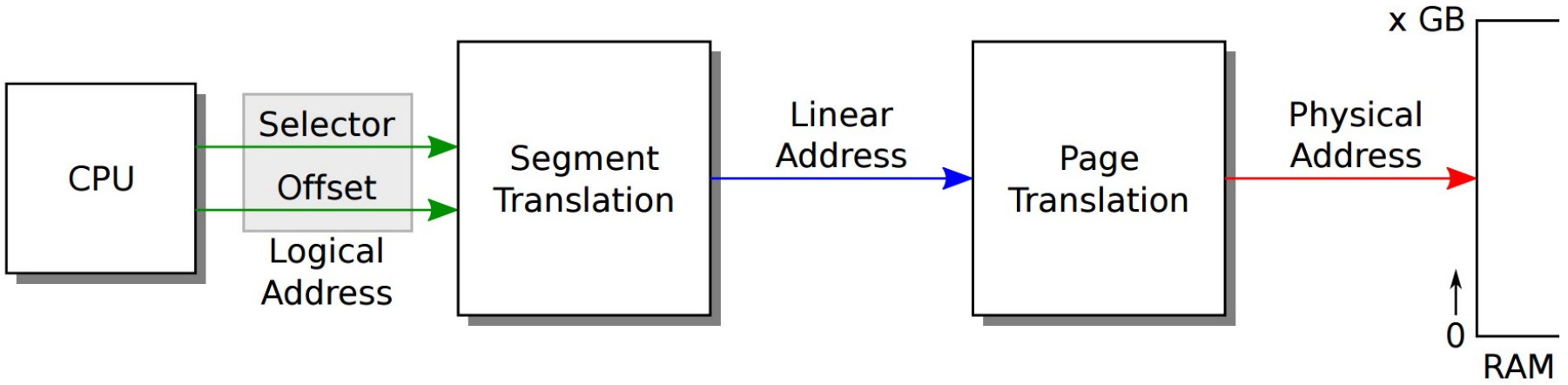
# ICS143A: Principles of Operating Systems

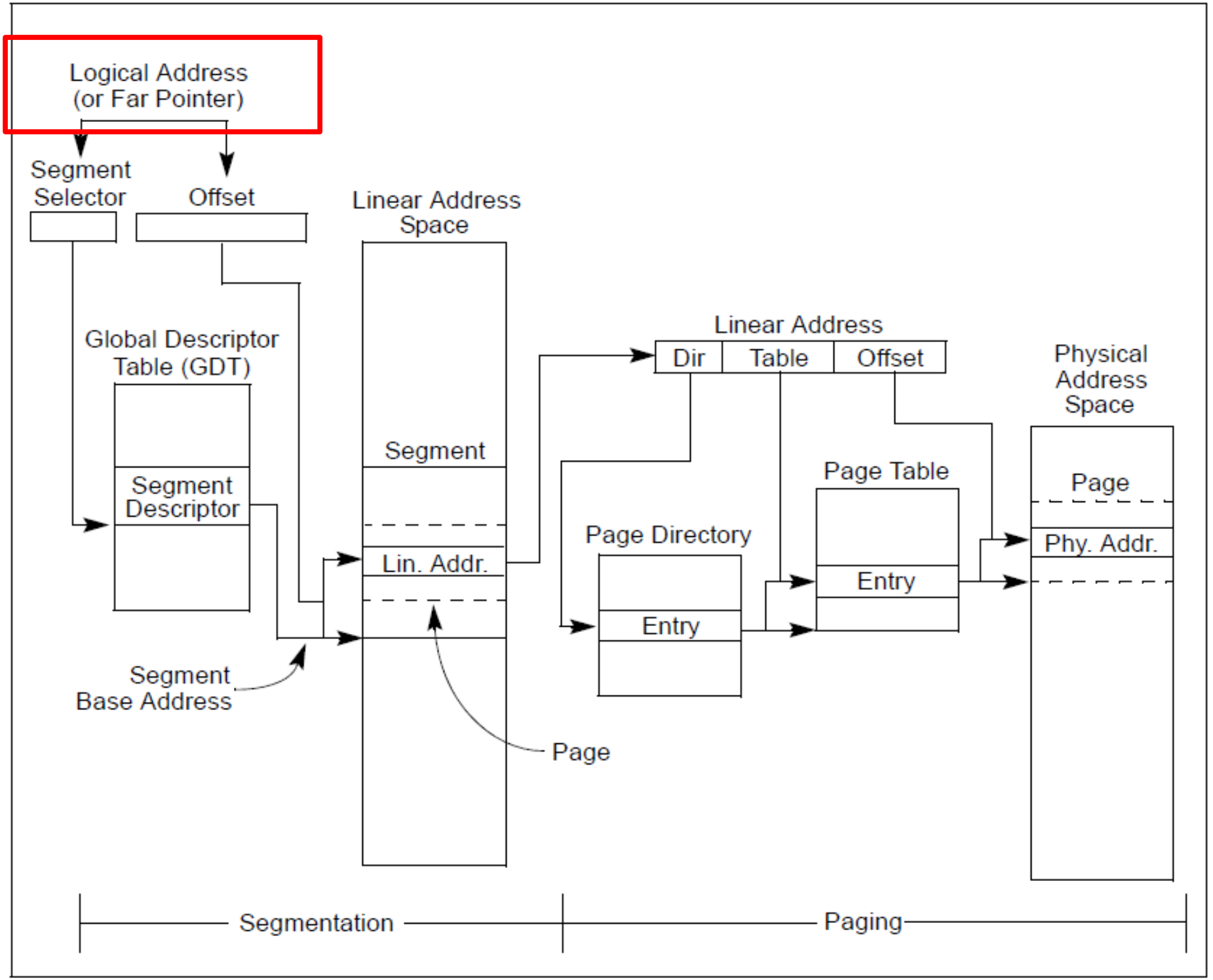
# Midterm recap, sample questions

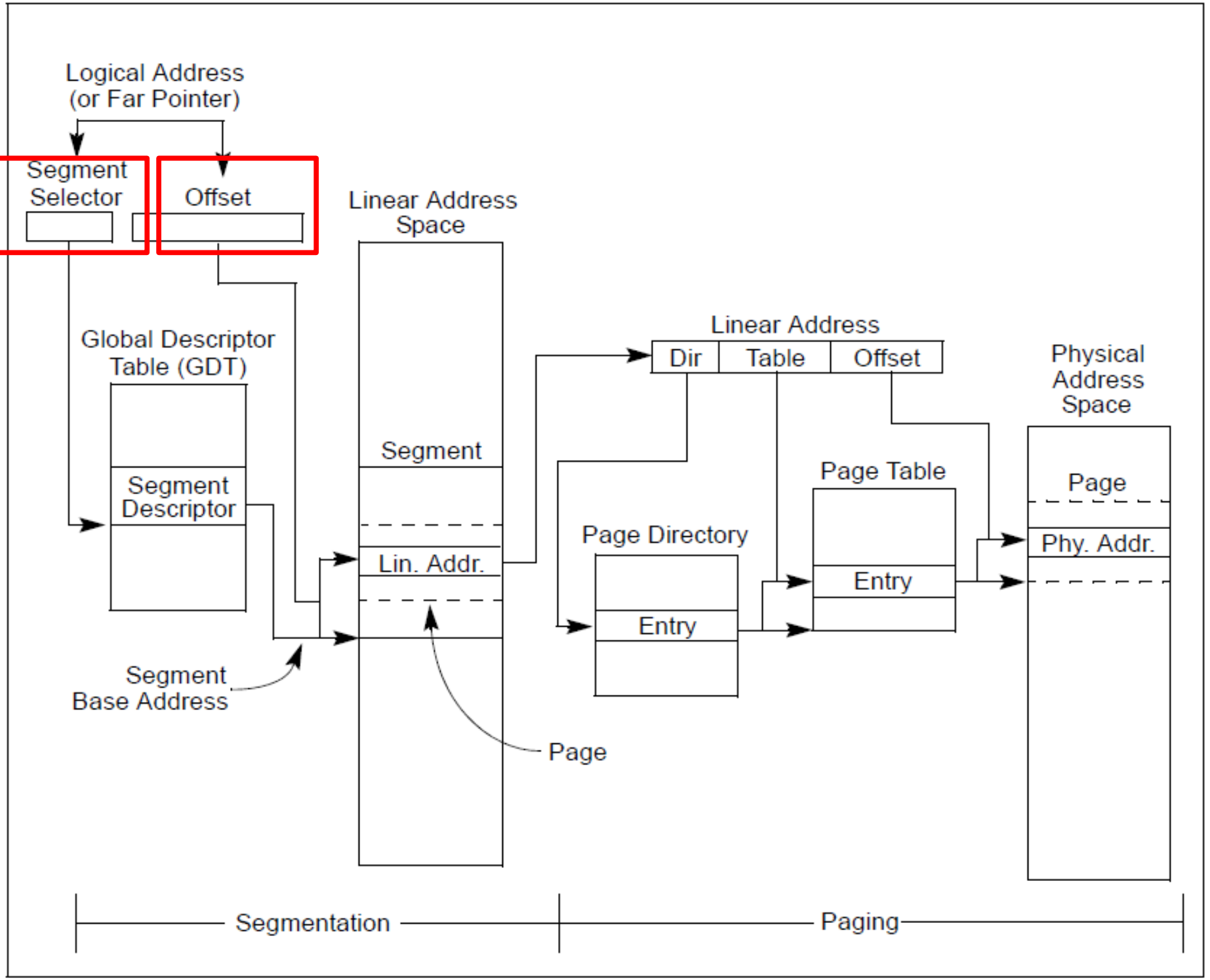
Anton Burtsev
February, 2017

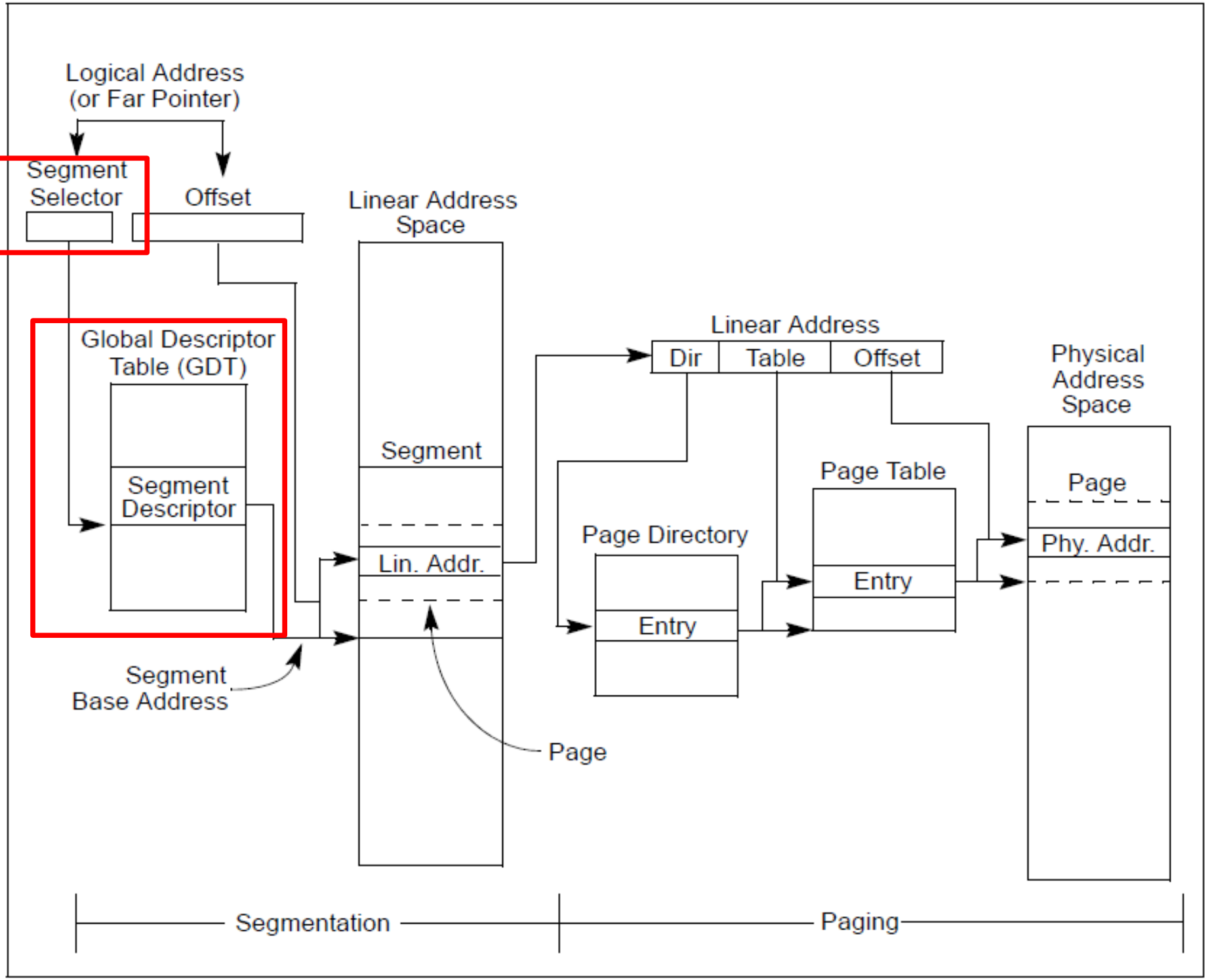
Describe the x86 address translation pipeline (draw figure), explain stages.

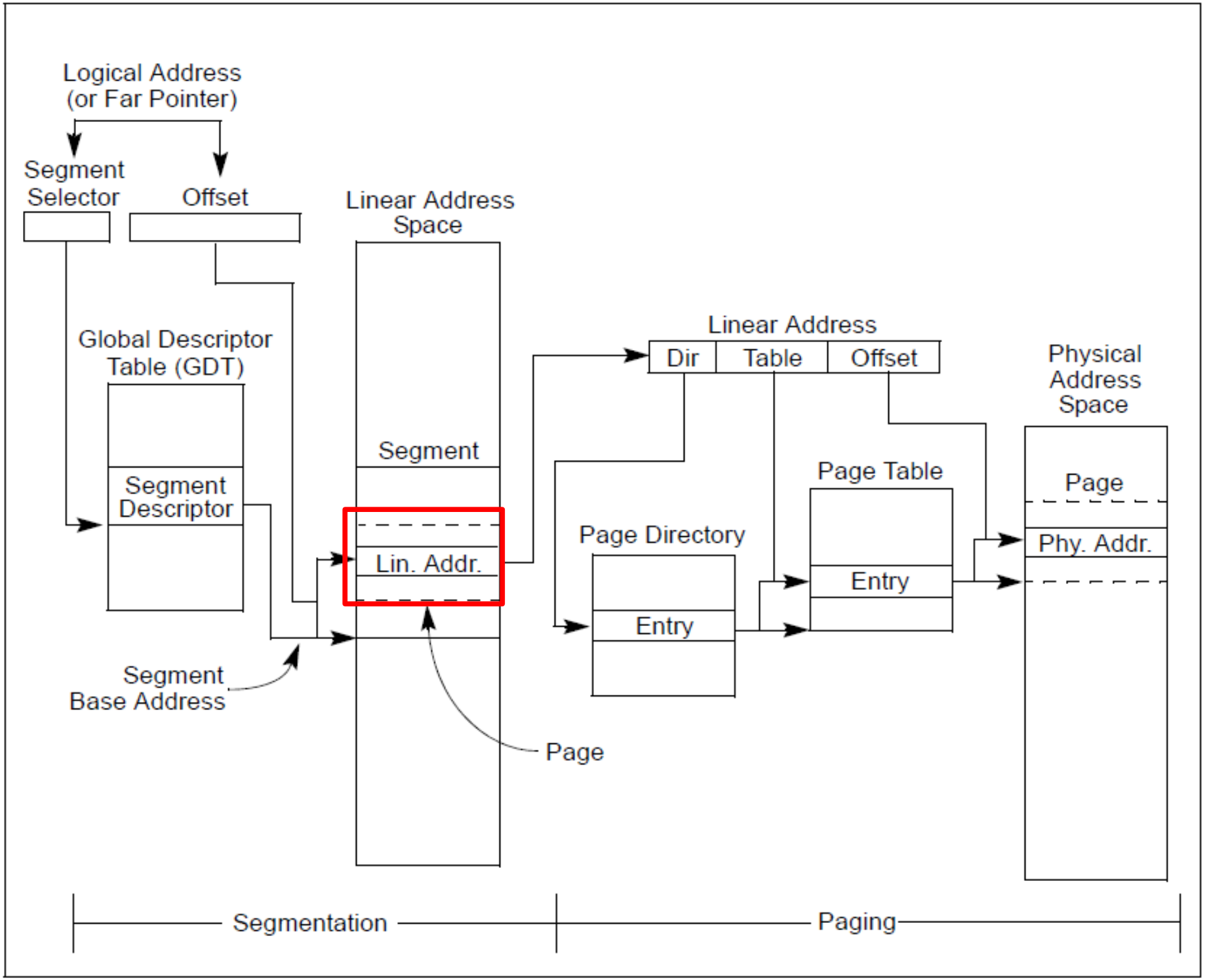
# Address translation

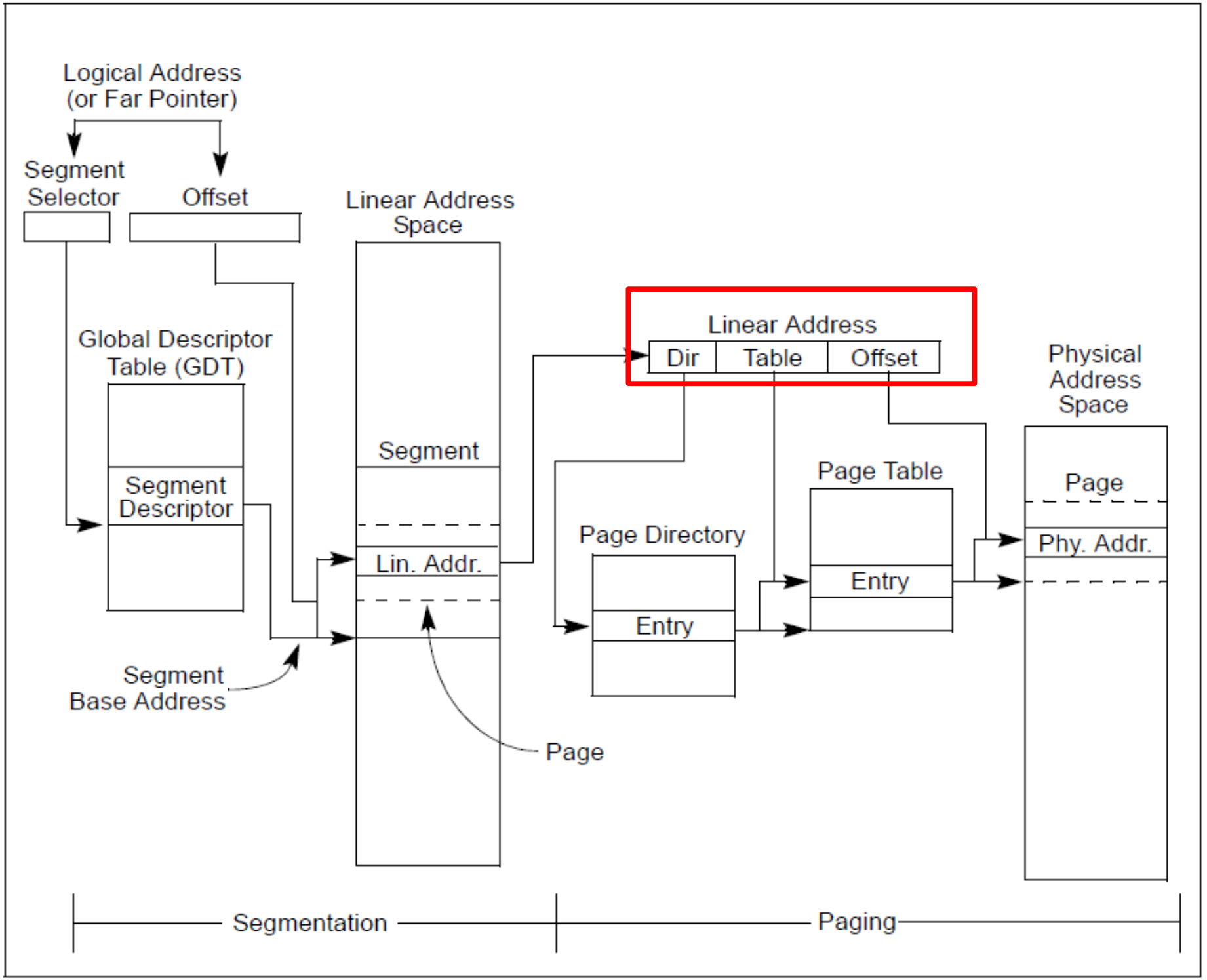


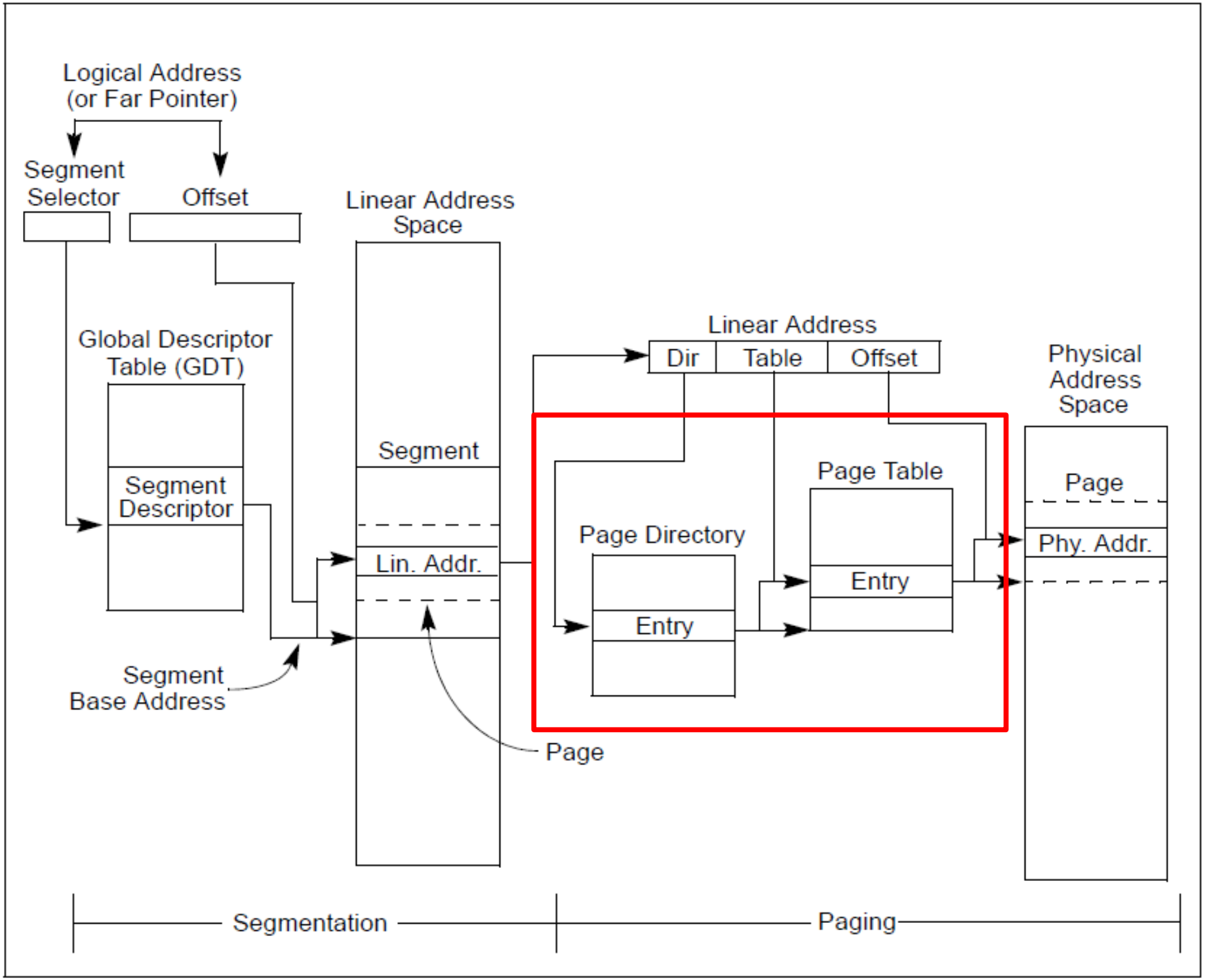

Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Linear Address
Dir
Table
Offset
Physical
Address
Space
Page Table
Page
Phy. Addr.
Page Directory
Entry
Entry
Segment
Base Address
Page
Segmentation
Paging

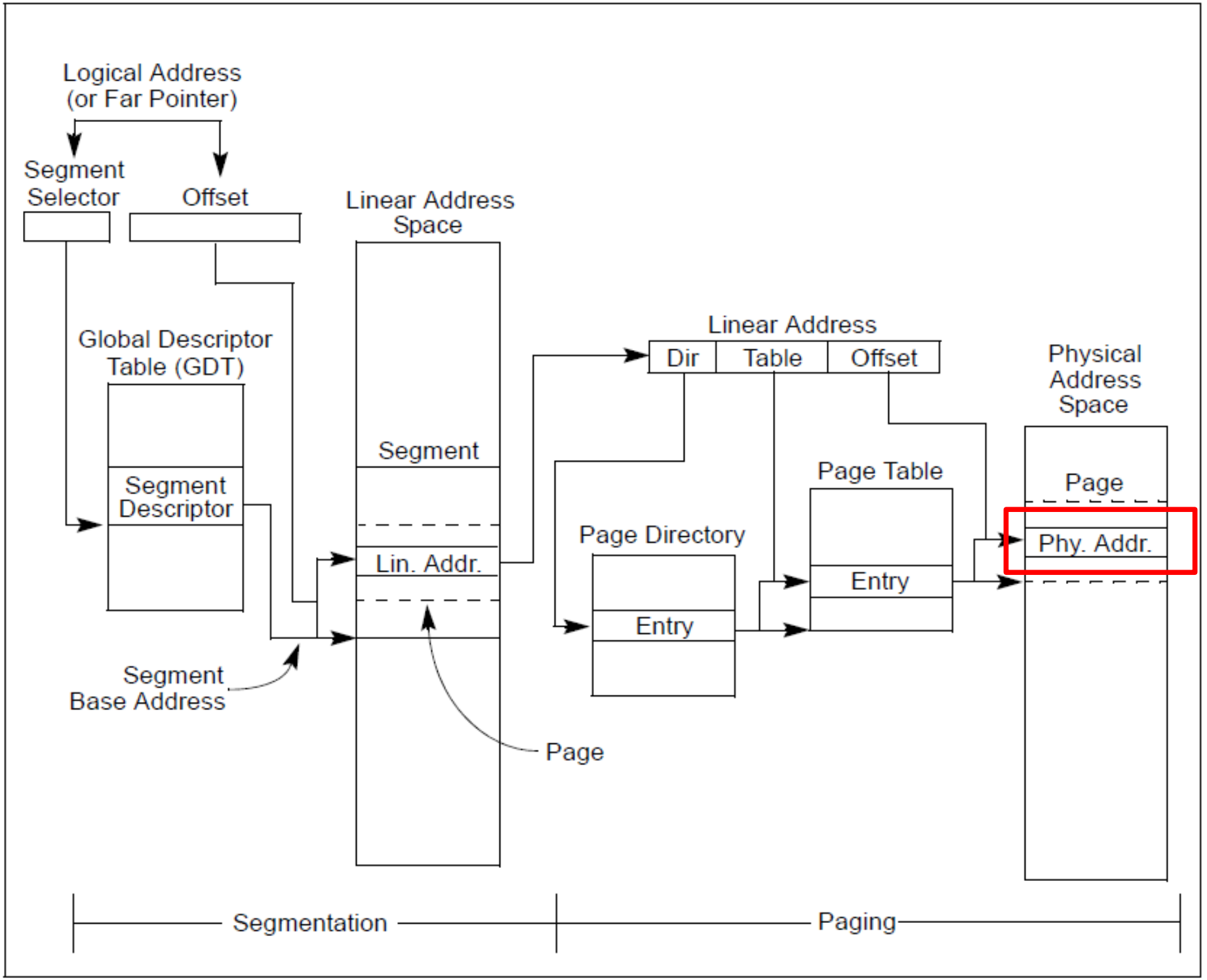

Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Linear Address
Dir
Table
Offset
Physical
Address
Space
Page Table
Page
Phy. Addr.
Entry
Page Directory
Entry
Segment
Base Address
Page
Segmentation
Paging

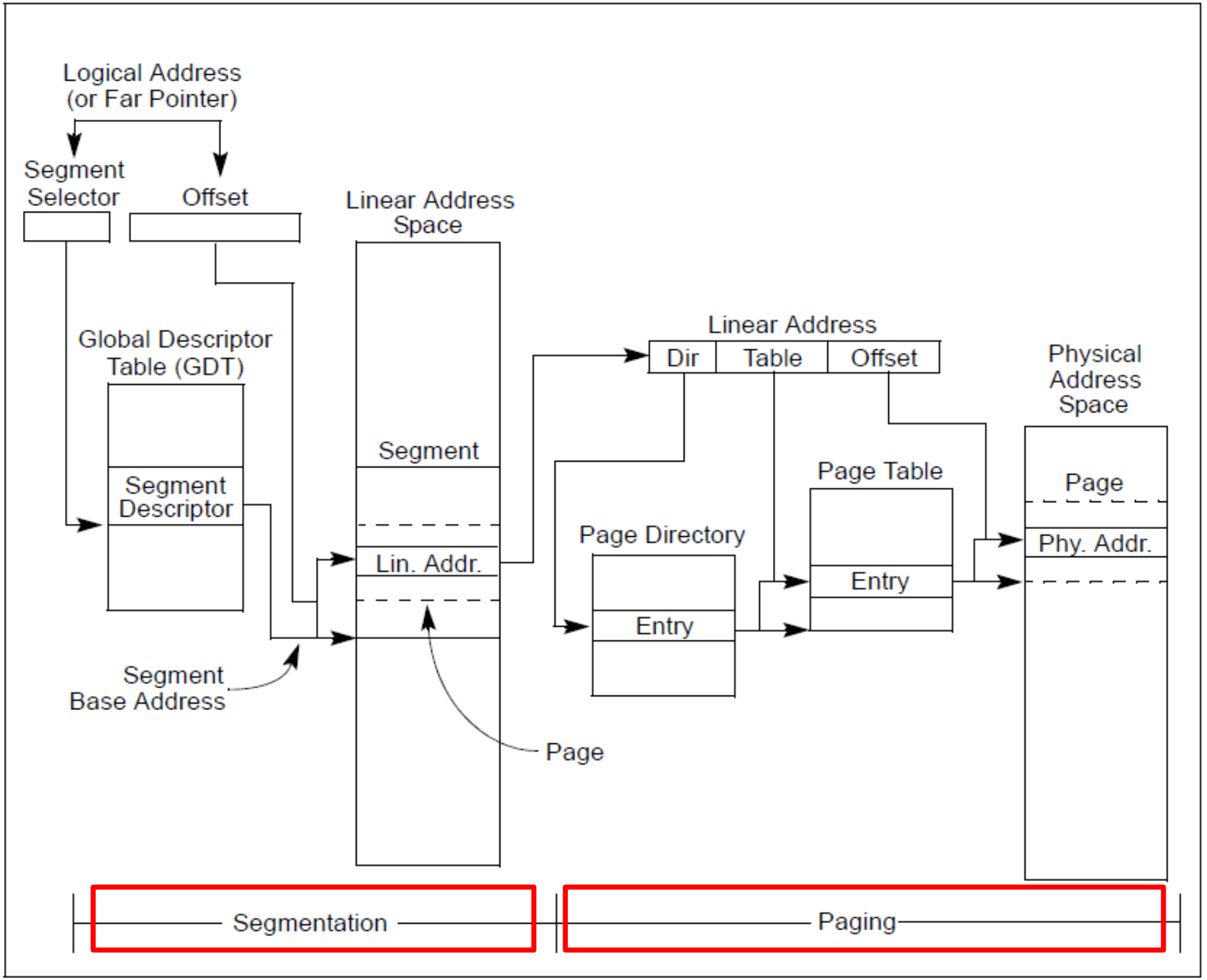

Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Linear Address
Dir
Table
Offset
Physical
Address
Space
Page Table
Page
Page Directory
Phy. Addr.
Entry
Entry
Segment
Base Address
Page
Segmentation
Paging


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

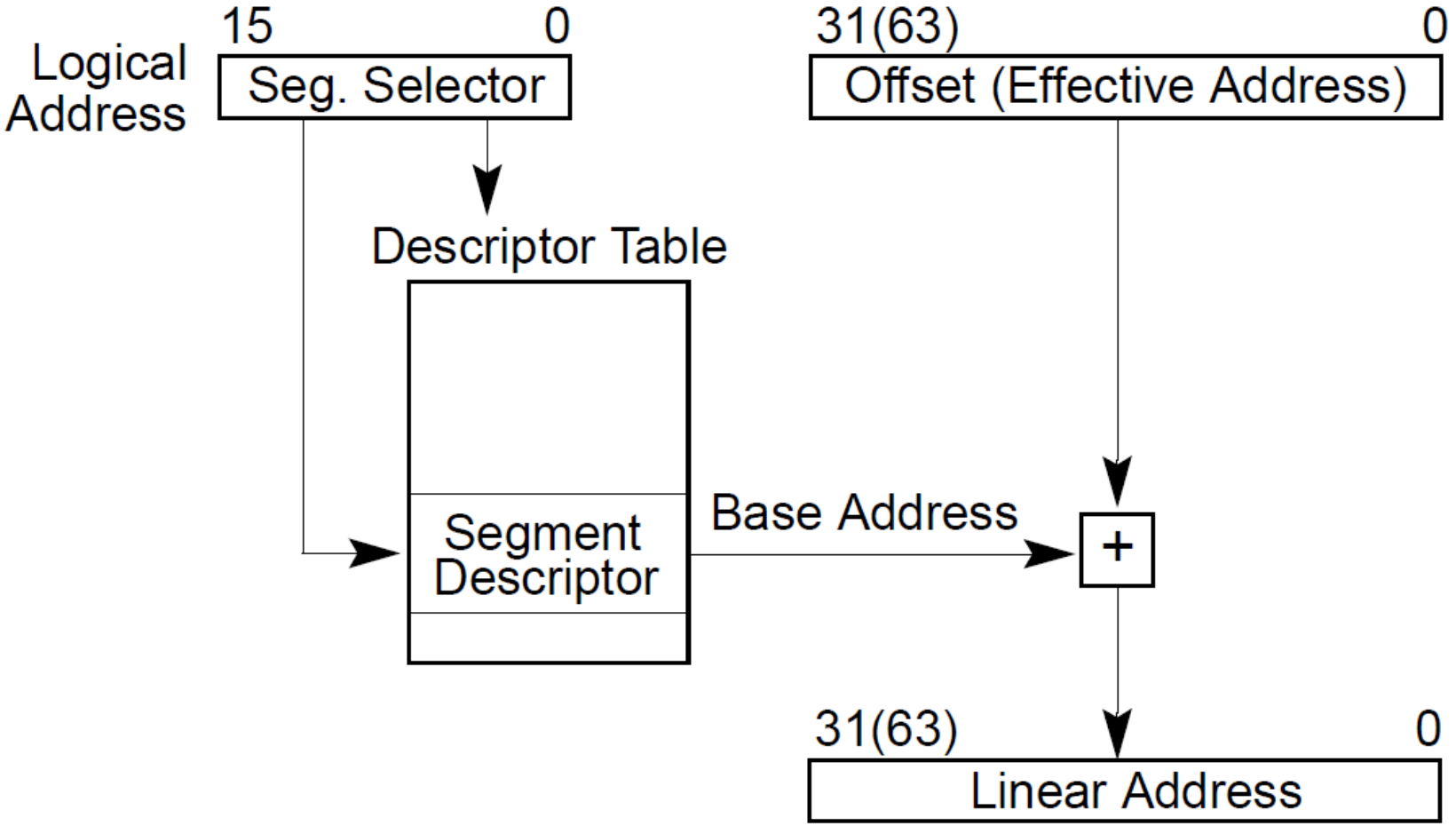

Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Linear Address
Dir
Table
Offset
Physical
Address
Space
Page Table
Page
Phy. Addr.
Page Directory
Entry
Entry
Segment
Base Address
Page
Segmentation
Paging


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)

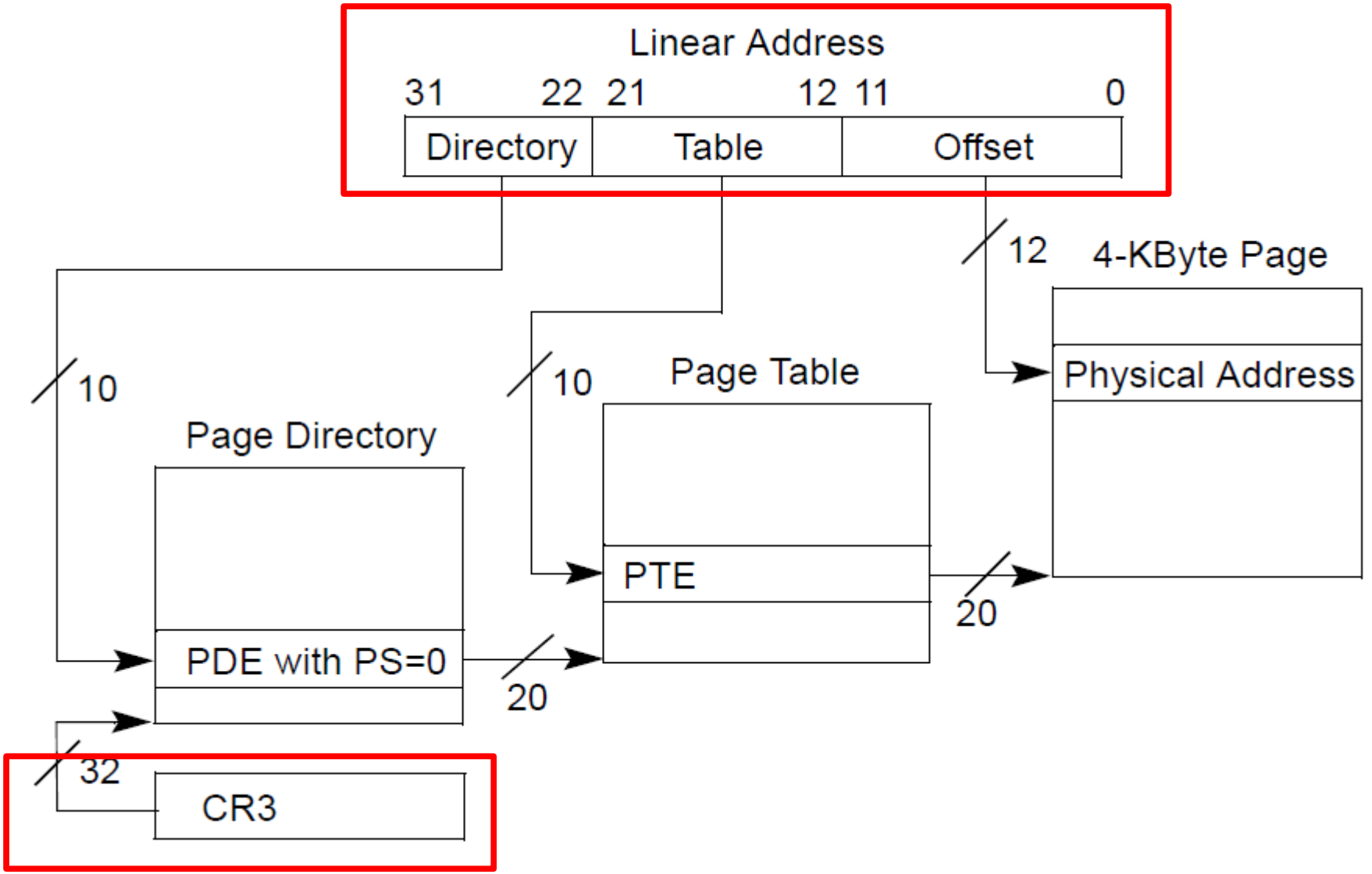
# Segmented programming (not real)

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

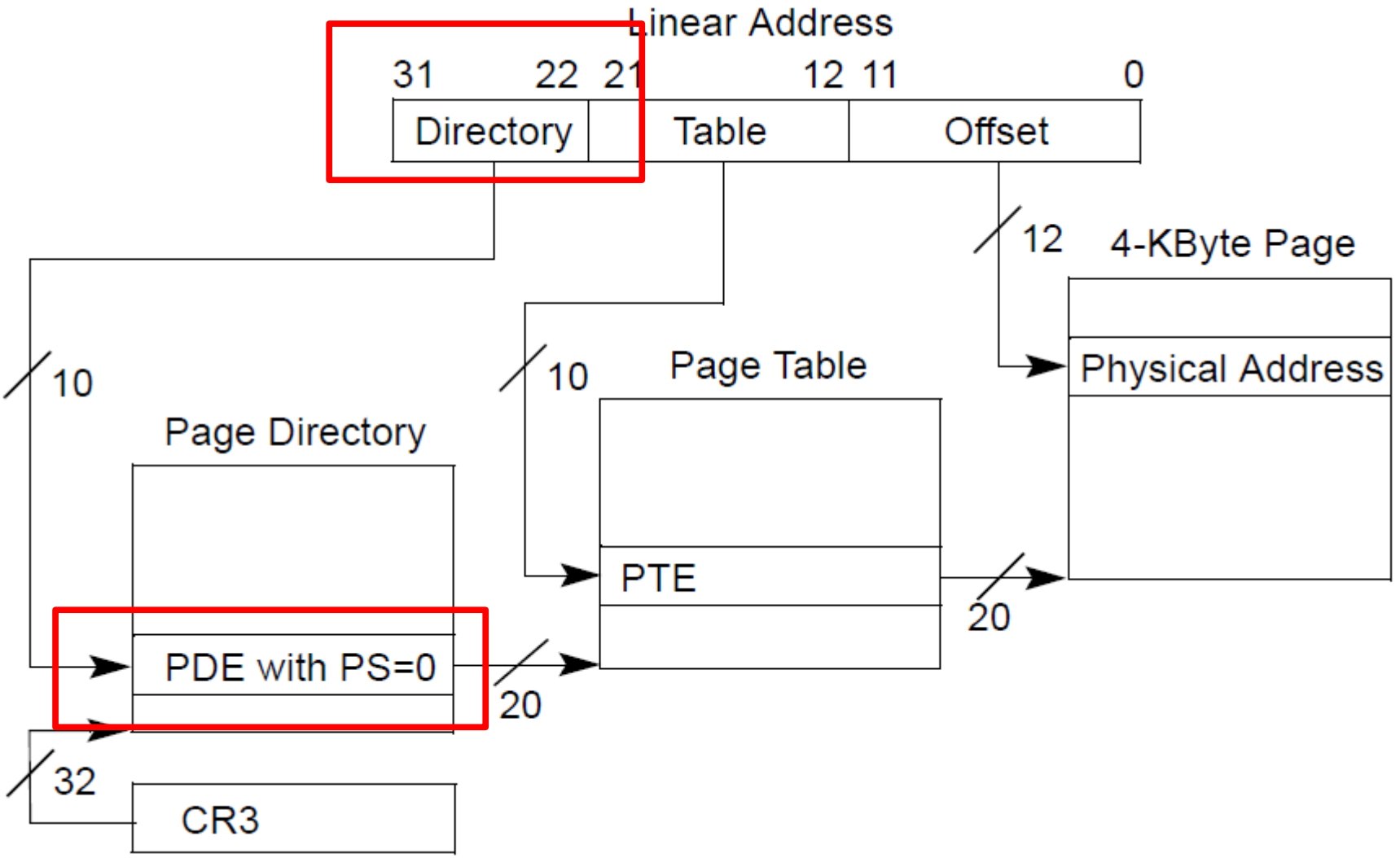
```
ds:x = 1; // data
ss:y;     // stack
if (ds:x) {
   ss:y = 1;
   cs:printf(ds:“Boo”);
} else
   ss:y = 0;
```

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

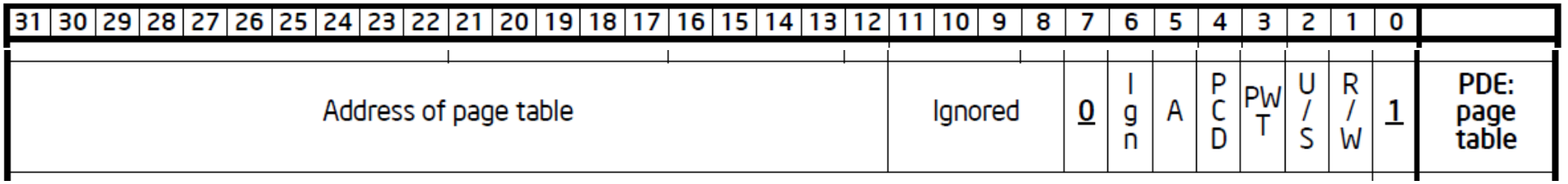
# Page translation



# Page translation

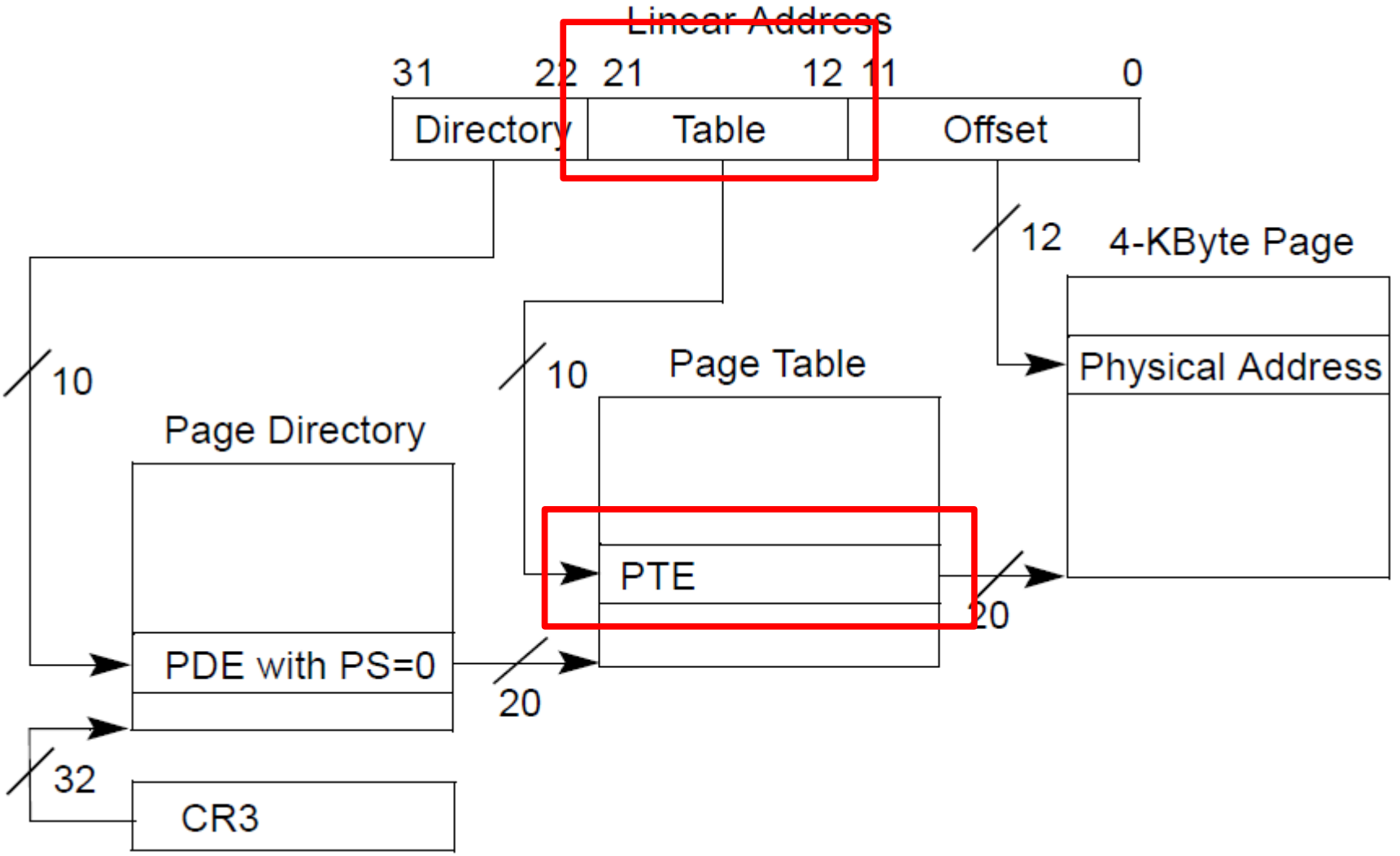


# Page directory entry (PDE)

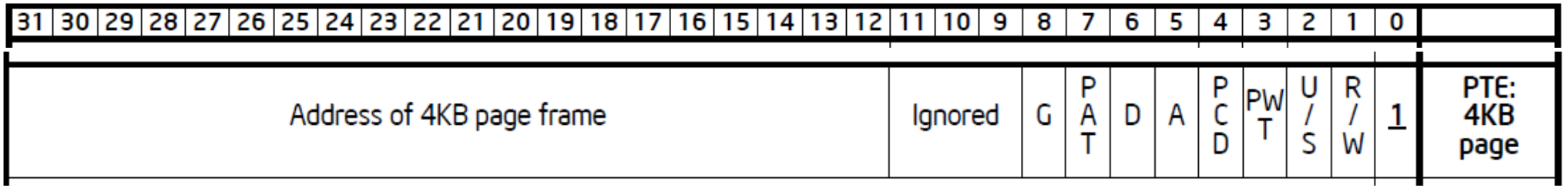



- 20 bit address of the page table
  - Pages 4KB each, we need 1M to cover 4GB
- R/W – writes allowed?
  - To a 4MB region controlled by this entry
- U/S – user/supervisor
  - If 0 – user-mode access is not allowed
- A – accessed

# Page translation

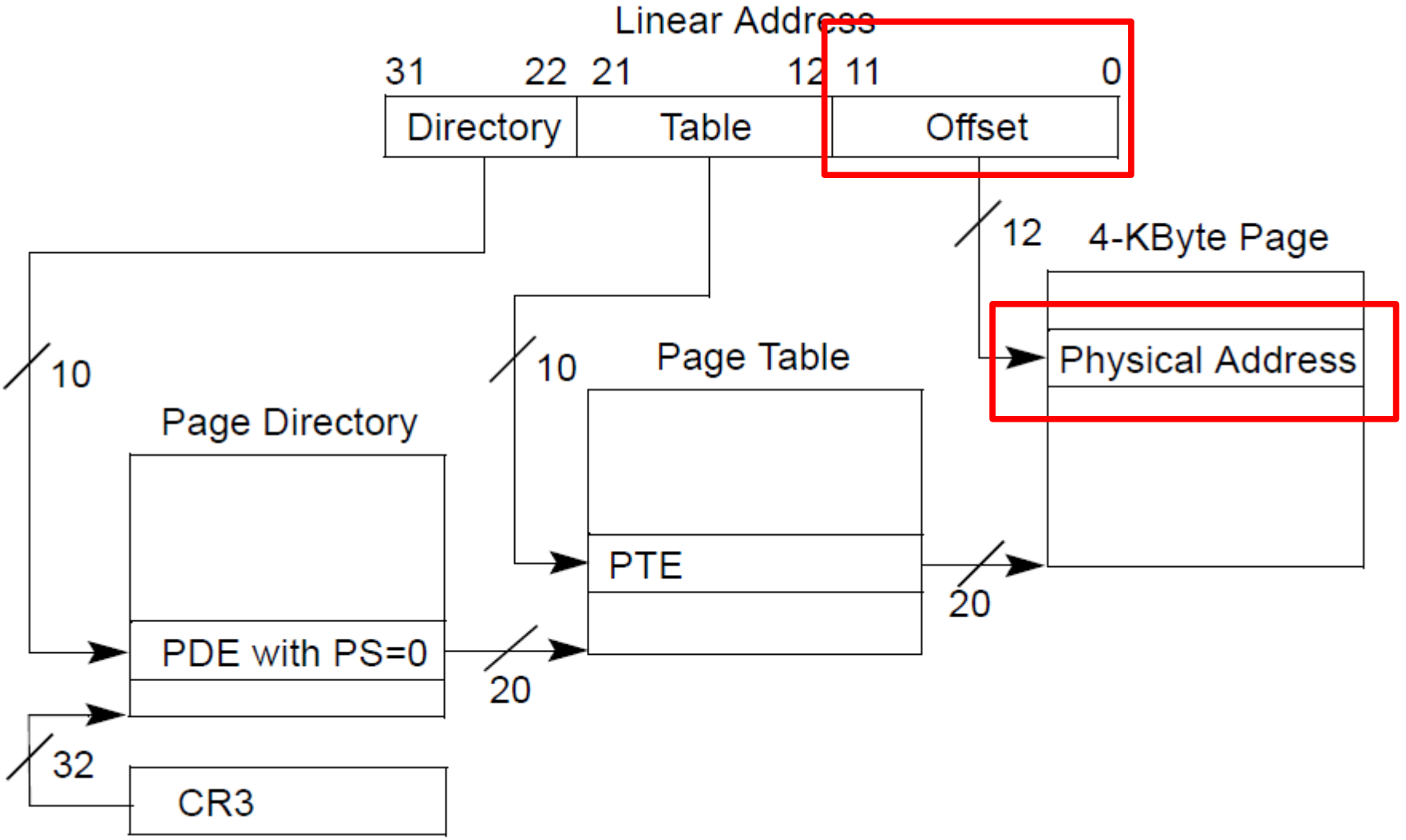


# Page table entry (PTE)




- 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

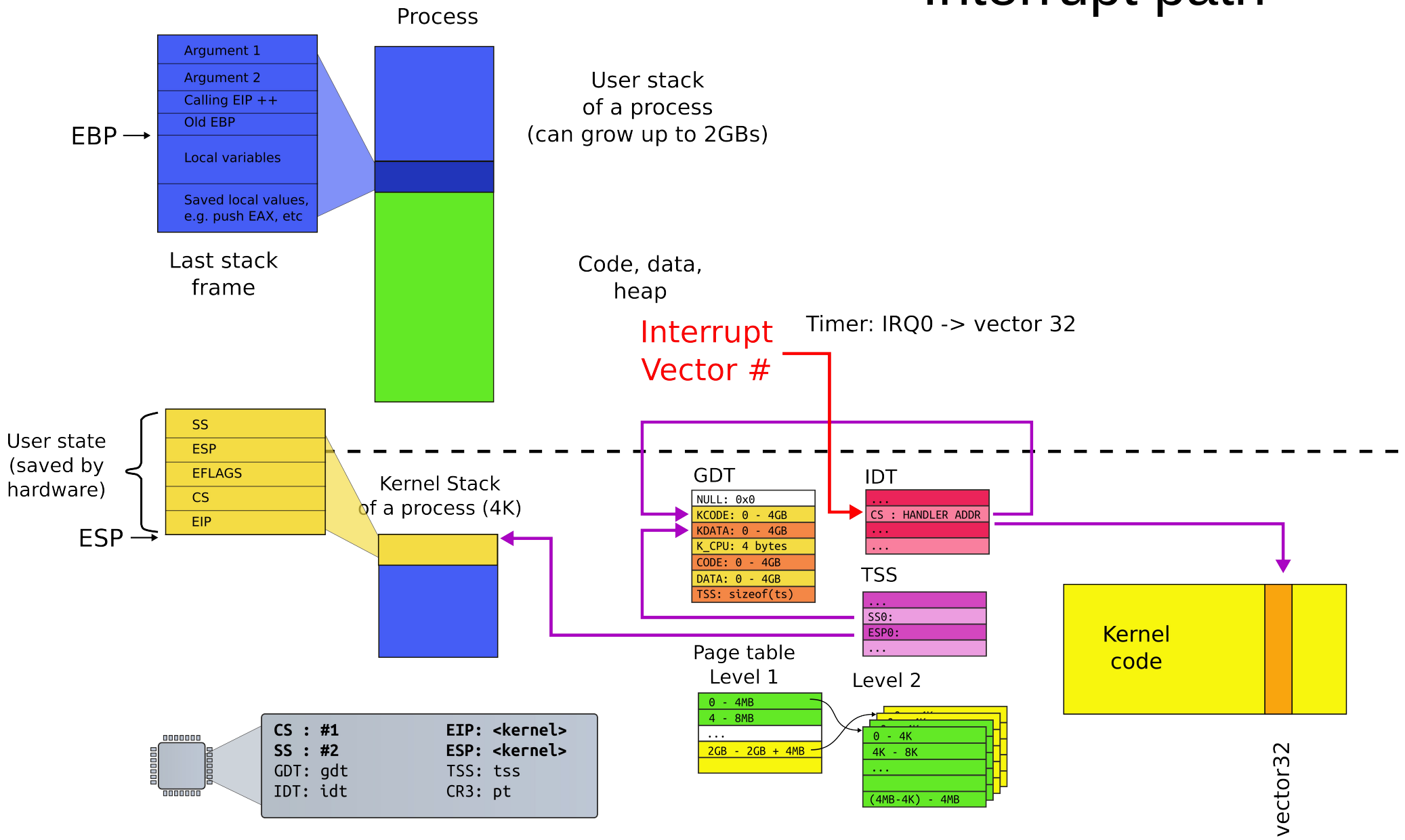
What's on the stack? Describe layout of a stack and how it changes during function invocation?

# 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)
```

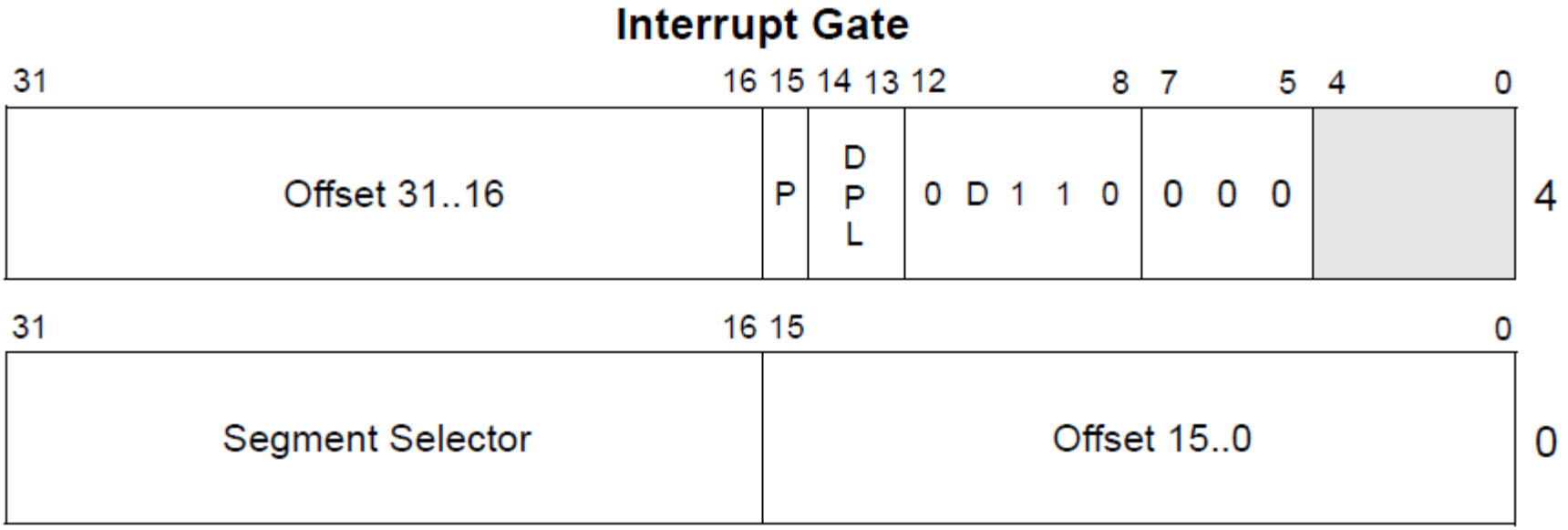
Describe the steps and data structures involved into a user to kernel transition (draw diagrams)

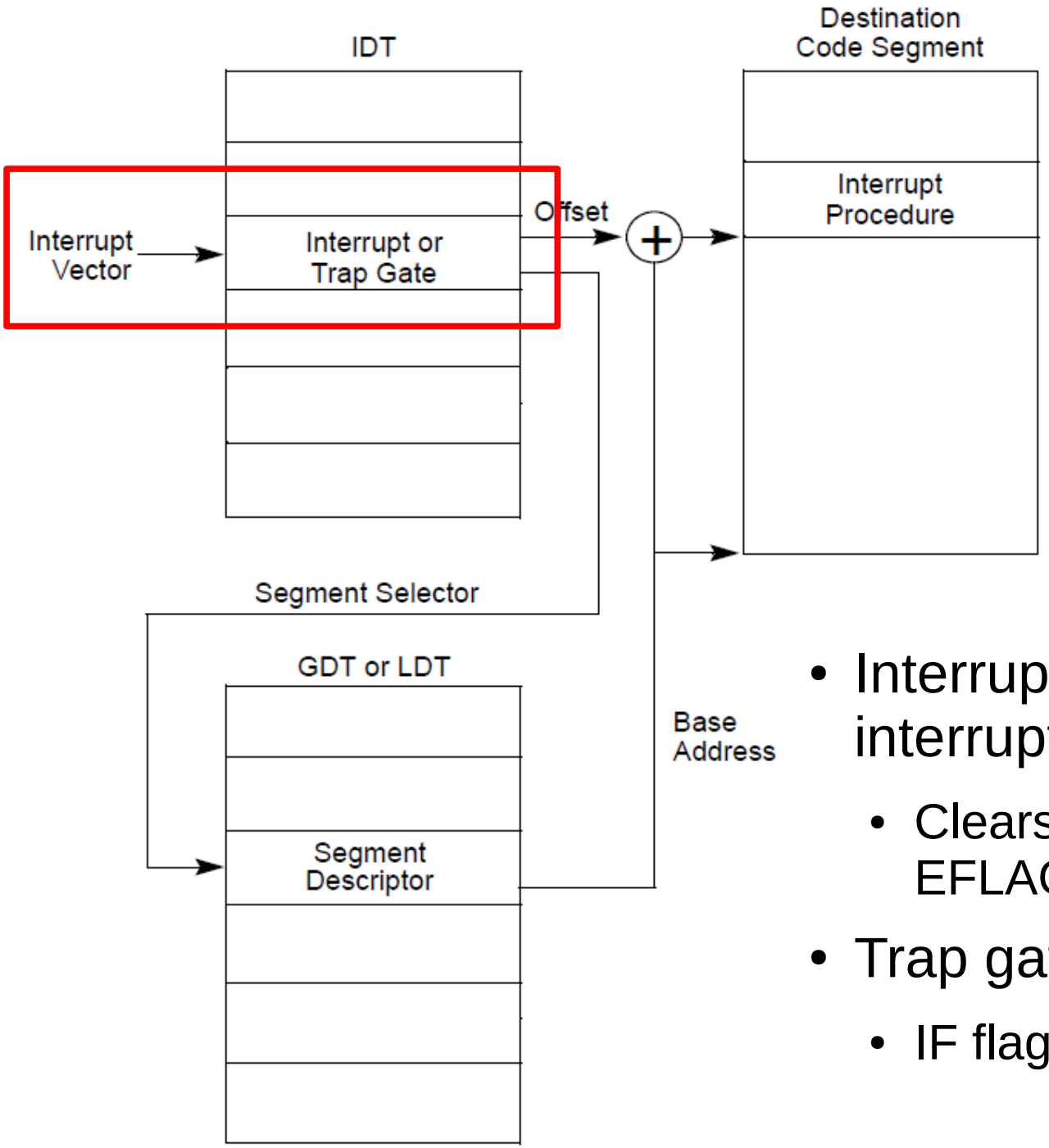
# Interrupt path



What segment is specified in the interrupt descriptor? Why?

# Interrupt descriptor

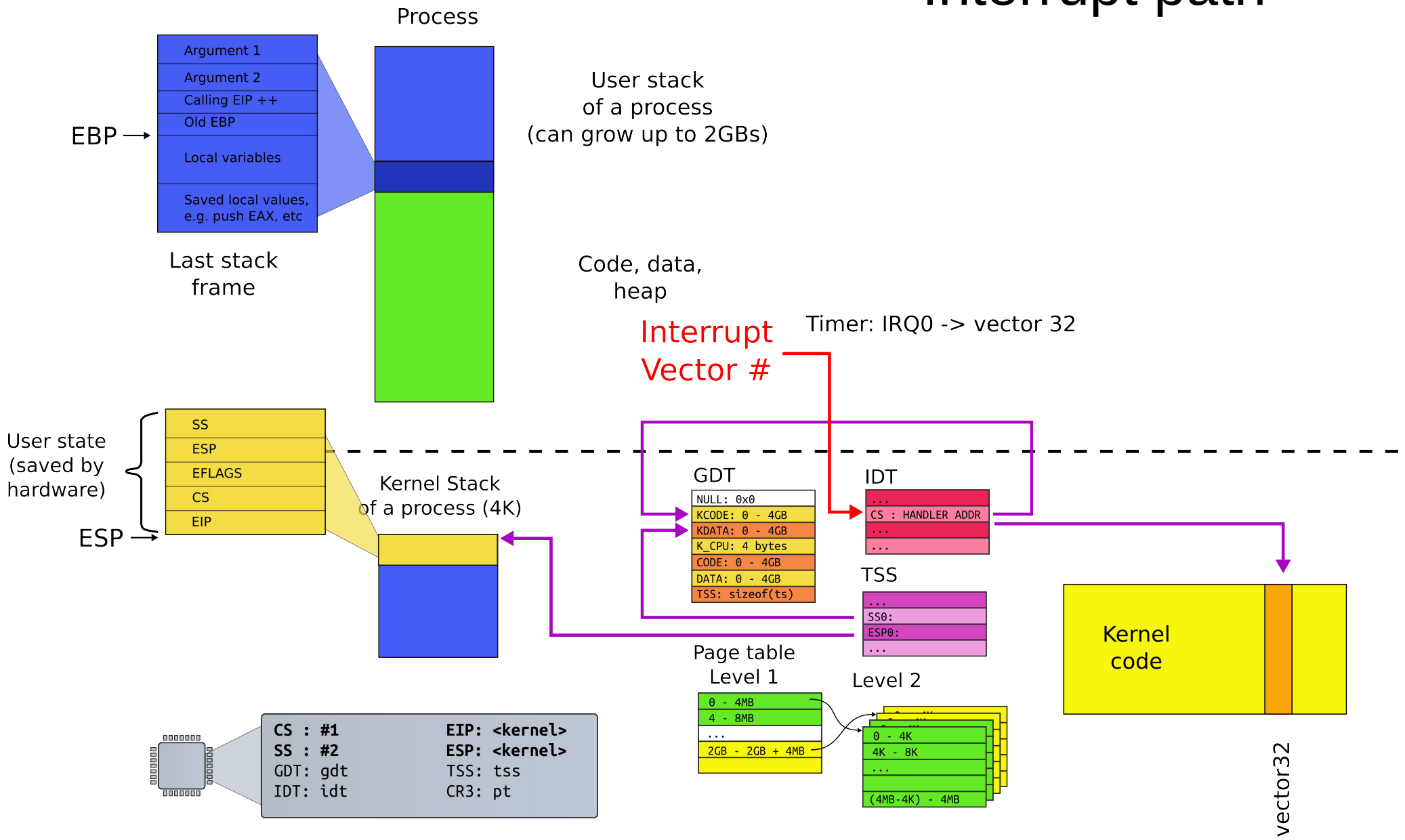





- Interrupt gate disables interrupts
  - Clears the IF flag in EFLAGS register
- Trap gate doesn't
  - IF flag is unchanged

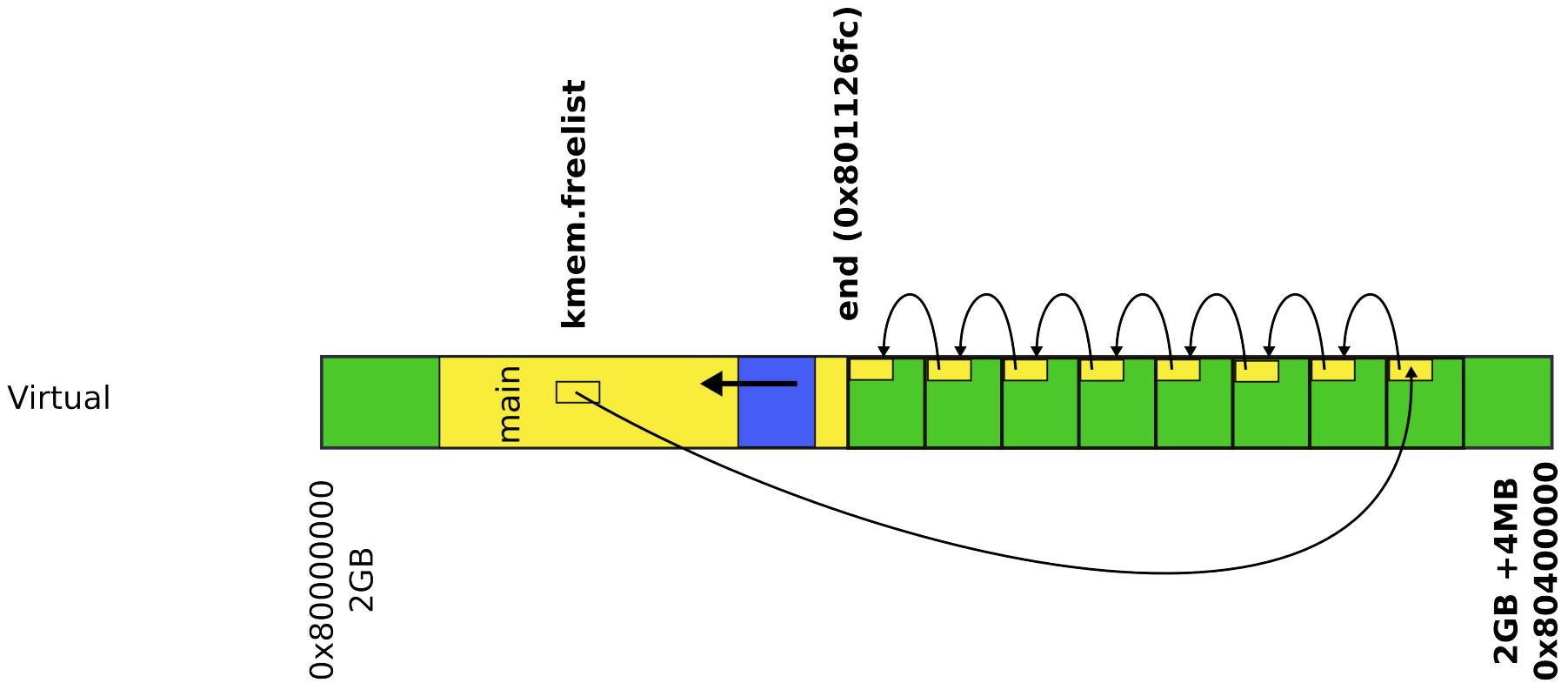
Which stack is used for execution of an interrupt handler? How does hardware find it?

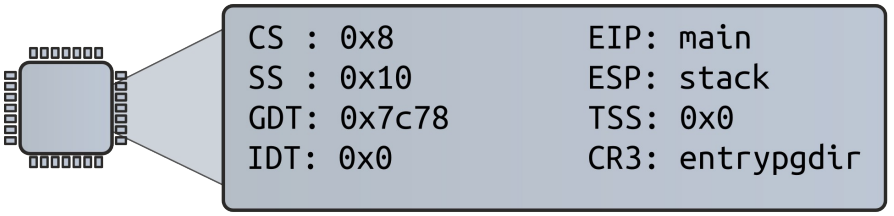
# Interrupt path



Describe organization of the memory allocator in xv6?

# Physical page allocator

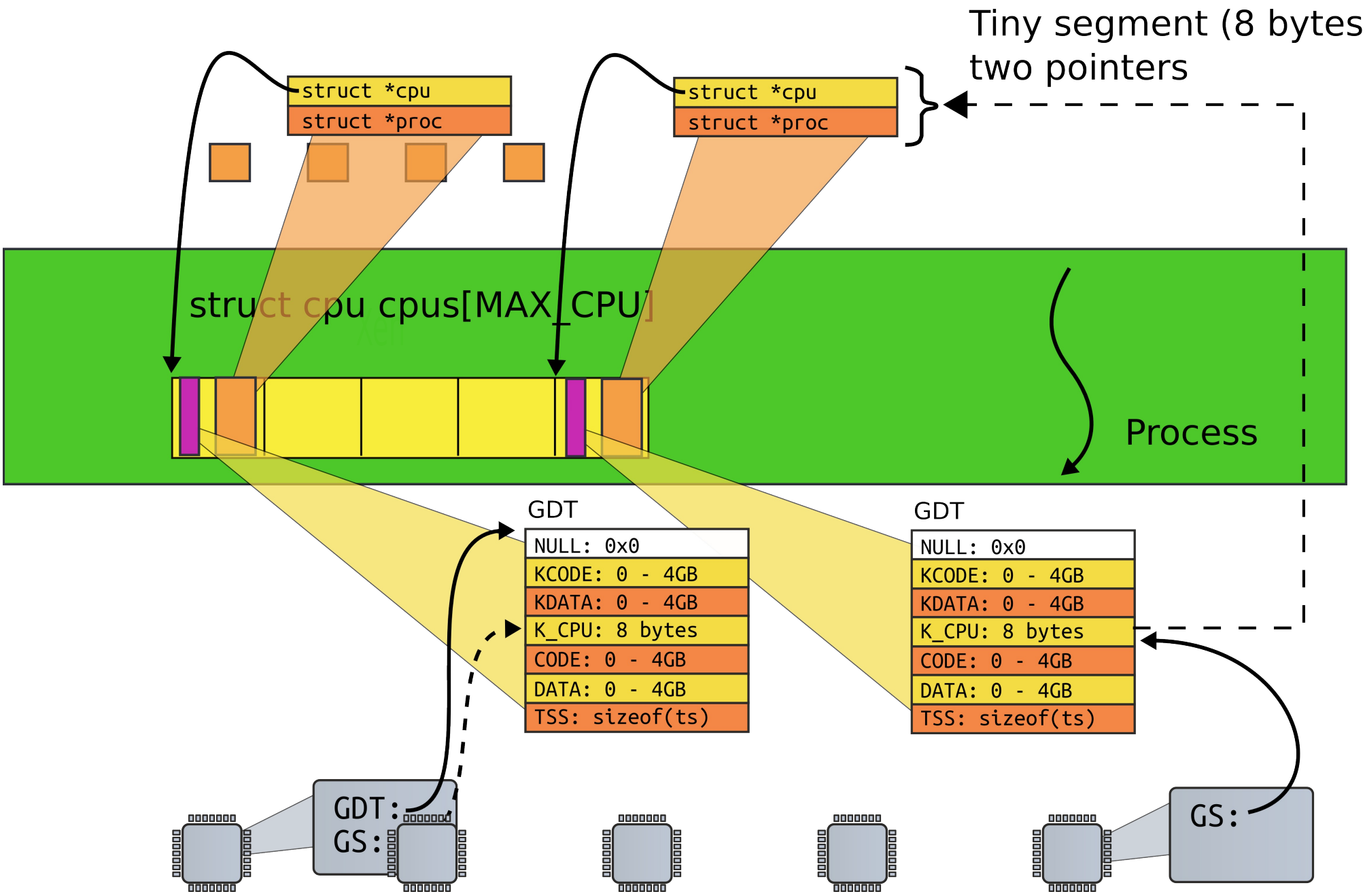



Protected Mode

# Where did free memory came from?

Describe how a per-CPU variables are stored in xv6?


Tiny segment (8 bytes),
two pointers
struct *cpu
struct *proc
struct *cpu
struct *proc
struct cpu cpus[MAX_CPU]
Process
GDT
NULL: 0x0
KCODE: 0 - 4GB
KDATA: 0 - 4GB
K_CPU: 8 bytes
CODE: 0 - 4GB
DATA: 0 - 4GB
TSS: sizeof(ts)
GDT
NULL: 0x0
KCODE: 0 - 4GB
KDATA: 0 - 4GB
K_CPU: 8 bytes
CODE: 0 - 4GB
DATA: 0 - 4GB
TSS: sizeof(ts)
GDT:
GS:
GS:

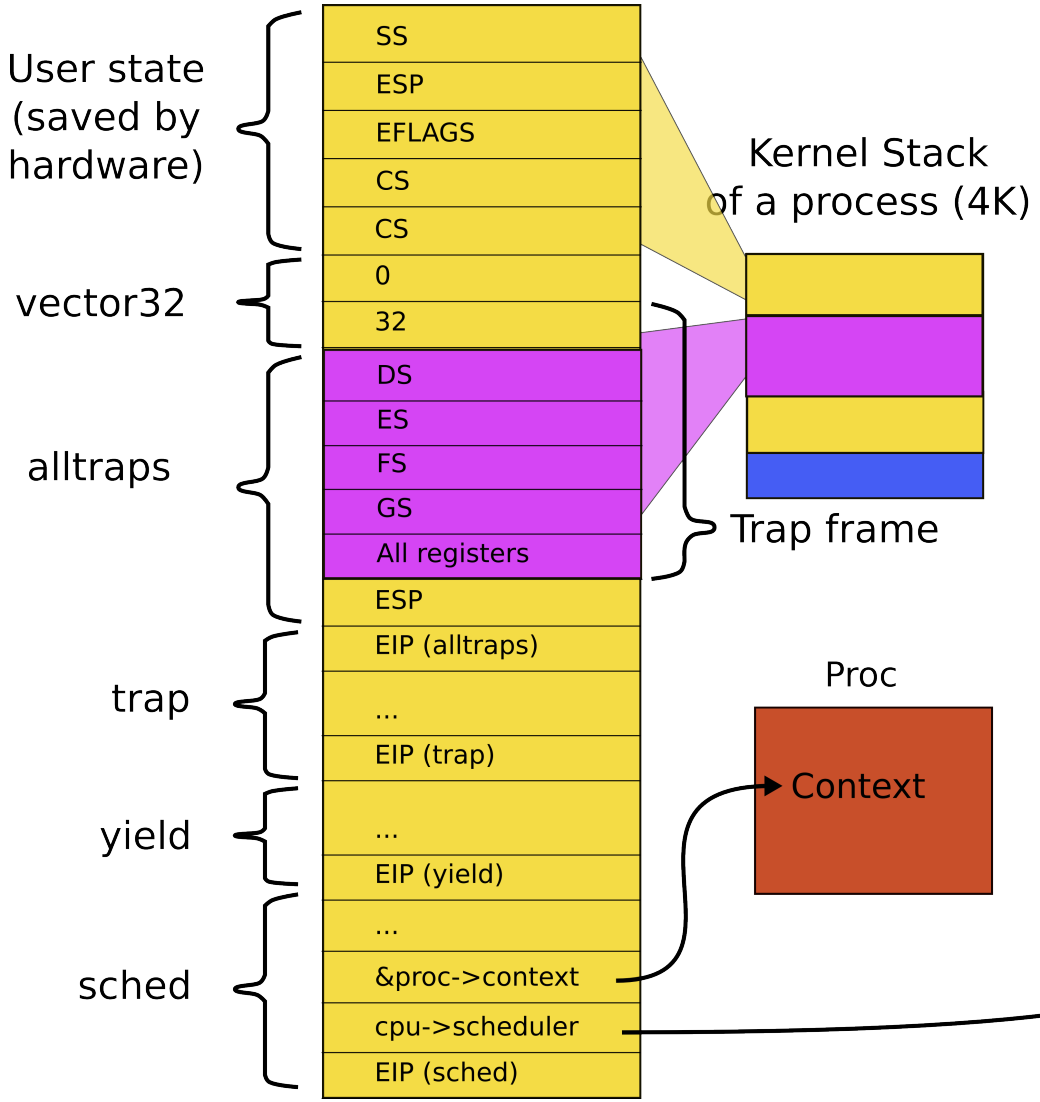
`swtch` in xv6 doesn’t explicitly save and restore all fields of struct context. Why is it okay that swtch doesn’t contain any code that saves `%eip`?

```
2958 swtch:
2959 movl 4(%esp), %eax
2960 movl 8(%esp), %edx
2961
2962 # Save old callee-save registers
2963 pushl %ebp
2964 pushl %ebx
2965 pushl %esi
2966 pushl %edi
2967
2968 # Switch stacksh
2969 movl %esp, (%eax)
2970 movl %edx, %esp
2971
2972 # Load new callee-save registers
2973 popl %edi
2974 popl %esi
2975 popl %ebx
2976 popl %ebp
2977 ret
```

# swtch()

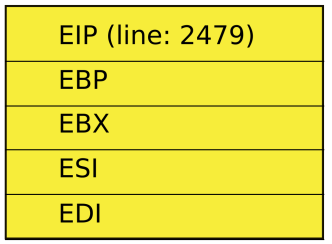
```
2093 struct context {
2094   uint edi;
2095   uint esi;
2096   uint ebx;
2097   uint ebp;
2098   uint eip;
2099 };
```

# Stack inside swtch()




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

Context



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.

```
3374 void
3375 syscall(void)
3376 {
3377   int num;
3378
3379   num = proc->tf->eax;
3380   if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
3381     proc->tf->eax = syscalls[num]();
3382   } else {
3383     cprintf("%d %s: unknown sys call %d\n",
3384     proc->pid, proc->name, num);
3385     proc->tf->eax = -1;
3386   }
3387 }
```

```
3374 void
3375 syscall(void)
3376 {
3377   int num;
3378
3379   num = proc->tf->eax;
3380   if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
3381     // proc->tf->eax = syscalls[num]();
       proc->tf->esp -= 4;
       *(int*)ptoc->tf->esp = syscalls[num]();
3382   } else {
3383     cprintf("%d %s: unknown sys call %d\n",
3384             proc->pid, proc->name, num);
3385     // proc->tf->eax = -1;
       proc->tf->esp -= 4;
       *(int*)ptoc->tf->esp = -1;
3386   }
3387 }
```

Thank you!