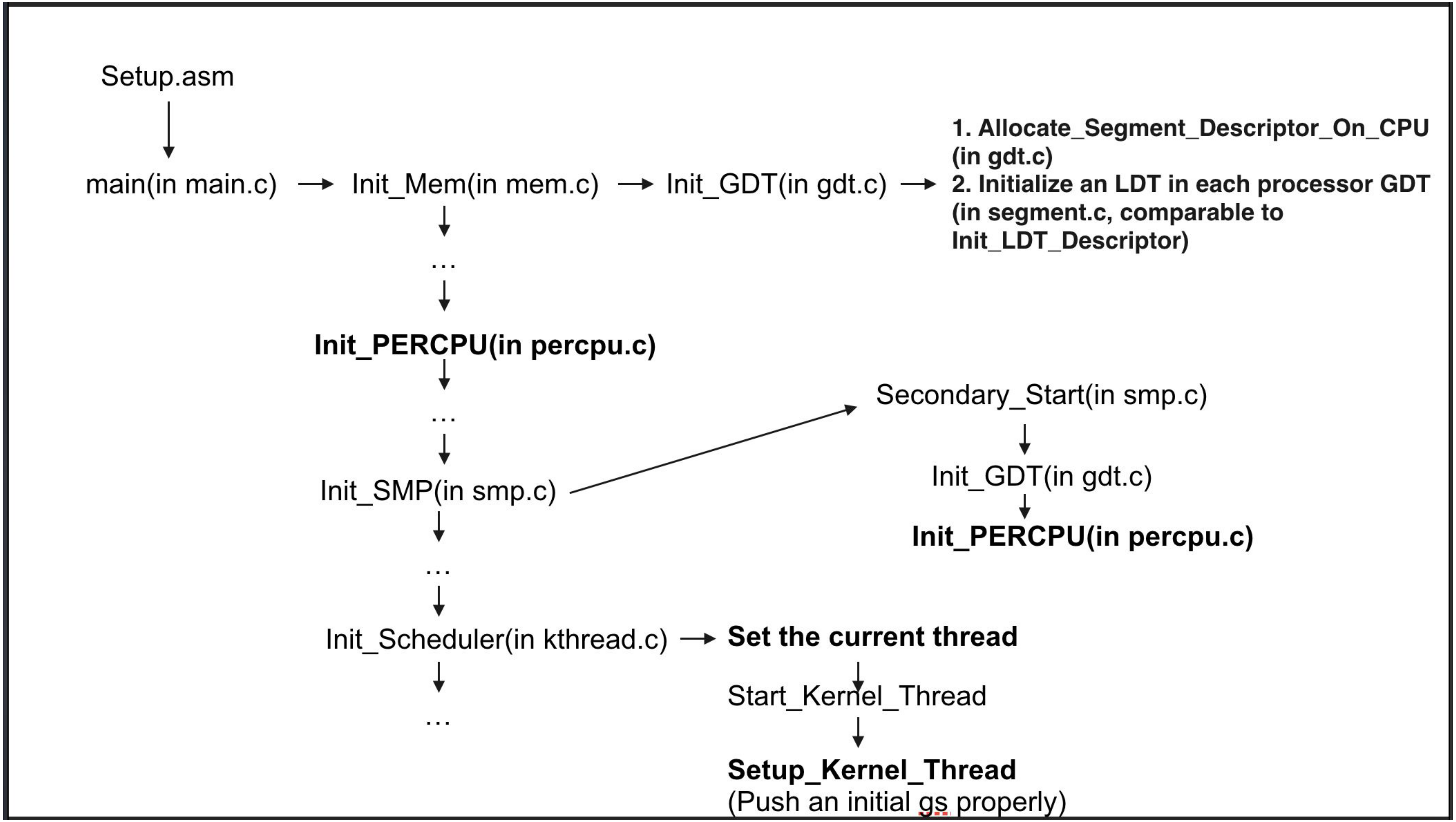
```
; Ensure that we're using the kernel data segment
mov ax, KERNEL_DS
mov ds, ax
mov es, ax
```

SET UP SEGMENT SELECTOR: KTHREAD.C

- In Setup Kernel Thread, push an initial gs appropriately.

SET UP LDT: USER.H AND USERSEG.C

- user.h
  - Make the Segment_Descriptor *ldtDescriptor an array, each element in the array points to the GDT entry that contains a descriptor corresponding to the LDT of the user process.
- Userseg.c
  - You might need to use extern CPU_Count to import the CPU_Count var from smp.c
  - Create_User_Context: originally only puts the ldtDescriptor into the GDT of one cpu (one row in s_GDT), but now you need to allocate a segment descriptor in each cpu's GDT(different rows in s_GDT) and initialize that entry as a descriptor for the LDT of the current process. (You can either use a loop or write a helper function.)
  - Destroy_User_Context: free the segment descriptors for the LDT in all the GDTs.



Setup.asm

main(in main.c) → Init_Mem(in mem.c) → Init_GDT(in gdt.c) →

1. Allocate_Segment_Descriptor_On_CPU (in gdt.c)

2. Initialize an LDT in each processor GDT (in segment.c, comparable to Init_LDT_Descriptor)

…

Init_PERCPU(in percpu.c)

…

Init_SMP(in smp.c)

Secondary_Start(in smp.c)

Init_GDT(in gdt.c)

Init_PERCPU(in percpu.c)

…

Init_Scheduler(in kthread.c) → **Set the current thread**

Start_Kernel_Thread

Setup_Kernel_Thread

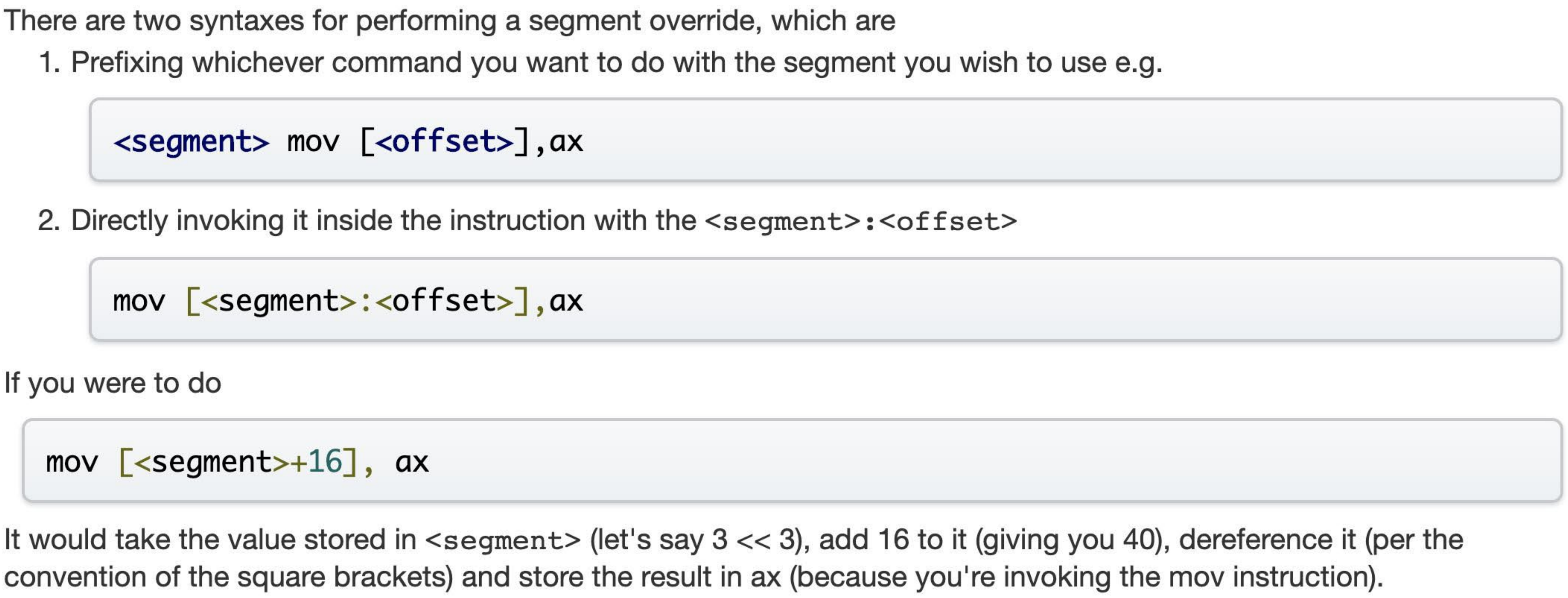
(Push an initial gs properly)

…

HINTS

- Extremely hard to debug, make sure you understand GDT/LDT and the slides before writing the code.
- Look for TODO_P(PROJECT_PERCPU, …) to know where to add code.
- The following files should be modified, check all of them if you run into any issue:
    - percpu.c
    - percpu.h (optional)
    - percpu.asm
    - main.c
    - smp.c
    - kthread.c (2 locations)
    - gdt.c
    - segment.c
    - defs.asm
    - defs.h
    - lowlevel.asm (2 locations)
    - user.h
    - userseg.c

HINTS



There are two syntaxes for performing a segment override, which are

1. Prefixing whichever command you want to do with the segment you wish to use e.g.

```
<segment> mov [<offset>],ax
```

2. Directly invoking it inside the instruction with the `<segment>:<offset>`

```
mov [<segment>:<offset>],ax
```

If you were to do

```
mov [<segment>+16], ax
```

It would take the value stored in `<segment>` (let's say 3 << 3), add 16 to it (giving you 40), dereference it (per the convention of the square brackets) and store the result in ax (because you're invoking the mov instruction).