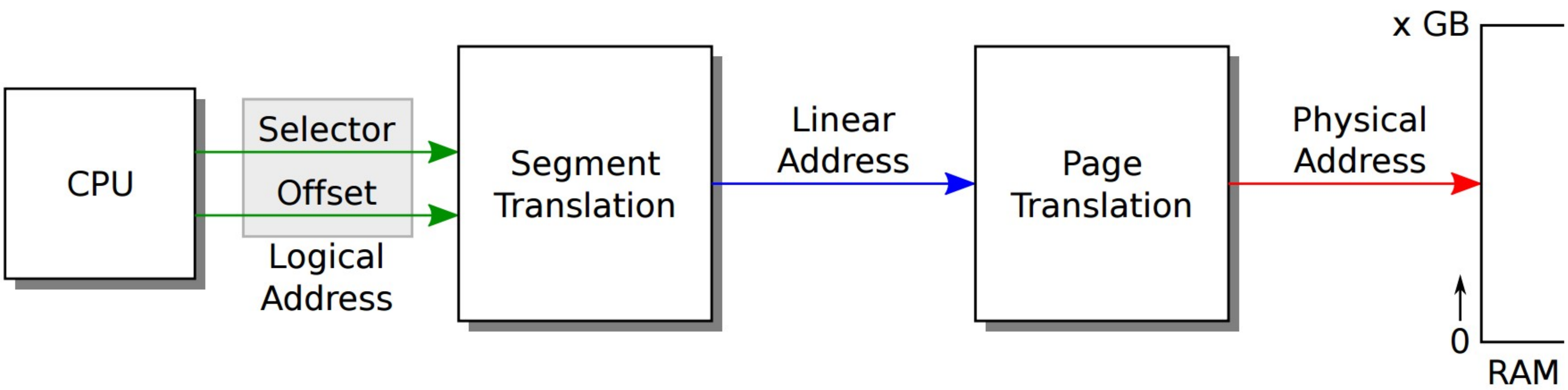
ICS143A: Principles of Operating Systems

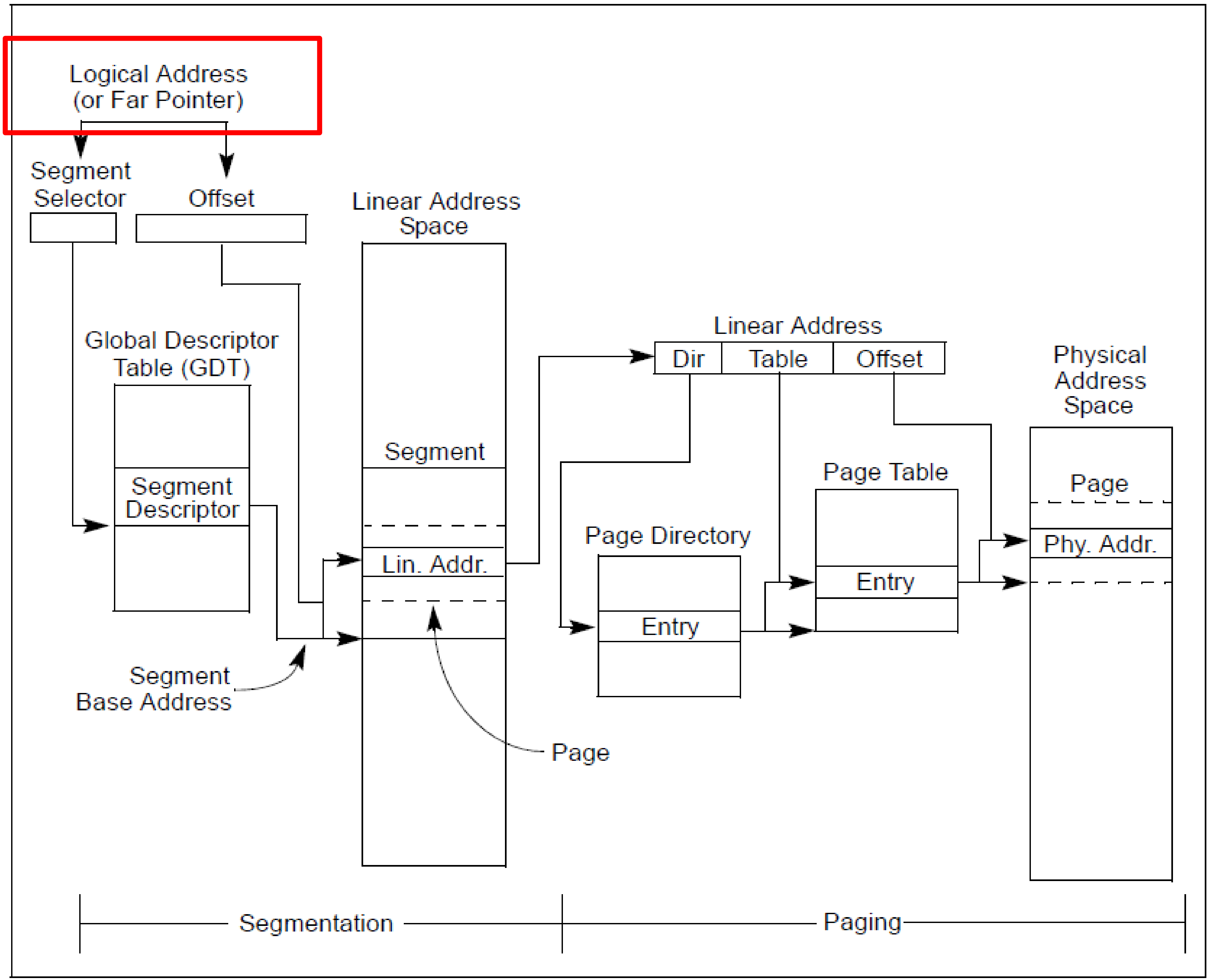
Midterm recap, sample questions

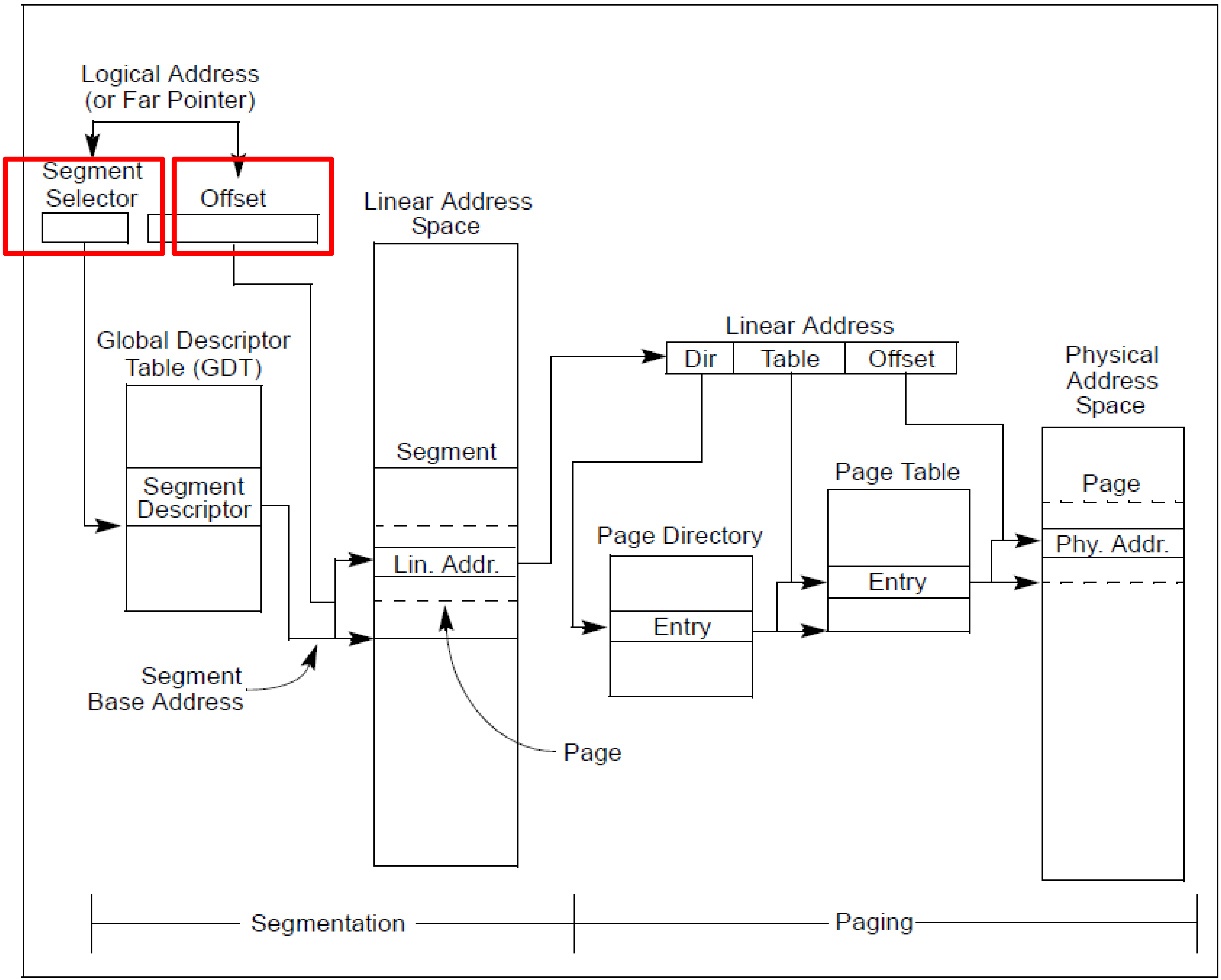
Anton Burtsev
November, 2017

Describe the x86 address translation pipeline (draw figure), explain stages.

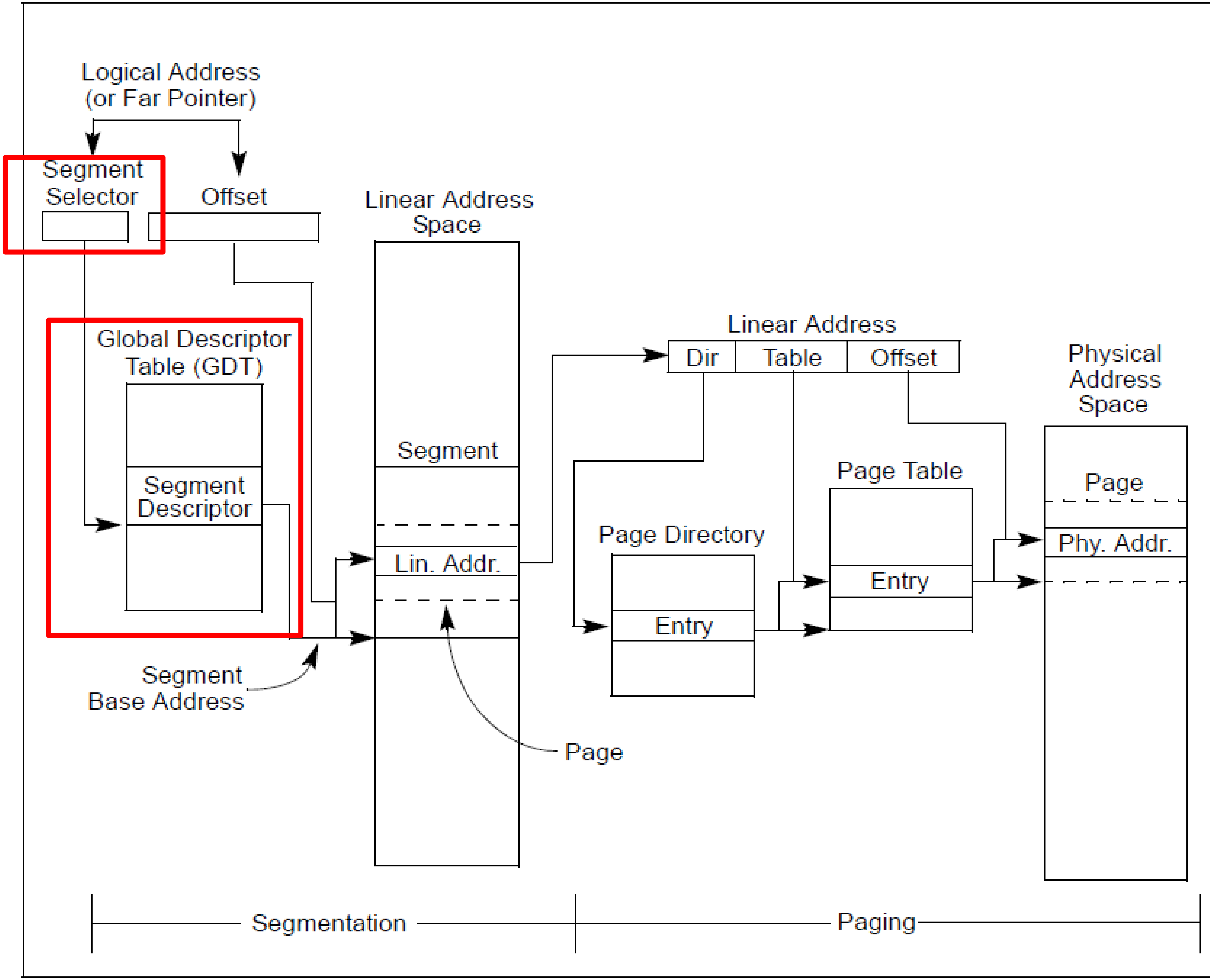
Address translation

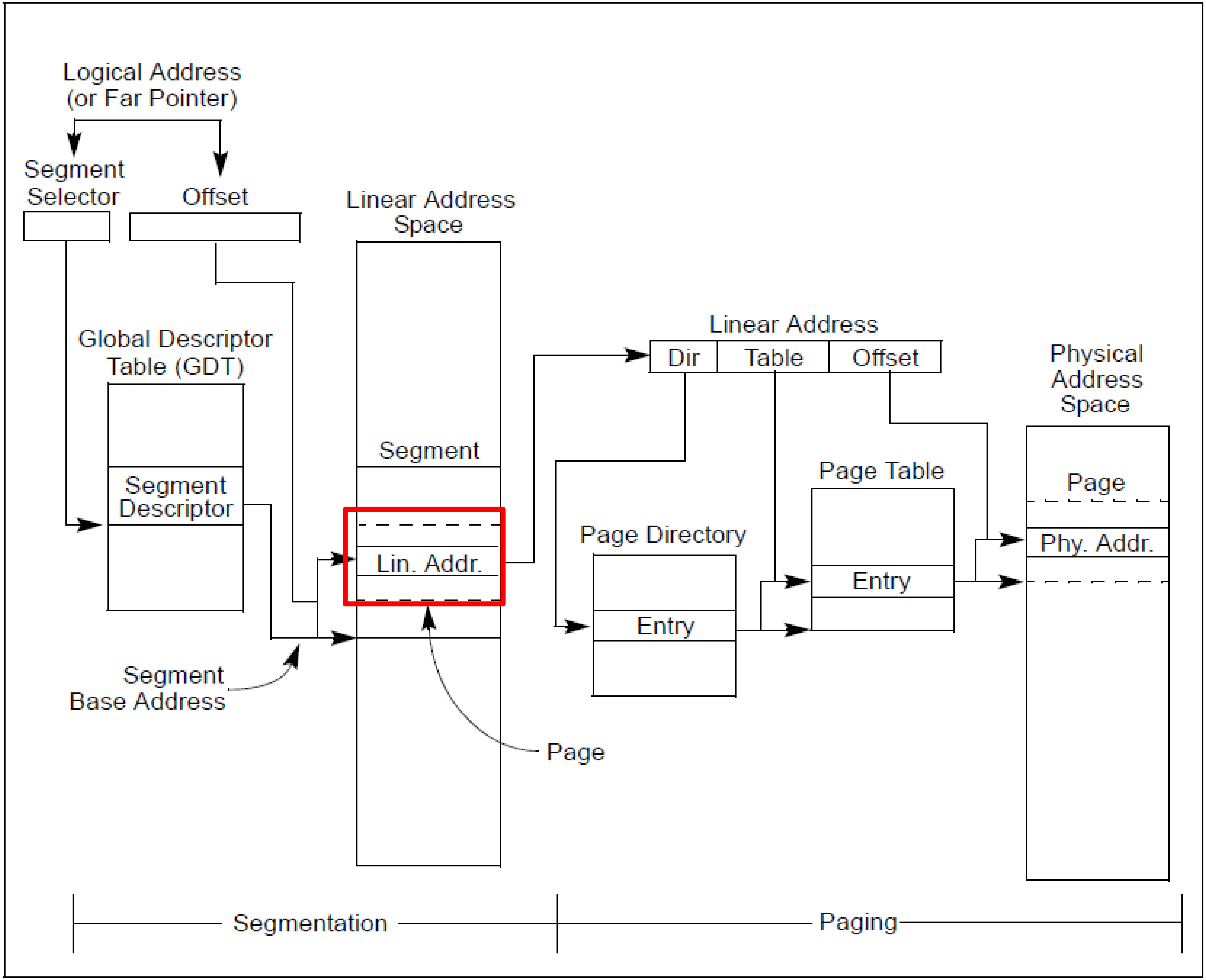


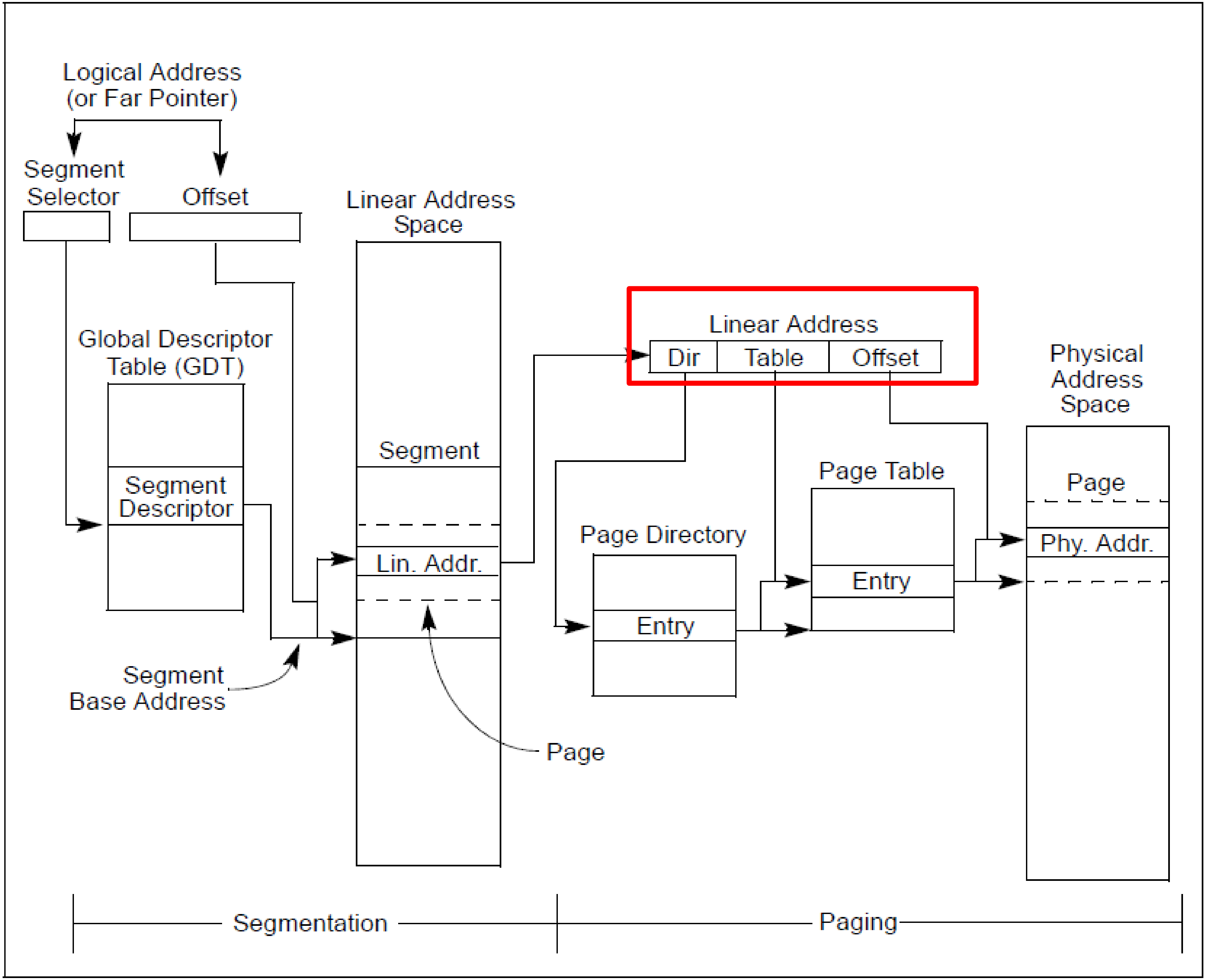

Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Segment
Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical
Address
Space
Page
Phy. Addr.
Segmentation
Paging

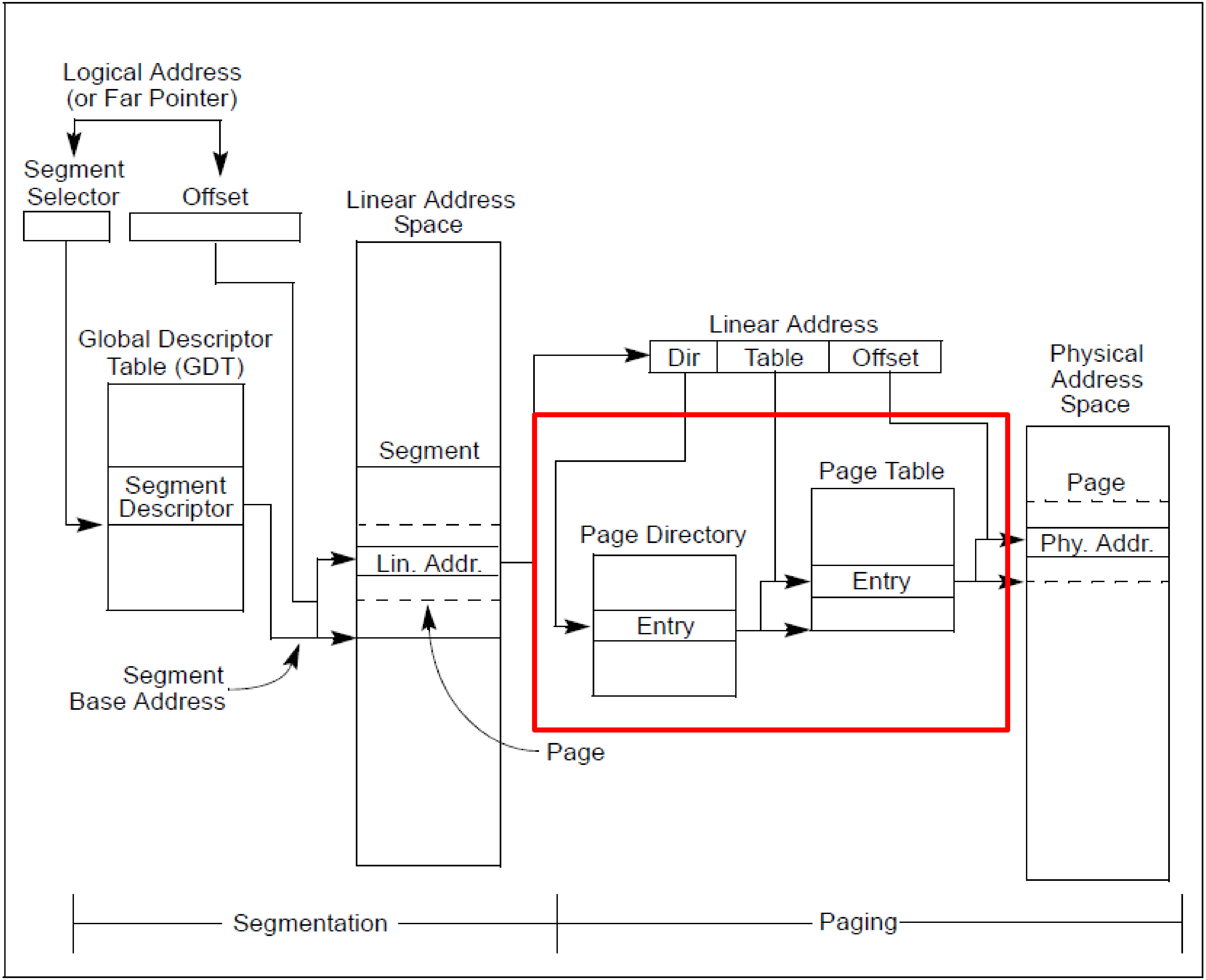


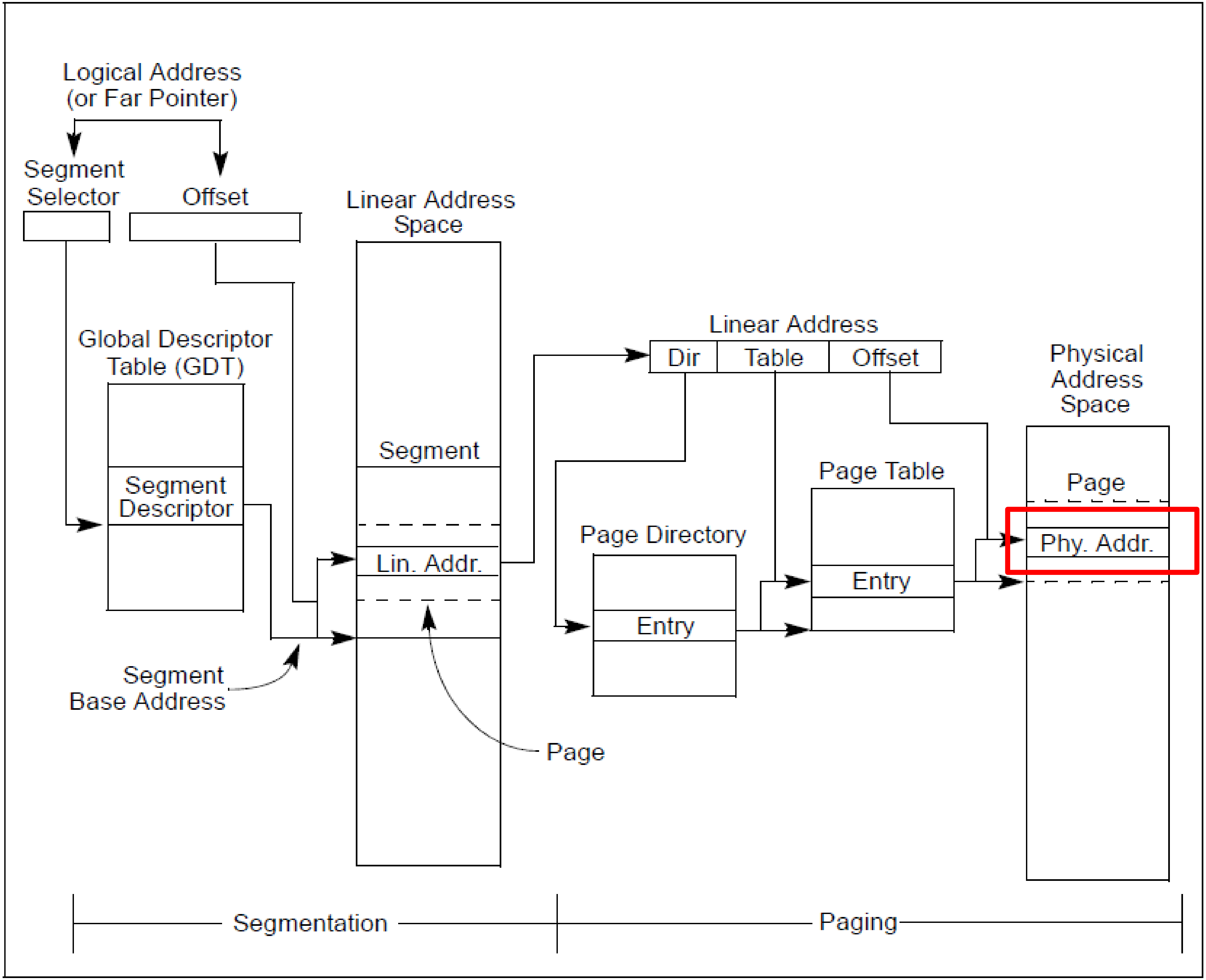
Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Segment
Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical
Address
Space
Page
Phy. Addr.
Segmentation
Paging

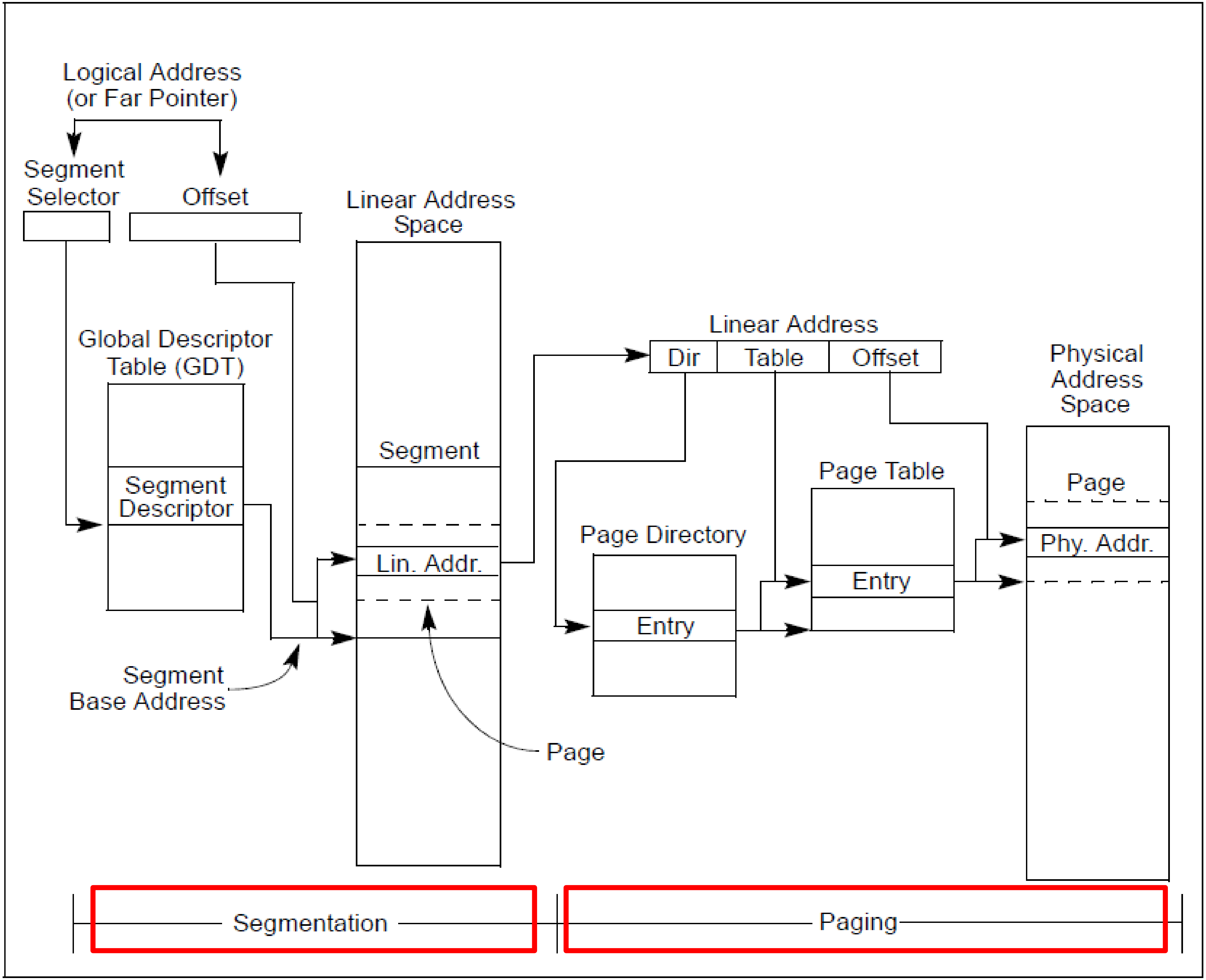

Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Segment
Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical
Address
Space
Page
Phy. Addr.
Segmentation
Paging


Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Segment
Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical
Address
Space
Page
Phy. Addr.
Segmentation
Paging


Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Segment
Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical
Address
Space
Page
Phy. Addr.
Segmentation
Paging


Logical Address
(or Far Pointer)
Segment Selector
Offset
Linear Address Space
Global Descriptor Table (GDT)
Segment Descriptor
Segment
Lin. Addr.
Segment Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical Address Space
Page
Phy. Addr.
Segmentation
Paging

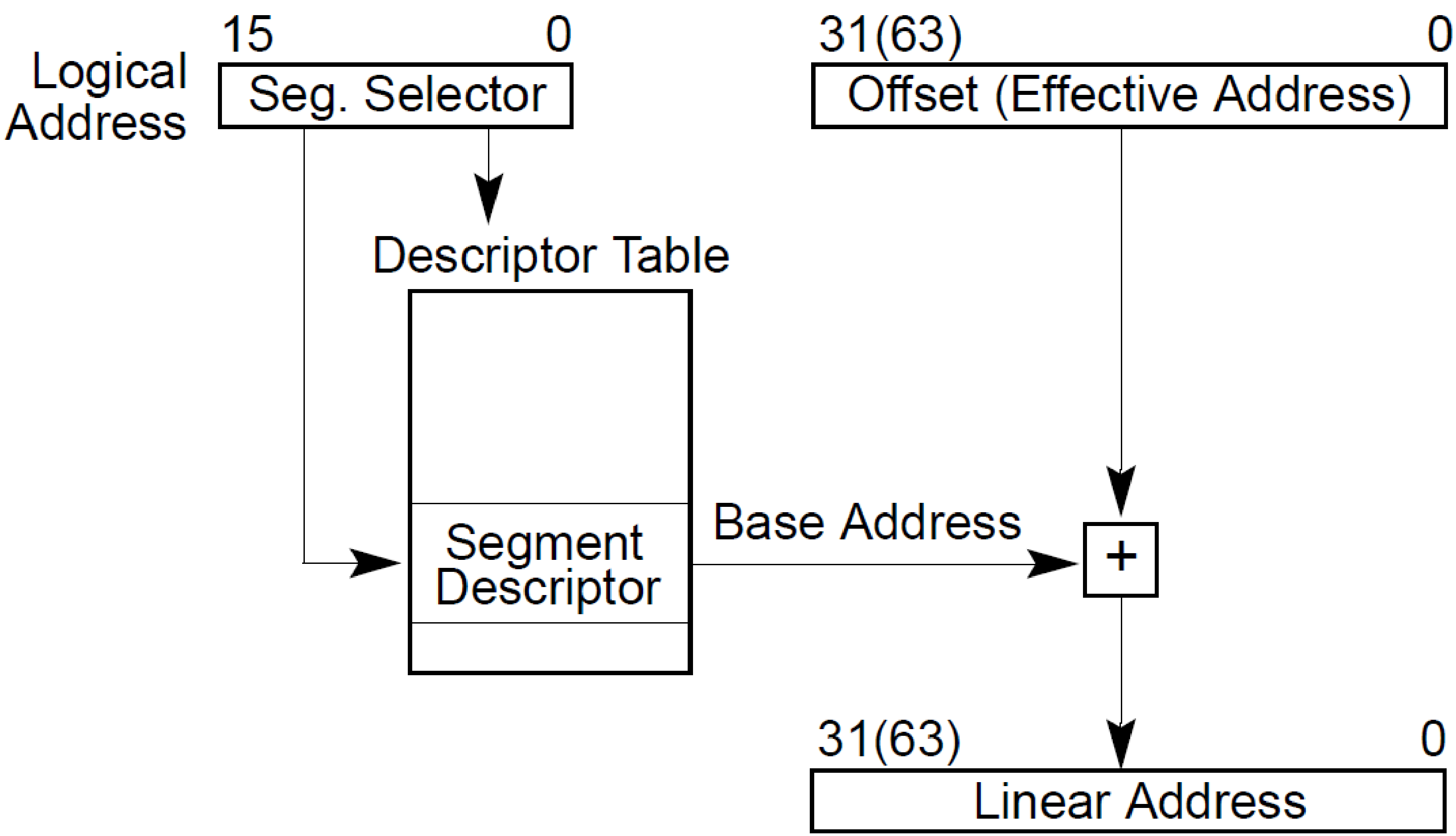

Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Segment
Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical
Address
Space
Page
Phy. Addr.
Segmentation
Paging


Logical Address
(or Far Pointer)
Segment Selector
Offset
Linear Address Space
Global Descriptor Table (GDT)
Segment Descriptor
Segment
Lin. Addr.
Segment Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical Address Space
Page
Phy. Addr.
Segmentation
Paging

What is the linear address? What address is in the registers, e.g., in %eax?

Logical and linear addresses

- Segment selector (16 bit) + offset (32 bit)



What segments do the following instructions use? push, jump, mov

Programming model

- Segments for: code, data, stack, “extra”
  - A program can have up to 6 total segments
  - Segments identified by registers: cs, ds, ss, es, fs, gs

- Prefix all memory accesses with desired segment:
  - `mov eax, ds:0x80` (load offset 0x80 from data into eax)
  - `jmp cs:0xab8` (jump execution to code offset 0xab8)
  - `mov ss:0x40, ecx` (move ecx to stack offset 0x40)

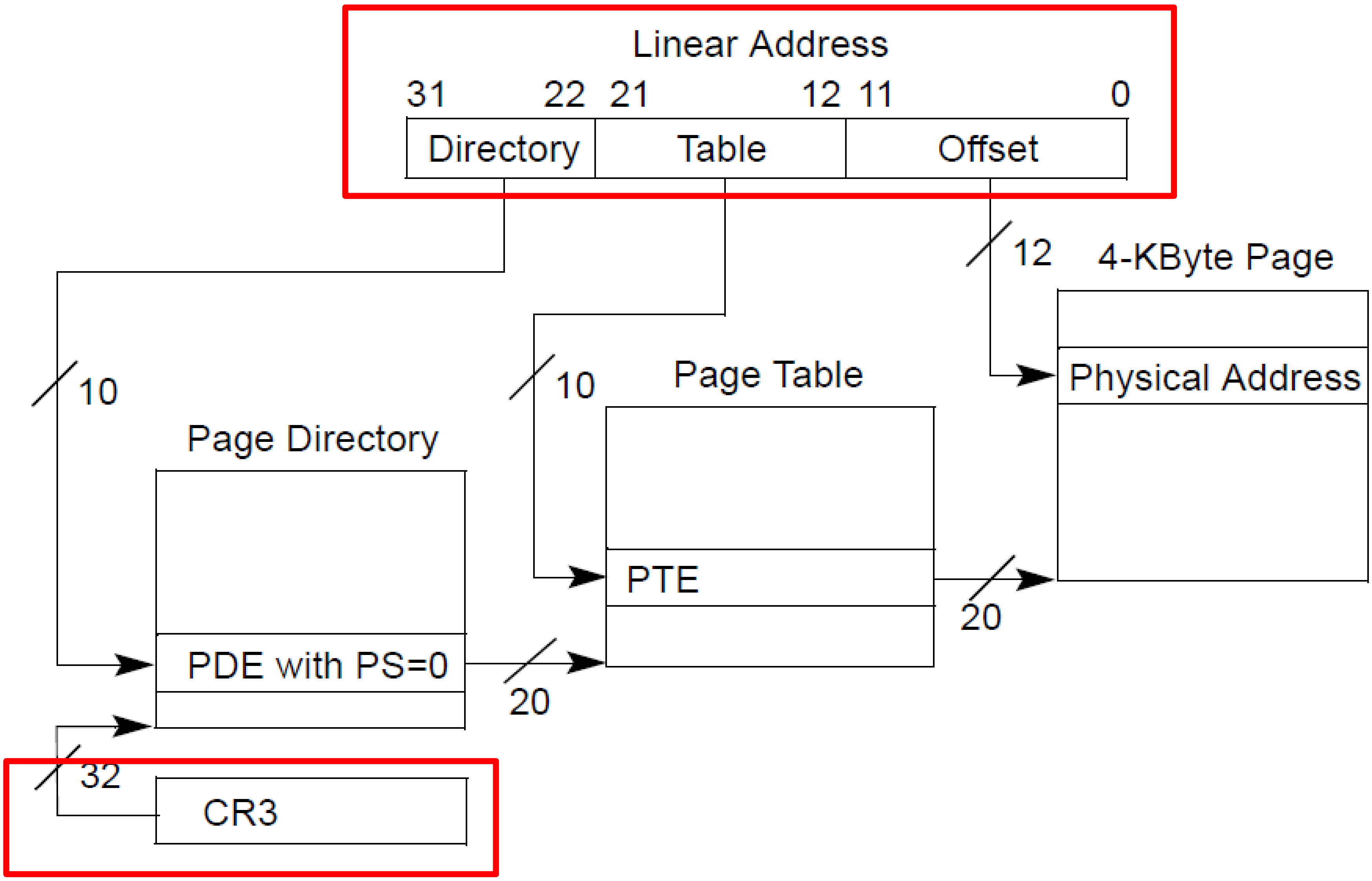
Segmented programming (not real)

```
static int x = 1;
int y; // stack
if (x) {
    y = 1;
    printf ("Boo");
} else
    y = 0;
```

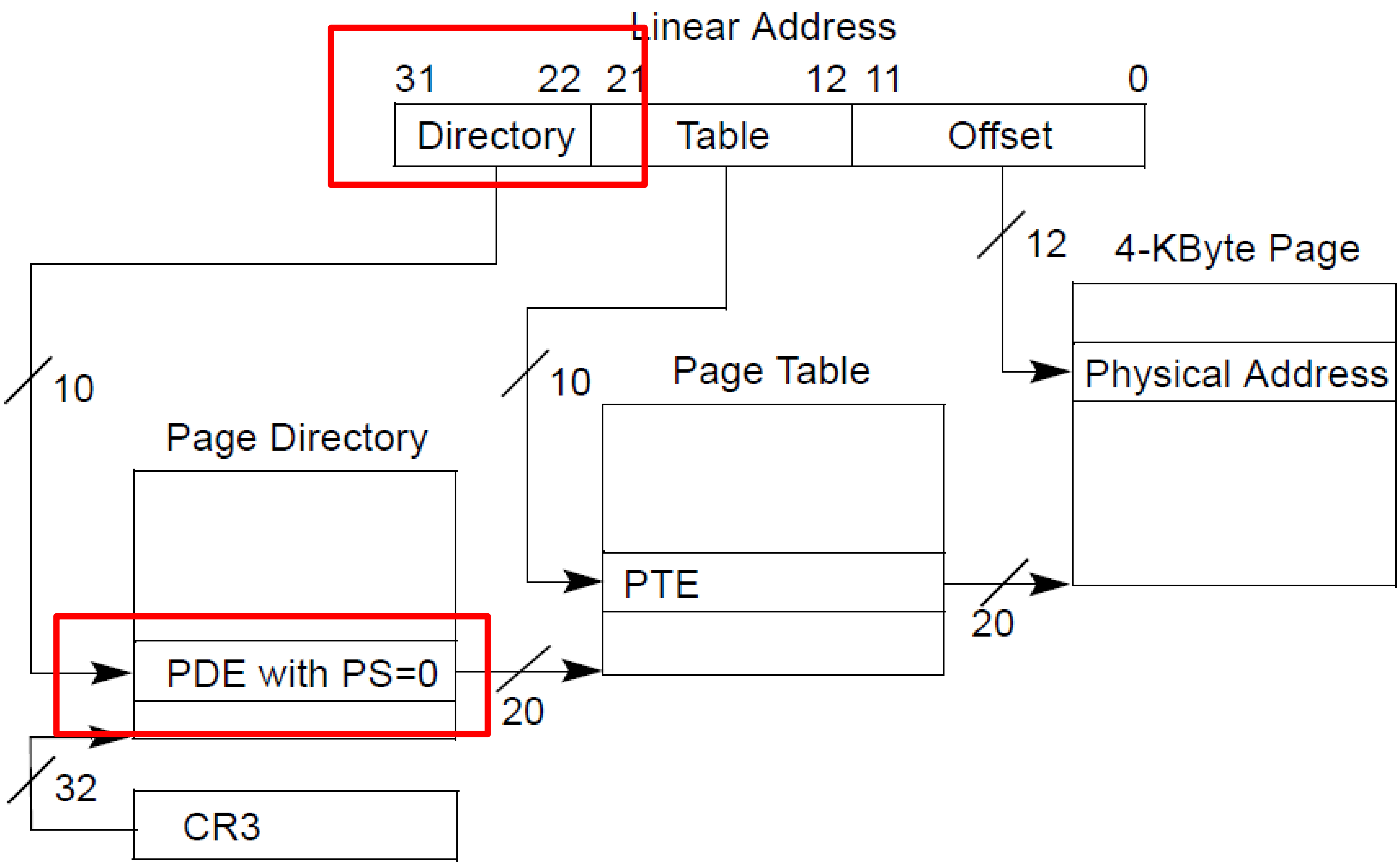
```
ds:x = 1; // data
ss:y;     // stack
if (ds:x) {
   ss:y = 1;
   cs:printf(ds:"Boo");
} else
   ss:y = 0;
```

Describe the linear to physical address translation with the paging mechanism (use provided diagram, mark and explain the steps).

Page translation



Page translation

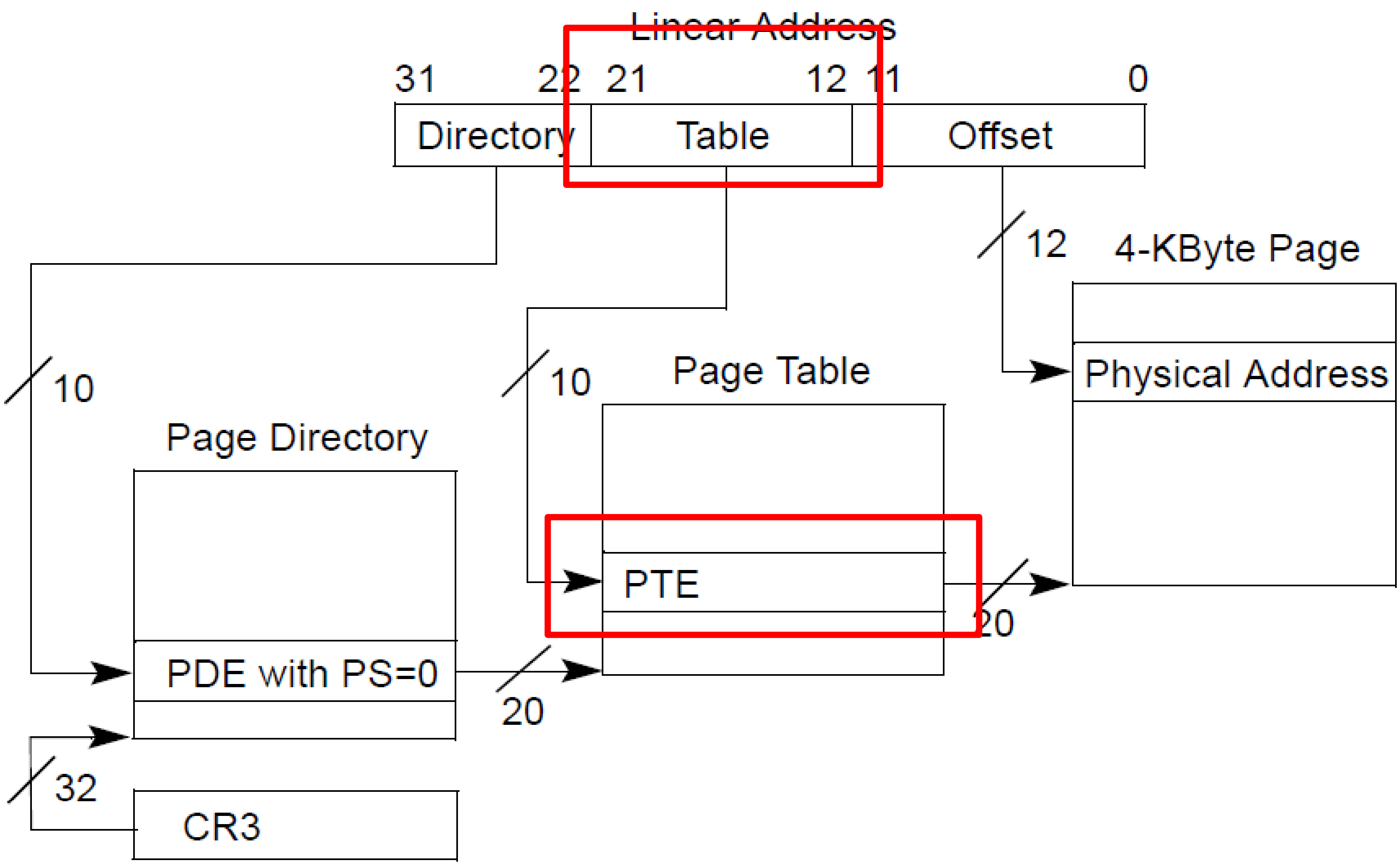


Page directory entry (PDE)

31 … 12	11 … 8	7	6	5	4	3	2	1	0	
Address of page table	Ignored	**0**	Ign	A	PCD	PWT	U/S	R/W	**1**	**PDE: page table**

- 20 bit address of the page table
  - Pages 4KB each, we need 1M to cover 4GB
- R/W – writes allowed?
  - To a 4MB region controlled by this entry
- U/S – user/supervisor
  - If 0 – user-mode access is not allowed
- A – accessed

Page translation

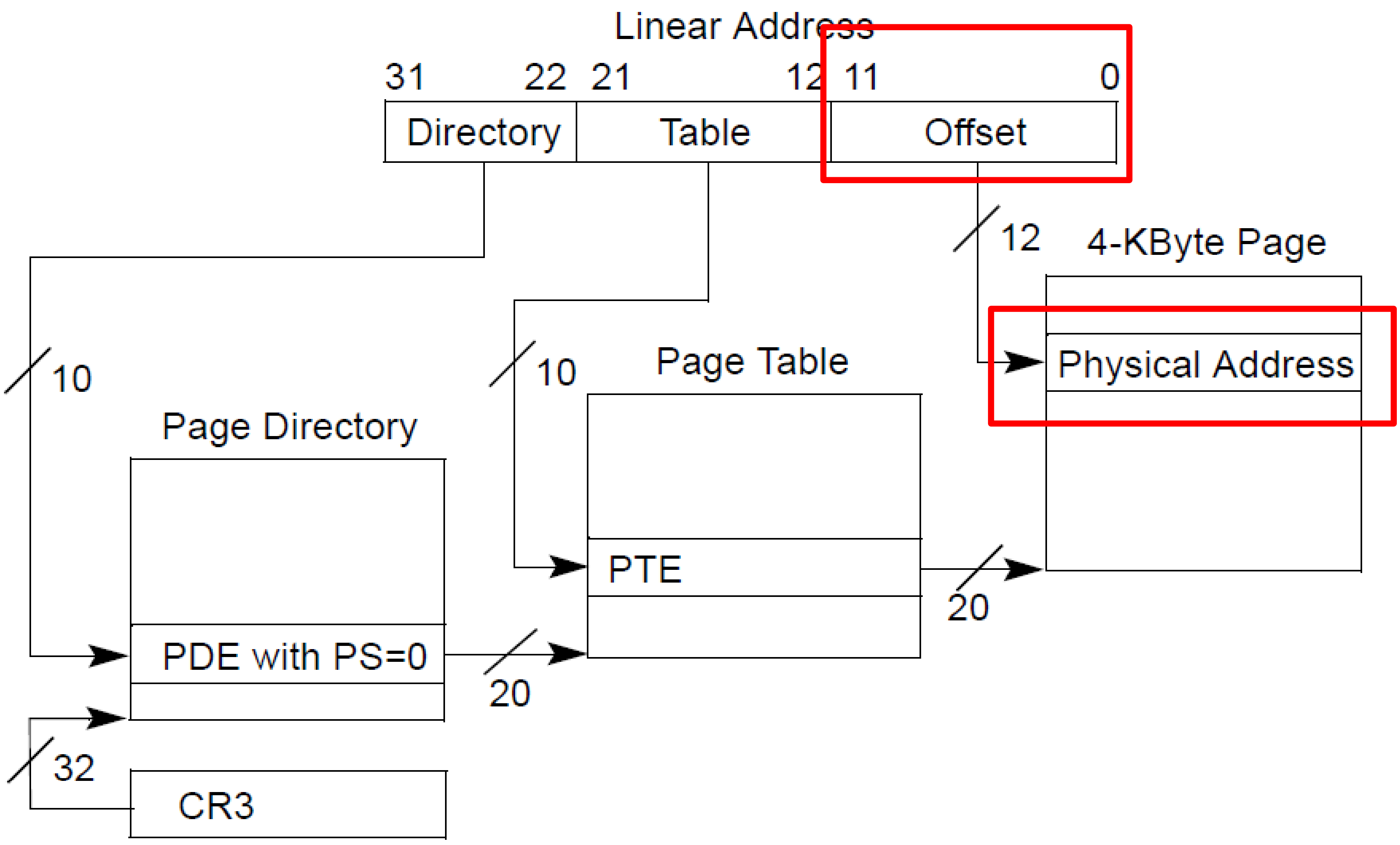


Page table entry (PTE)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12	11 10 9	8	7	6	5	4	3	2	1	0	
Address of 4KB page frame	Ignored	G	PAT	D	A	PCD	PWT	U/S	R/W	1	**PTE: 4KB page**

- 20 bit address of the 4KB page
  - Pages 4KB each, we need 1M to cover 4GB
- R/W – writes allowed?
  - To a 4KB page
- U/S – user/supervisor
  - If 0 user-mode access is not allowed
- A – accessed
- D – dirty – software has written to this page

Page translation



Consider the following 32-bit x86 page table setup.
%cr3 holds 0x00001000.
The Page Directory Page at physical address 0x00001000:

```
PDE 0: PPN=0x00002, PTE_P, PTE_U, PTE_W
PDE 1: PPN=0x00003, PTE_P, PTE_U, PTE_W
PDE 2: PPN=0x00002, PTE_P, PTE_U, PTE_W
```

... all other PDEs are zero
The Page Table Page at physical address 0x00002000 (which is PPN 0x00002):

```
PTE 0: PPN=0x00005, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00006, PTE_P, PTE_U, PTE_W
```

... all other PTEs are zero
The Page Table Page at physical address 0x00003000:

```
PTE 0: PPN=0x00005, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00005, PTE_P, PTE_U, PTE_W
```

... all other PTEs are zero

List all virtual addresses that map to physical address 0x00005555

Consider the following 32-bit x86 page table setup.
%cr3 holds 0x00001000.
The Page Directory Page at physical address 0x00001000:

```
PDE 0: PPN=0x00002, PTE_P, PTE_U, PTE_W
PDE 1: PPN=0x00003, PTE_P, PTE_U, PTE_W
PDE 2: PPN=0x00002, PTE_P, PTE_U, PTE_W
```

... all other PDEs are zero
The Page Table Page at physical address 0x00002000 (which is PPN 0x00002):

```
PTE 0: PPN=0x00005, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00006, PTE_P, PTE_U, PTE_W
```

... all other PTEs are zero
The Page Table Page at physical address 0x00003000:

```
PTE 0: PPN=0x00005, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00005, PTE_P, PTE_U, PTE_W
```

... all other PTEs are zero

List all virtual addresses that map to physical address 0x00005555
Answer: 0x00000555, 0x00400555, 0x00401555, 0x00800555

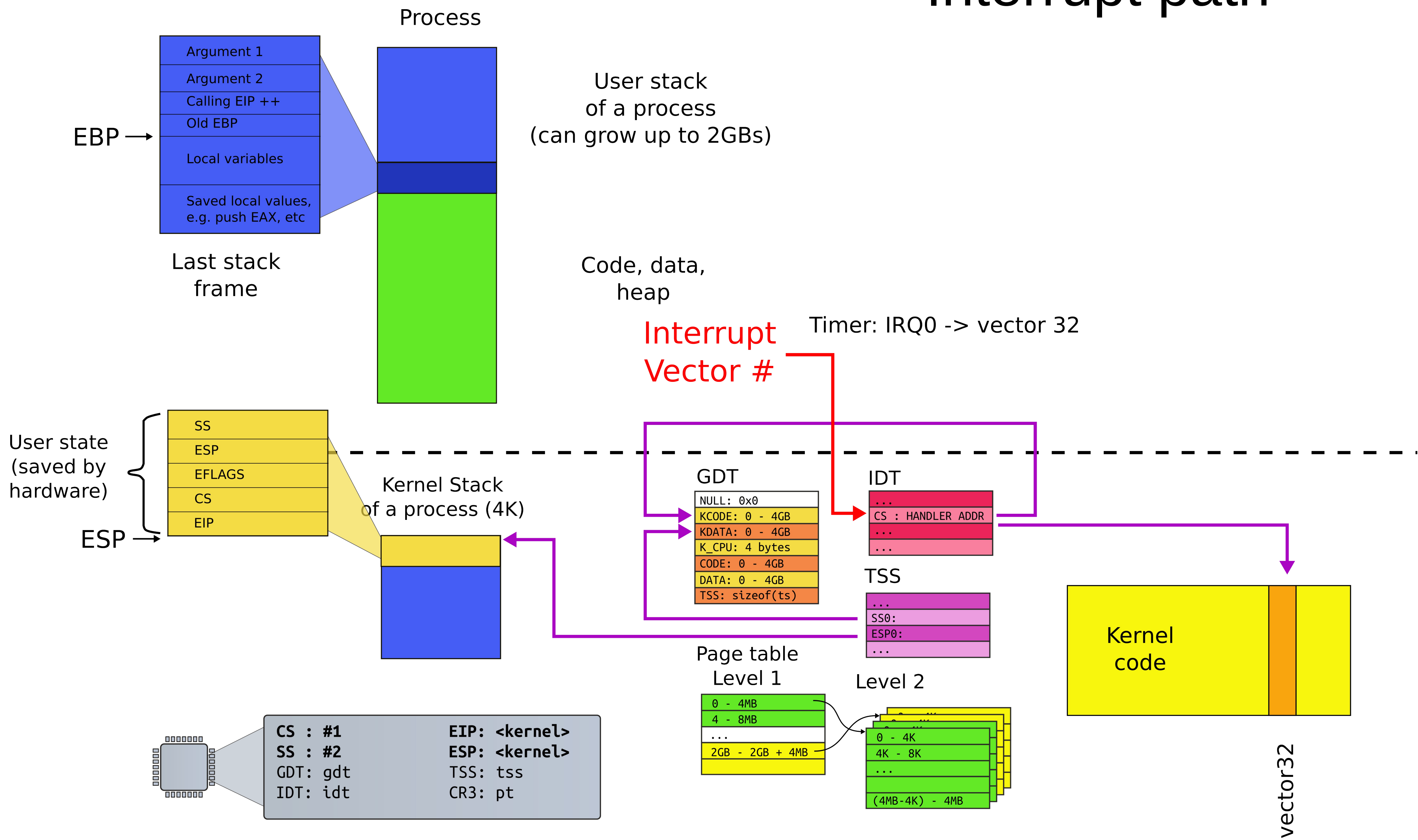
What's on the stack? Describe layout of a stack and how it changes during function invocation?

Example stack

```
:    :
| 10 | [ebp + 16] (3rd function argument)
|  5 | [ebp + 12] (2nd argument)
|  2 | [ebp + 8]  (1st argument)
| RA | [ebp + 4]  (return address)
| FP | [ebp]      (old ebp value)
|    | [ebp - 4]  (1st local variable)
:    :
:    :
|    | [ebp - X]  (esp - the current stack pointer)
```

Describe the steps and data structures involved into a user to kernel transition (draw diagrams)

Interrupt path



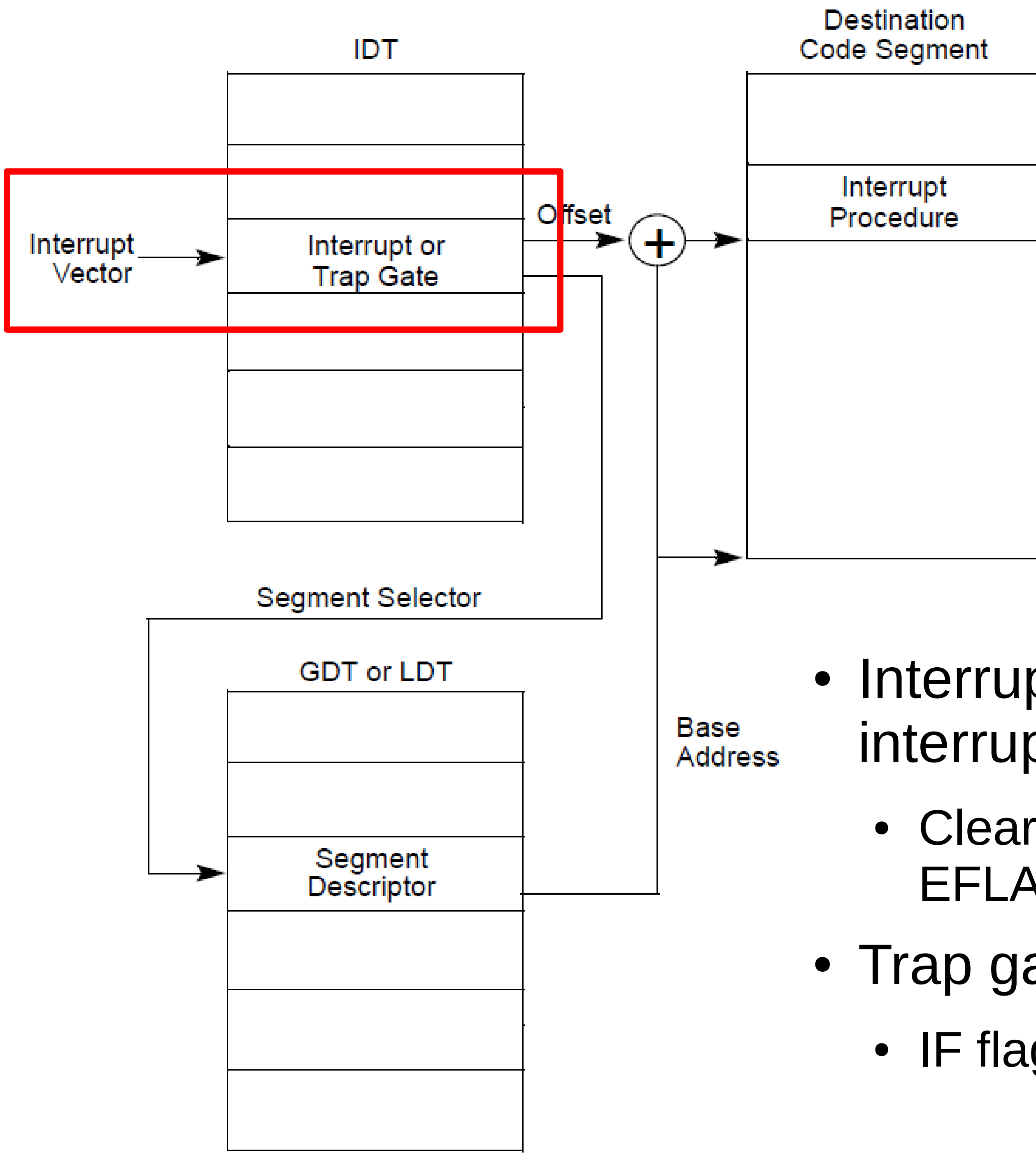
What segment is specified in the interrupt descriptor? Why?

Interrupt descriptor

Interrupt Gate

31–16	15	14–13	12–8	7–5	4–0	
Offset 31..16	P	DPL	0 D 1 1 0	0 0 0		4

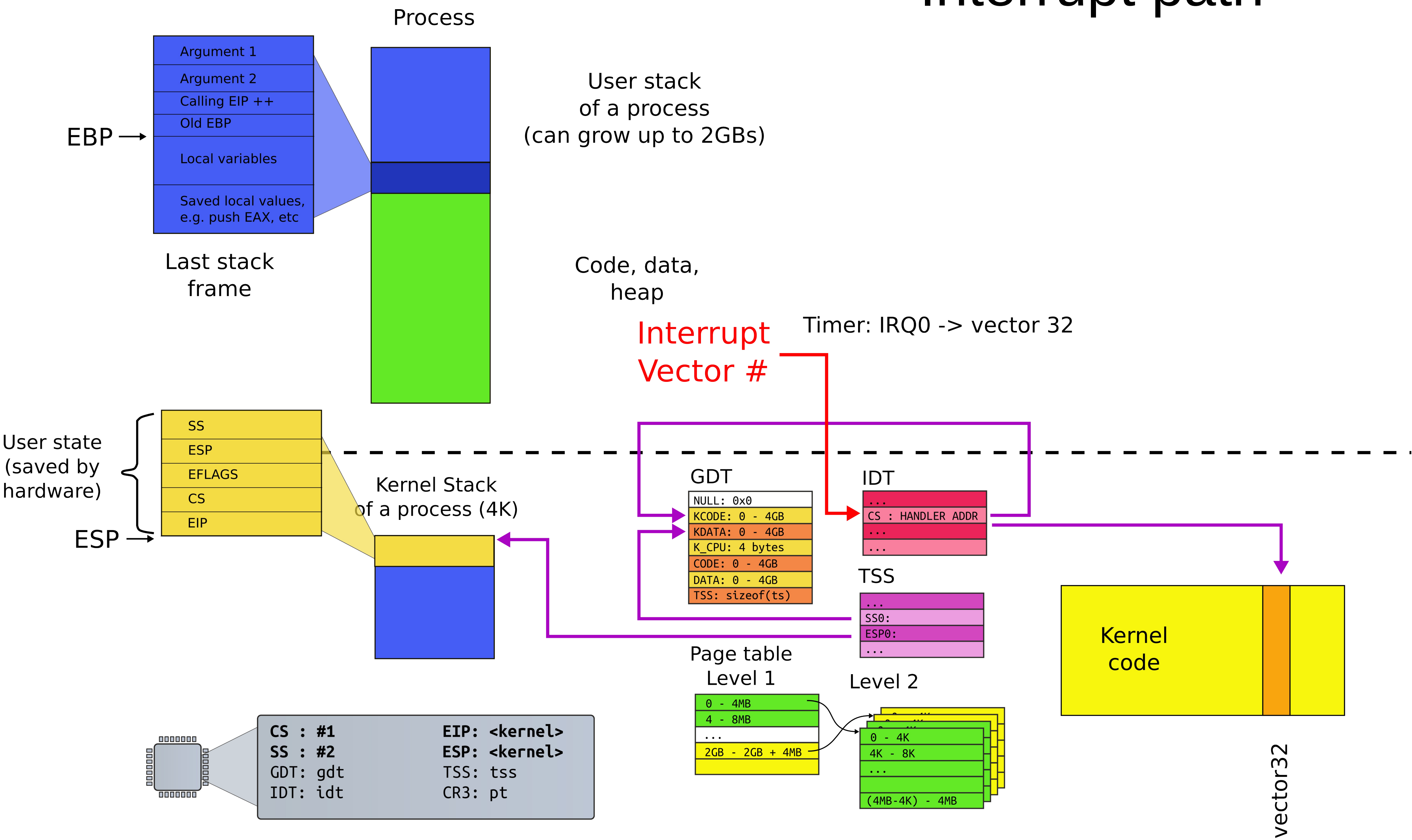
31–16	15–0	
Segment Selector	Offset 15..0	0



- Interrupt gate disables interrupts
  - Clears the IF flag in EFLAGS register
- Trap gate doesn't
  - IF flag is unchanged

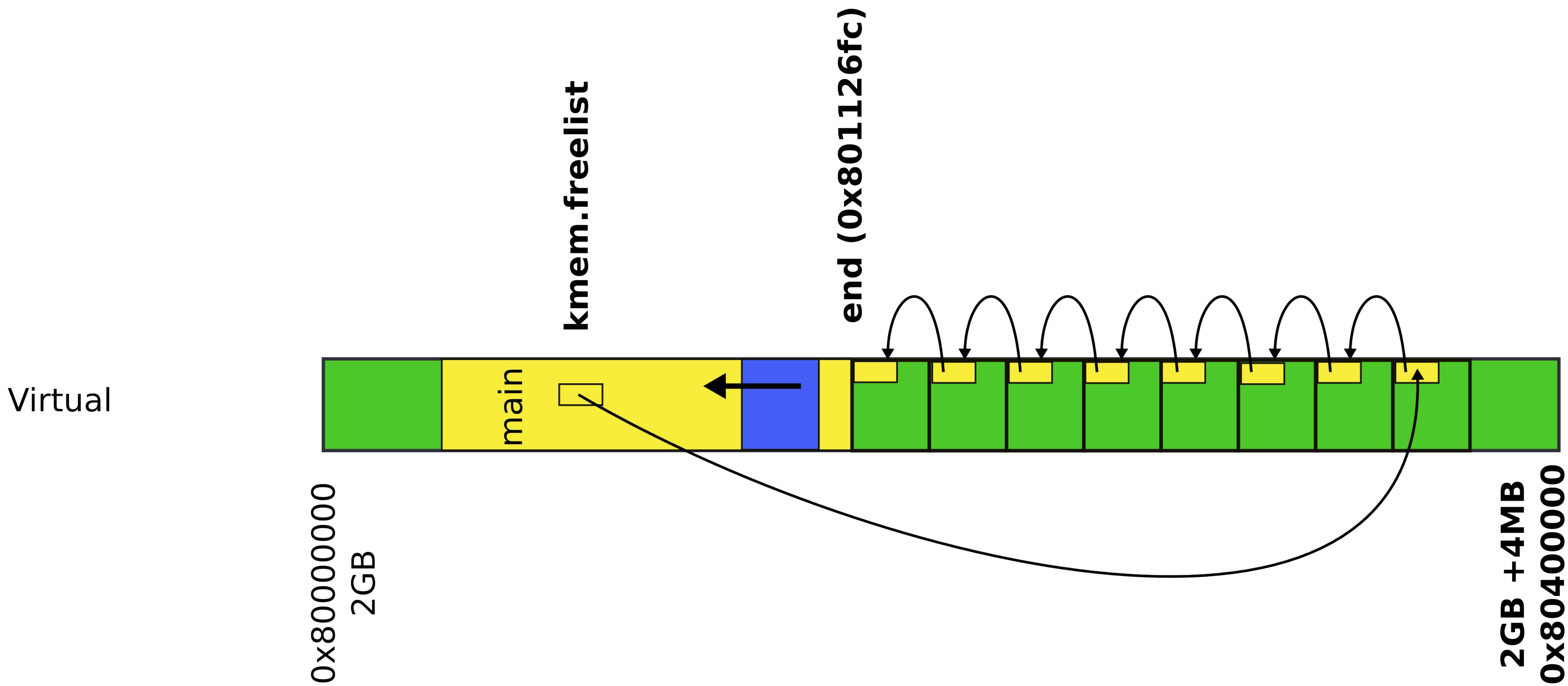
Which stack is used for execution of an interrupt handler? How does hardware find it?

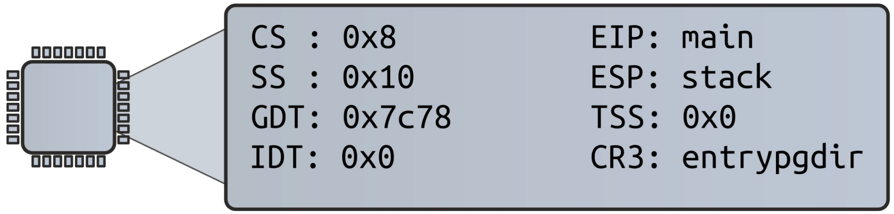
Interrupt path



Describe organization of the memory allocator in xv6?

Physical page allocator





Protected Mode

Where did free memory came from?

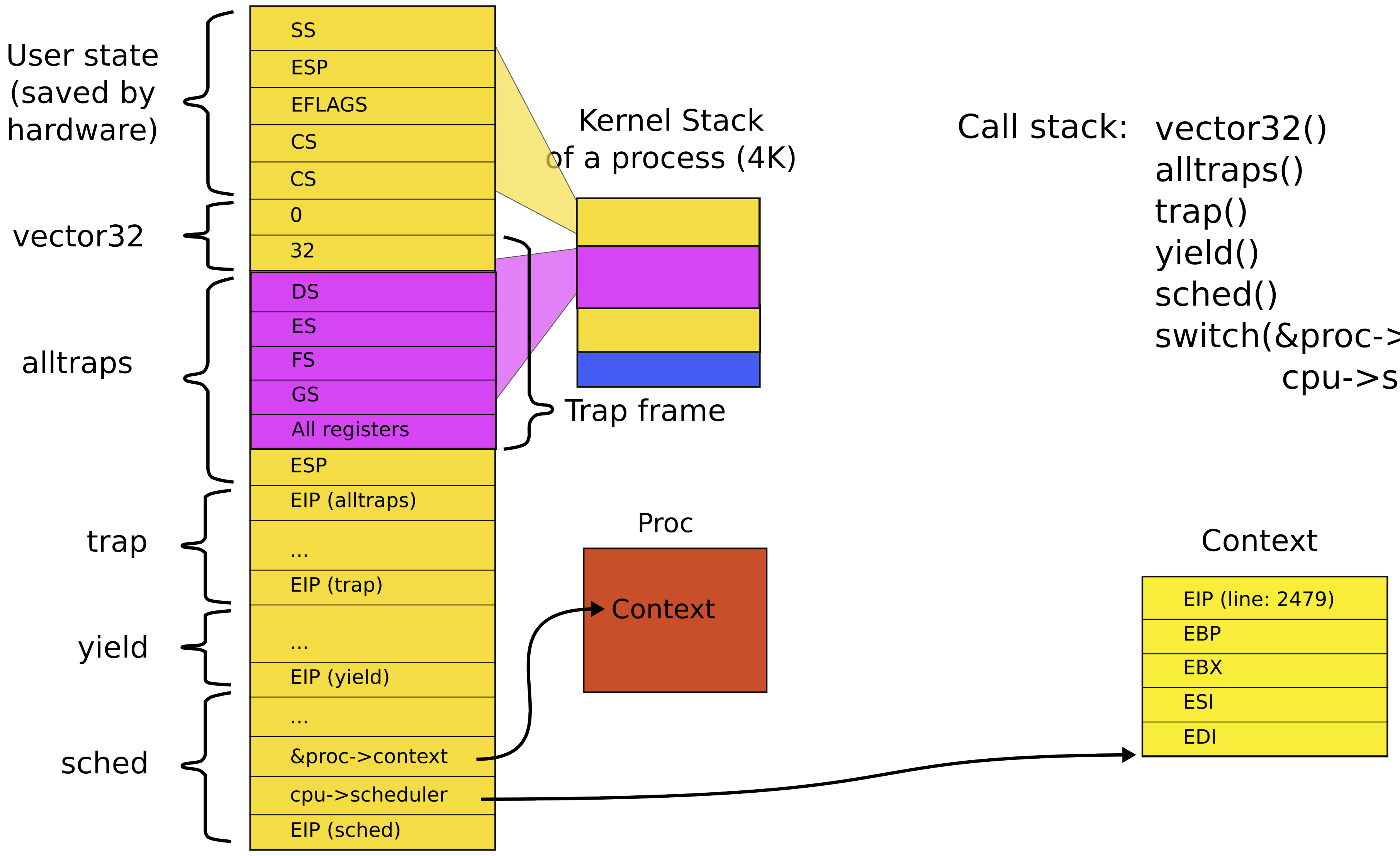
`swtch` in xv6 doesn't explicitly save and restore all fields of struct context. Why is it okay that swtch doesn't contain any code that saves `%eip`?

swtch()

```
swtch:
movl 4(%esp), %eax
movl 8(%esp), %edx

# Save old callee-save registers
pushl %ebp
pushl %ebx
pushl %esi
pushl %edi

# Switch stacksh
movl %esp, (%eax)
movl %edx, %esp

# Load new callee-save registers
popl %edi
popl %esi
popl %ebx
popl %ebp
ret
```

```
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

Stack inside swtch()



Call stack: vector32()
alltraps()
trap()
yield()
sched()
switch(&proc->context,
cpu->scheduler)

Suppose you wanted to change the system call interface in xv6 so that, instead of returning the system call result in EAX, the kernel pushed the result on to the user space stack. Fill in the code below to implement this. For the purposes of this question, you can assume that the user stack pointer points to valid memory.

```
void
syscall(void)
{
  int num;

  num = proc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    proc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
    proc->pid, proc->name, num);
    proc->tf->eax = -1;
  }
}
```

```
void
syscall(void)
{
  int num;

  num = proc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    // proc->tf->eax = syscalls[num]();
    proc->tf->esp -= 4;
    *(int*)ptoc->tf->esp = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            proc->pid, proc->name, num);
    // proc->tf->eax = -1;
    proc->tf->esp -= 4;
    *(int*)ptoc->tf->esp = -1;
  }
}
```

Thank you!