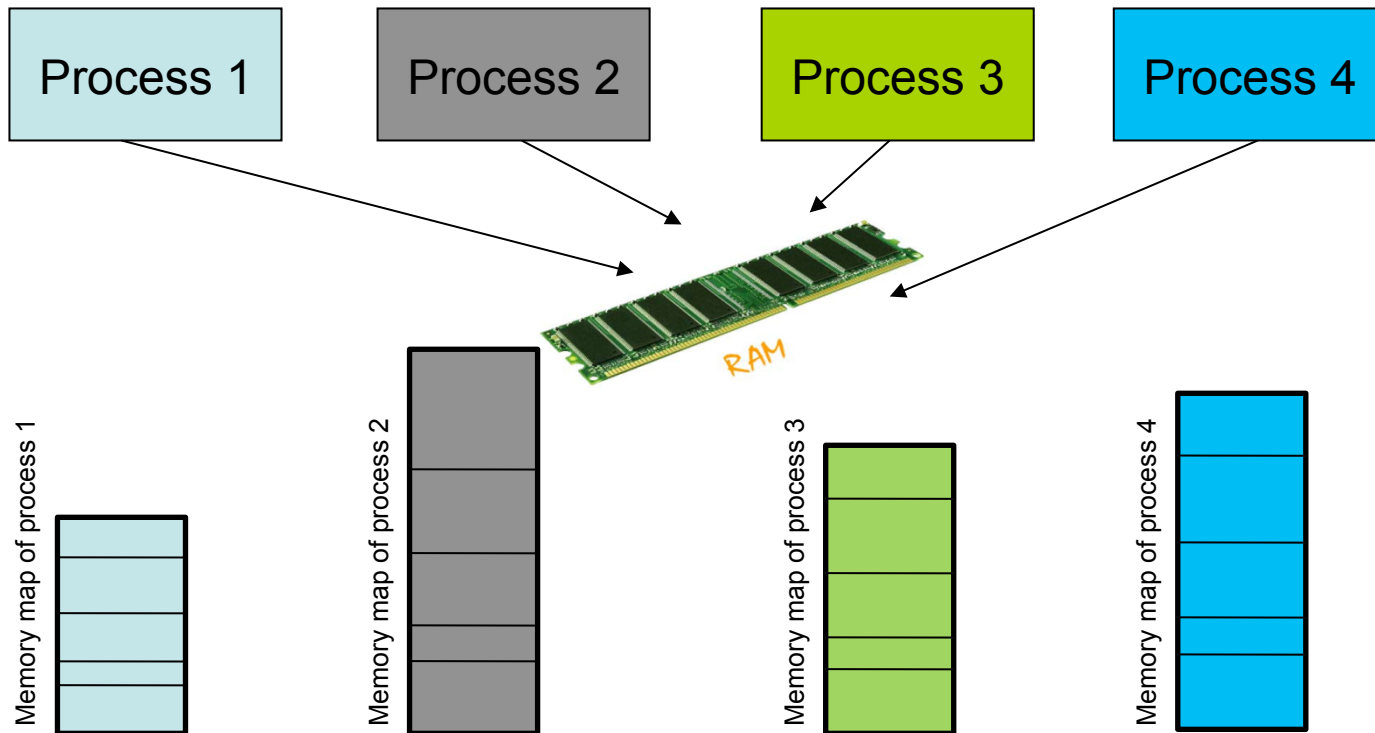


Memory Management

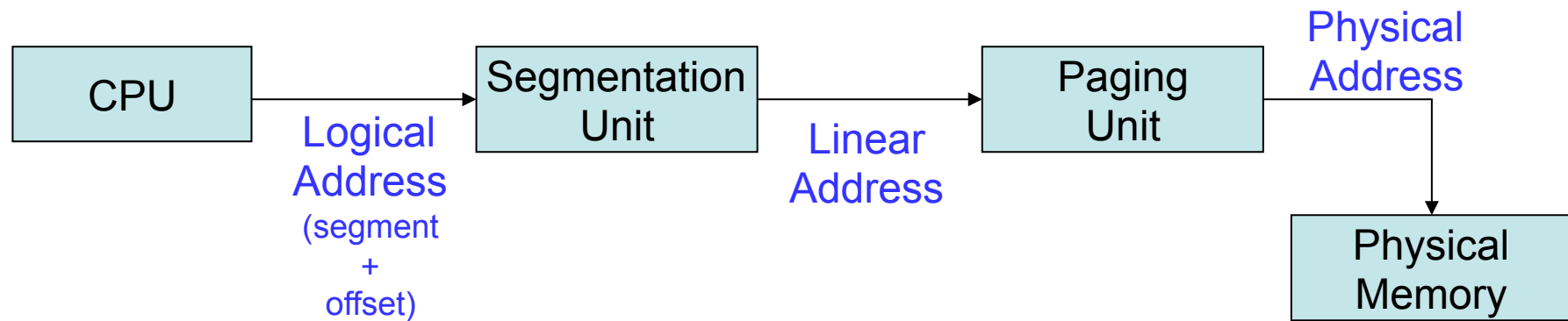
Chester Rebeiro
IIT Madras

CR

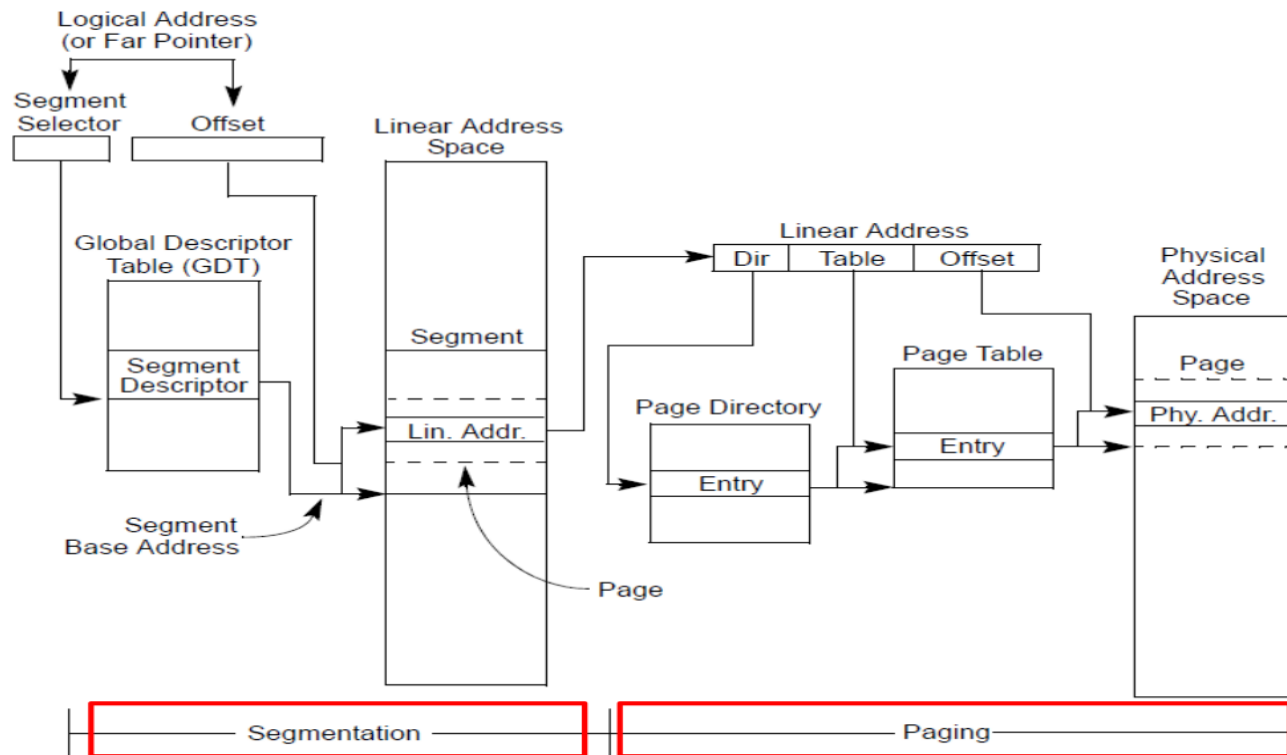
Sharing RAM



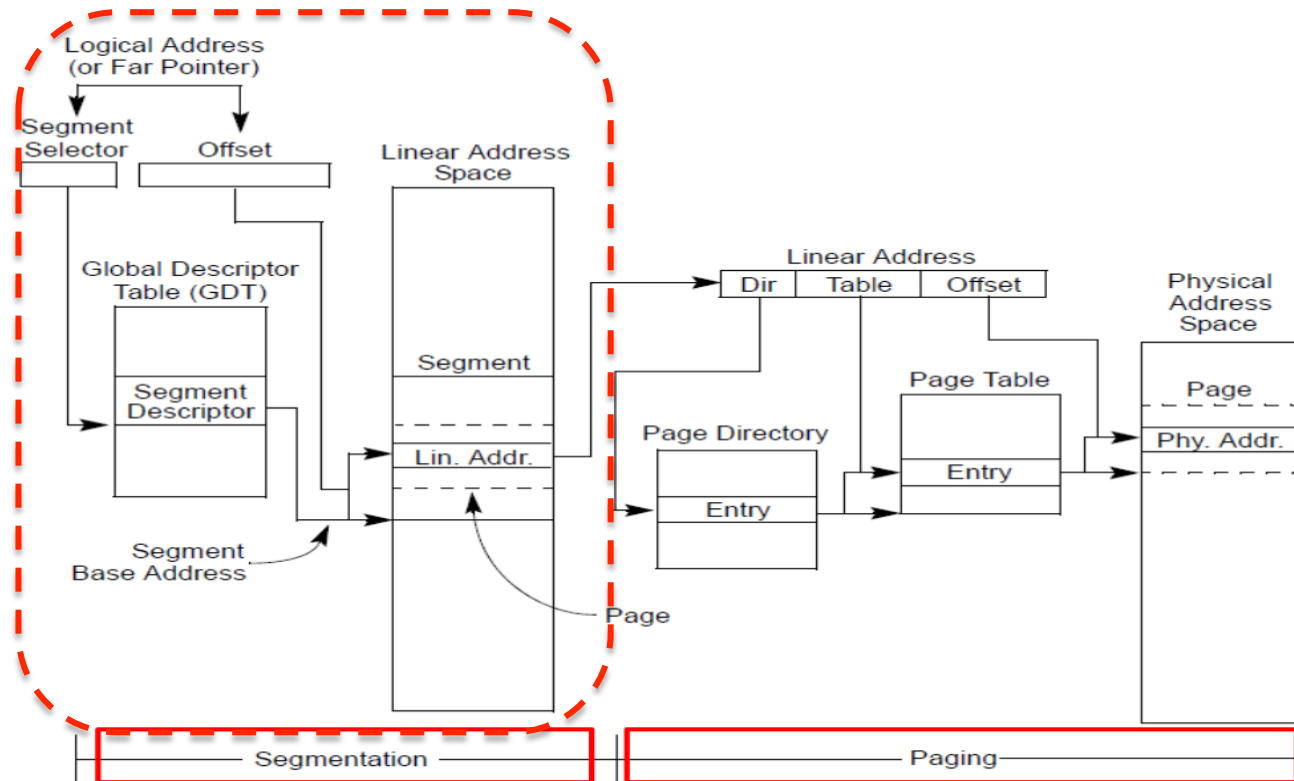
x86 address translation



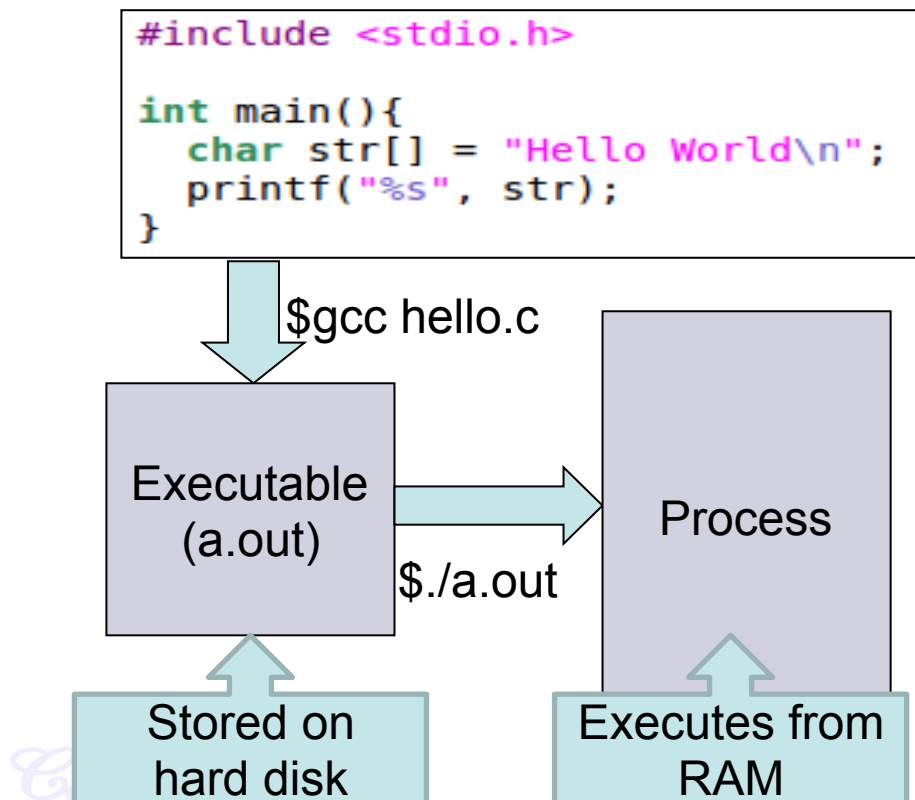
x86 Memory Management



Segmentation



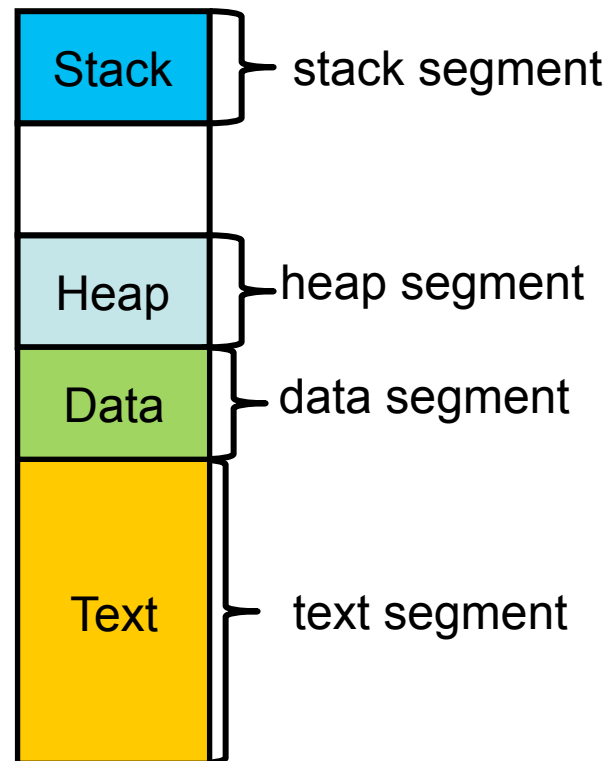
Executing Programs (Process)



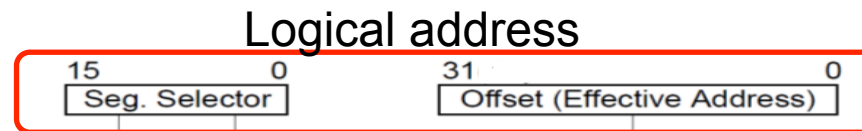
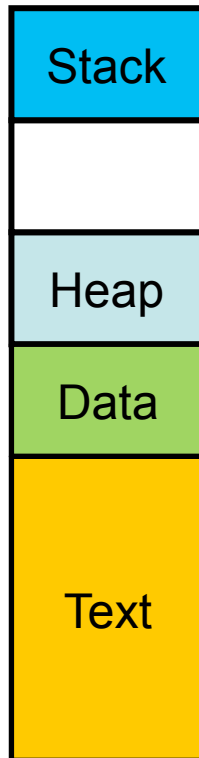
- Process

- A program in execution
- Present in the RAM
- Comprises of
 - Executable instructions
 - Stack
 - Heap
 - State in the OS (in kernel)
- State contains : registers, list of open files, related processes, etc.

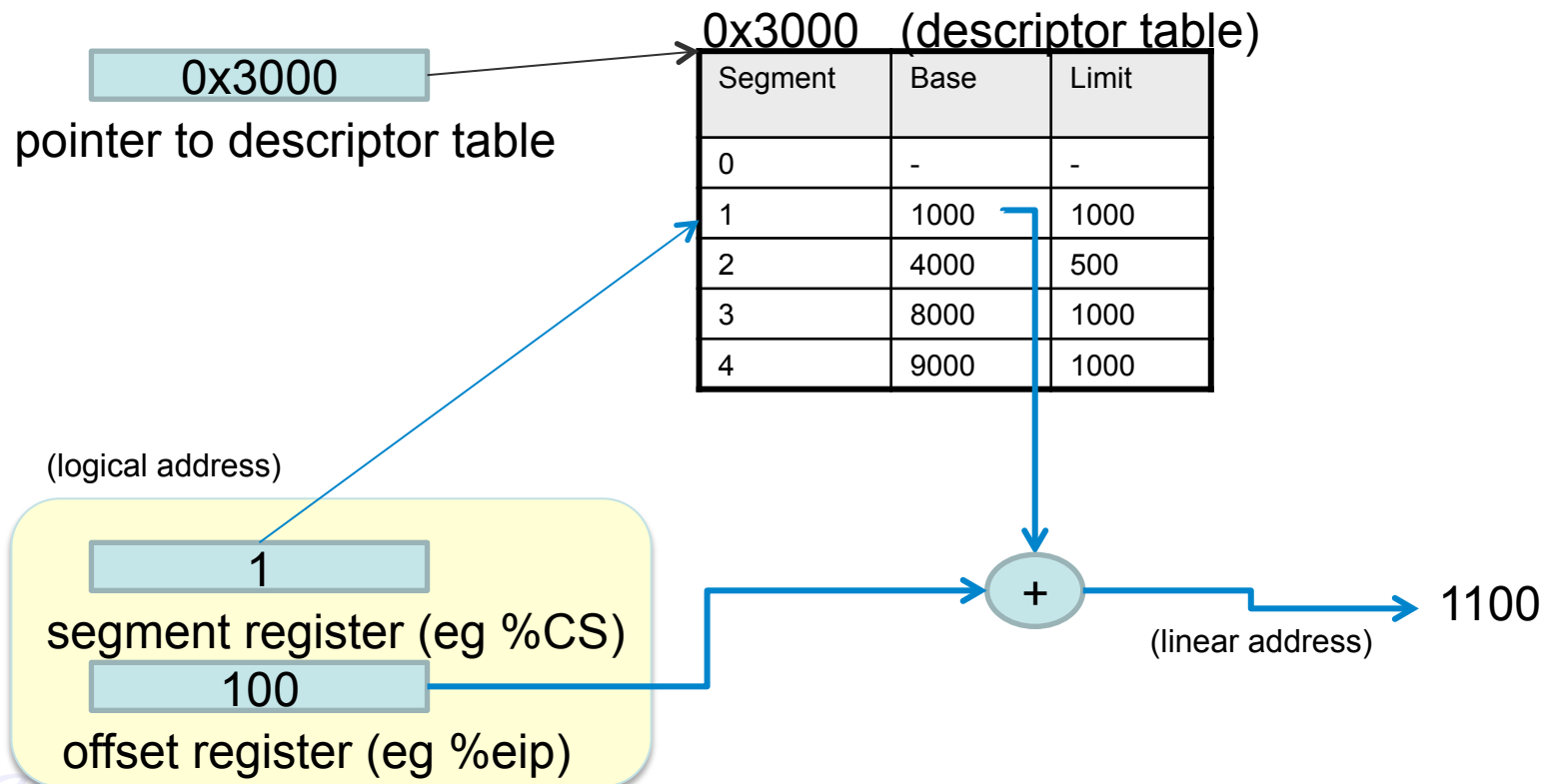
Segments (an example)



Segmentation (*logical to linear* address)

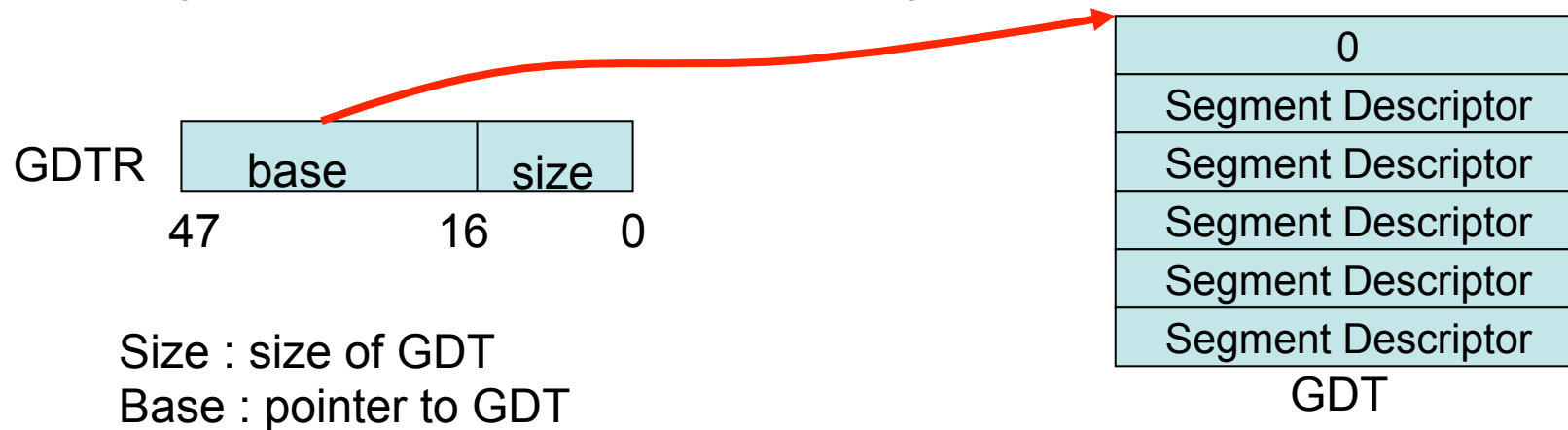


Example

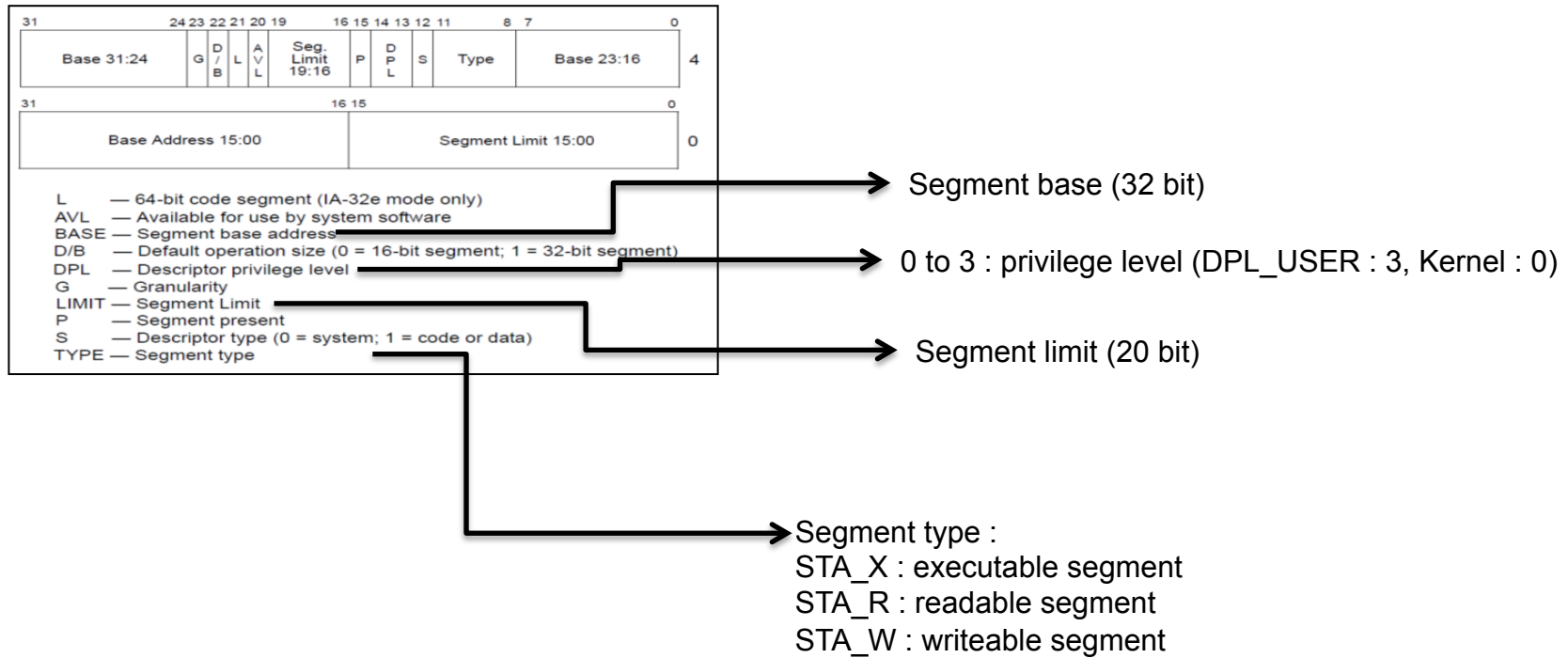


Pointer to Descriptor Table

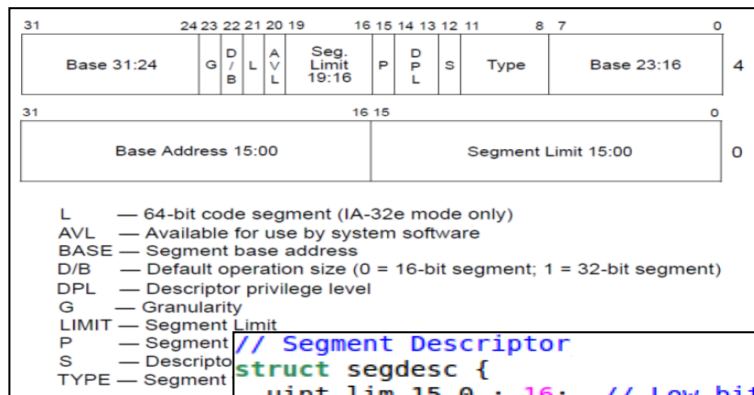
- Global Descriptor Table (GDT)
 - Stored in memory
- Pointed to by GDTR (GDT Register)
 - lgdt (instruction used to load the GDT register)



Segment Descriptor



Segment Descriptor in xv6



```

// Normal segment
#define SEG(type, base, lim, dpl) (struct segdesc) \
{ ((lim) >> 12) & 0xffff, (uint)(base) & 0xffff, \
  ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1, \
  (uint)(lim) >> 28, 0, 0, 1, 1, (uint)(base) >> 24 }
  
```

SEG(STA_W, 0, 0xFFFFFFFF, DPL_USER)

```

// Segment Descriptor
struct segdesc {
  uint lim_15_0 : 16; // Low bits of segment limit
  uint base_15_0 : 16; // Low bits of segment base address
  uint base_23_16 : 8; // Middle bits of segment base address
  uint type : 4; // Segment type (see STS_constants)
  uint s : 1; // 0 = system, 1 = application
  uint dpl : 2; // Descriptor Privilege Level
  uint p : 1; // Present
  uint lim_19_16 : 4; // High bits of segment limit
  uint avl : 1; // Unused (available for software use)
  uint rsv1 : 1; // Reserved
  uint db : 1; // 0 = 16-bit segment, 1 = 32-bit segment
  uint g : 1; // Granularity: limit scaled by 4K when set
  uint base_31_24 : 8; // High bits of segment base address
};
  
```



ref : mmu.h ([7], 0752, 0769)

Segments in xv6

Segment	Base	Limit	Type	DPL
Kernel Code	0	4 GB	X, R	0
Kernel Data	0	4 GB	W	0
User Code	0	4 GB	X, R	3
User Data	0	4 GB	W	3

Loading the GDT

2308

```
struct segdesc gdt[NSEGS];
```

1724

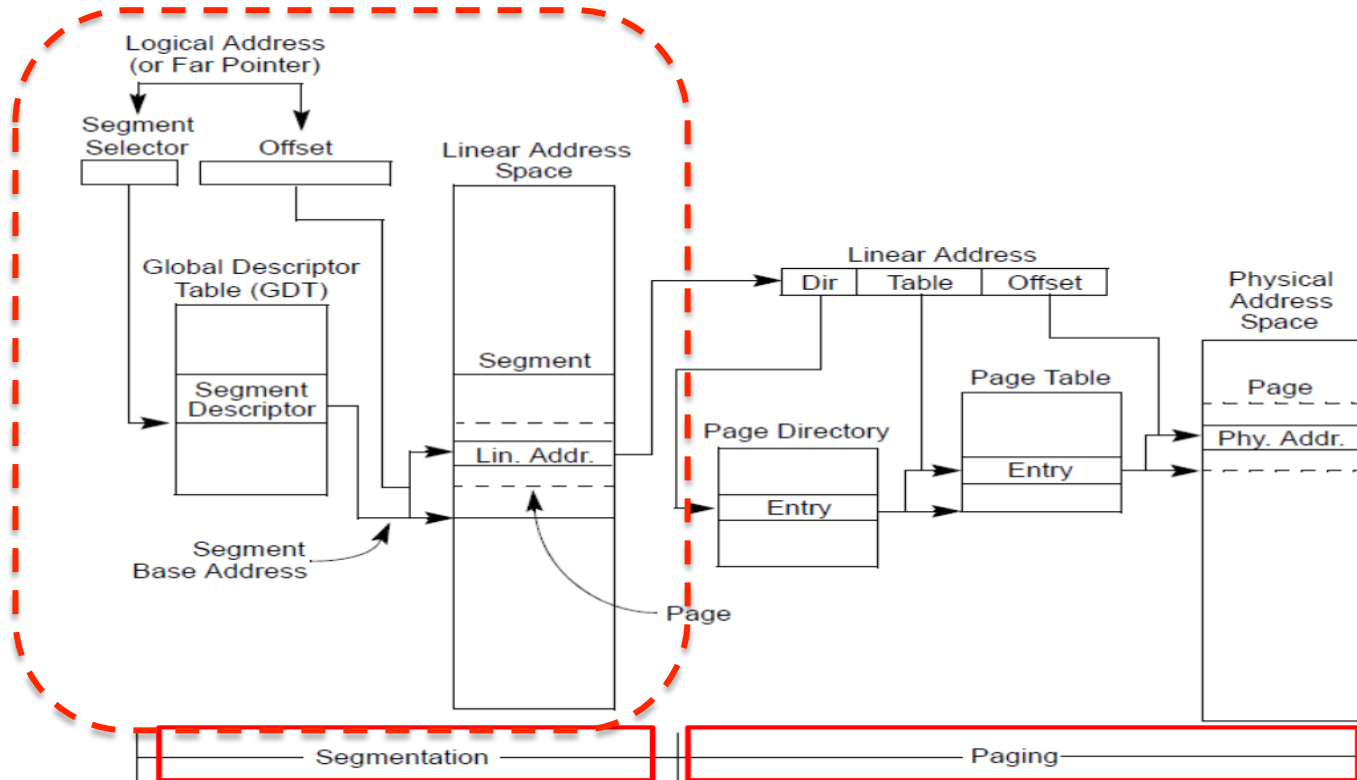
```
c = &cpus[cpunum()];  
c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);  
c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);  
c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);  
c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
```

```
lgdt(c->gdt, sizeof(c->gdt));
```

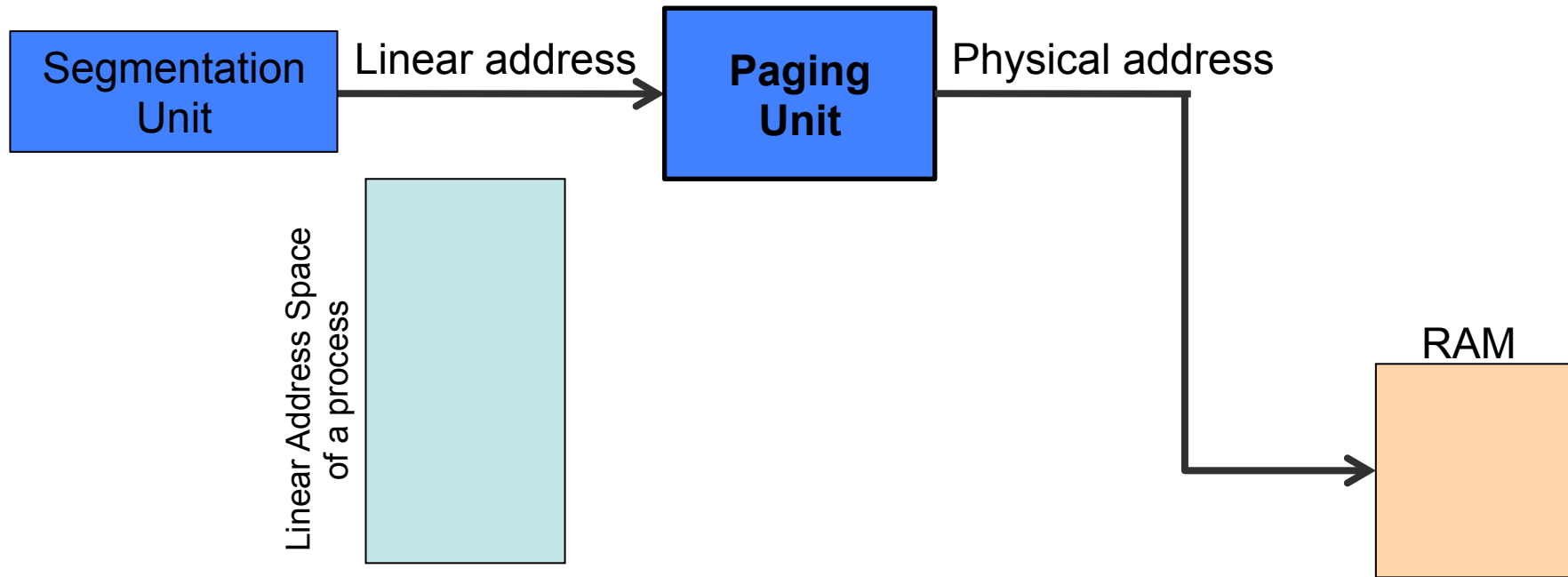
0512

```
static inline void  
lgdt(struct segdesc *p, int size)  
{  
    volatile ushort pd[3];  
  
    pd[0] = size-1;  
    pd[1] = (uint)p;  
    pd[2] = (uint)p >> 16;  
  
    asm volatile("lgdt (%0)" : : "r" (pd));  
}
```

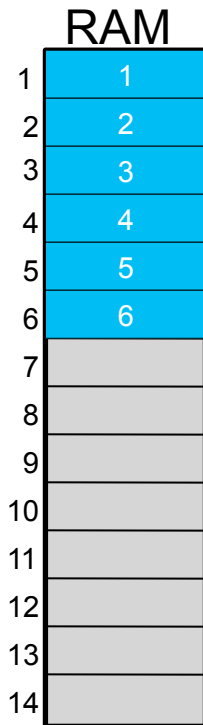
Virtual Memory



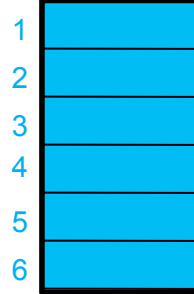
Paging Unit



Virtual Memory



Linear address space of process1



process page table

block	page frame
1	1
2	2
3	3
4	4
5	5
6	6

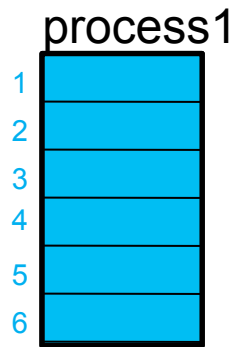
Because of the page table, blocks need not be in contiguous page frames

Every time a memory location is accessed, the processor looks into the page table to identify the corresponding page frame number.

Virtual Memory

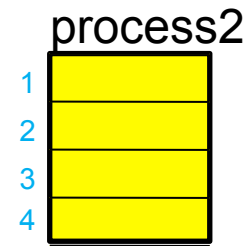


Blocks from
Several processes
can share pages in
RAM
simultaneously



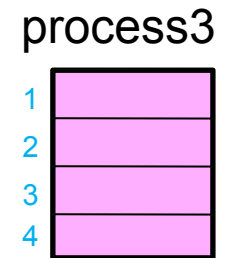
process page table

block	page frame
1	14
2	2
3	13
4	4
5	1
6	8



process page table

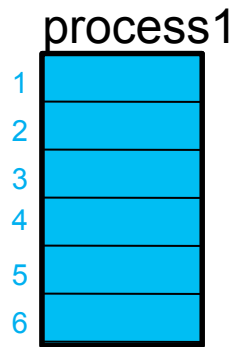
block	page frame
1	10
2	7
3	12
4	9



process page table

block	page frame
1	11
2	6
3	3
4	5

Virtual Memory



process page table

block	page frame
1	14
2	2
3	13
4	4
5	1
6	8

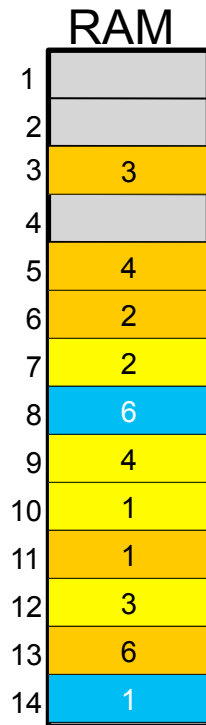
Do we really need to load all blocks into memory before the process starts executing?

No.

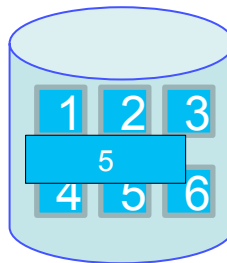
Not all parts of the program are accessed simultaneously. Infact, some code may not even be executed.

Virtual memory takes advantage of this by using a concept called demand paging.

Demand Paging



Swap space
(on disk)



process page table in RAM

block	page frame	P ← present bit
1	14	1
2		0
3		0
4		0
5	1	0
6	8	1

Pages are loaded from disk to RAM, only when needed.

A 'present bit' in the page table indicates if the block is in RAM or not.

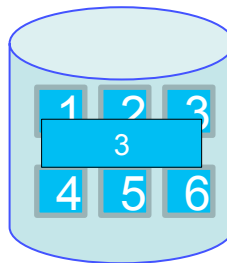
If (present bit = 1) { block in RAM }
else { block not in RAM }

If a page is accessed that is not present in RAM, the processor issues a page fault interrupt, triggering the OS to load the page into RAM and mark the present bit to 1

Demand Paging



Swap space
(on disk)



process page table in RAM

block	page frame	P
1	14	0
2	2	1
3	14	1
4	4	1
5	1	1
6	8	1

If there are no pages free for a new block to be loaded, the OS makes a decision to remove another block from RAM.

This is based on a replacement policy, implemented in the OS.

Some replacement policies are

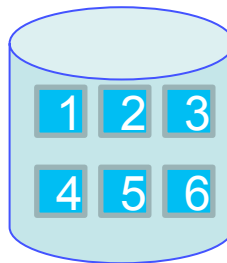
- * First in first out
- * Least recently used
- * Least frequently used

The replaced block **may** need to be written back to the swap (swap out)

Demand Paging



Swap space
(on disk)



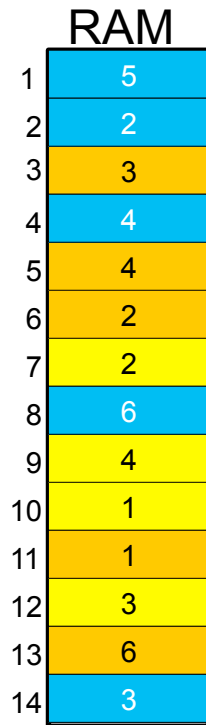
process page table in RAM

block	page frame	P	D
1	14	0	1
2	2	1	1
3	14	1	0
4	4	1	1
5	1	1	0
6	8	1	1

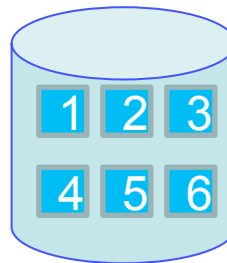
The **dirty bit**, in the page table indicates if a page needs to be written back to disk

If the dirty bit is 1, indicates the page needs to be written back to disk.

Demand Paging



Swap space
(on disk)



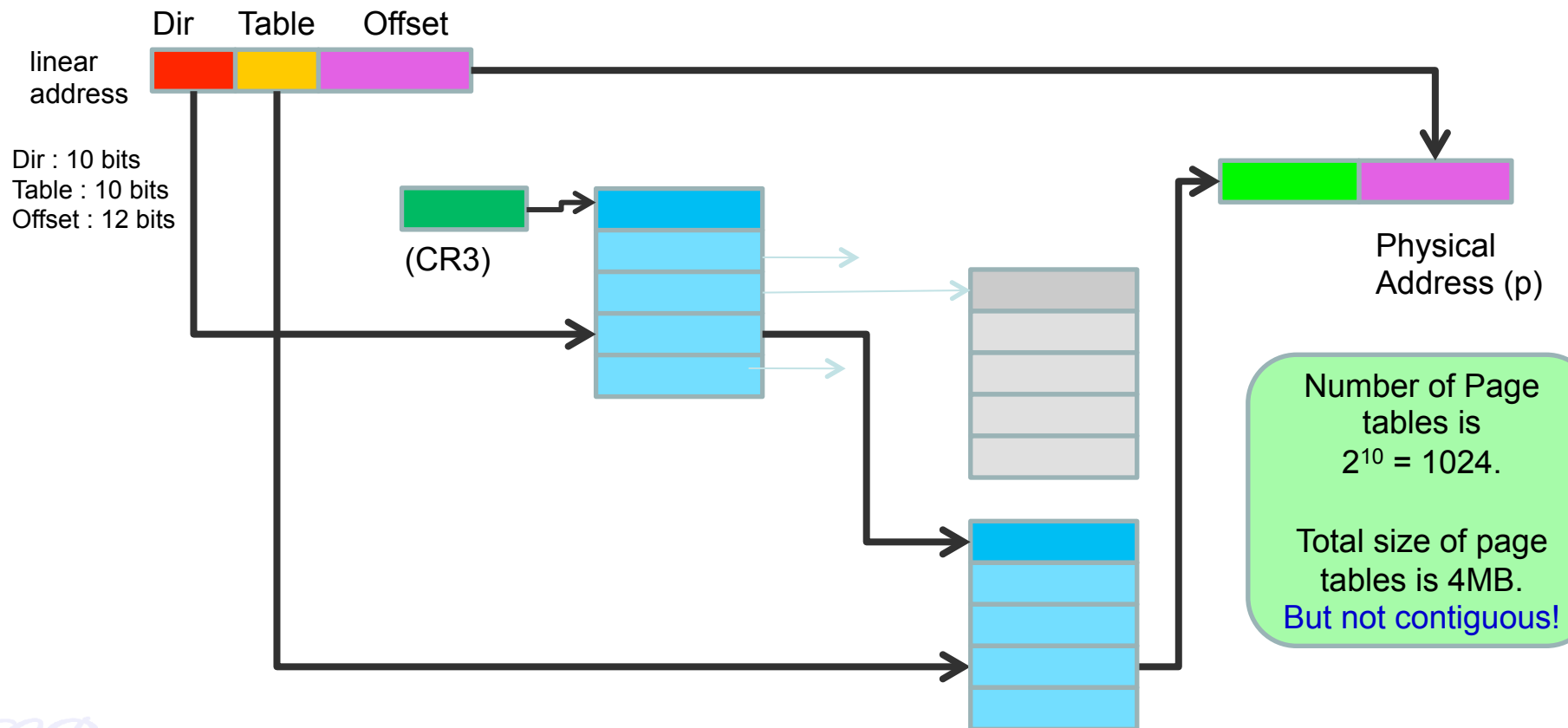
Protection bits, in the page table determine if the page is executable, readonly, and accessible by a user process.

process page table in RAM

block	page frame	P	D	
1	14	0	1	11
2	2	1	1	10
3	14	1	0	00
4	4	1	1	11
5	1	1	0	01
6	8	1	1	10

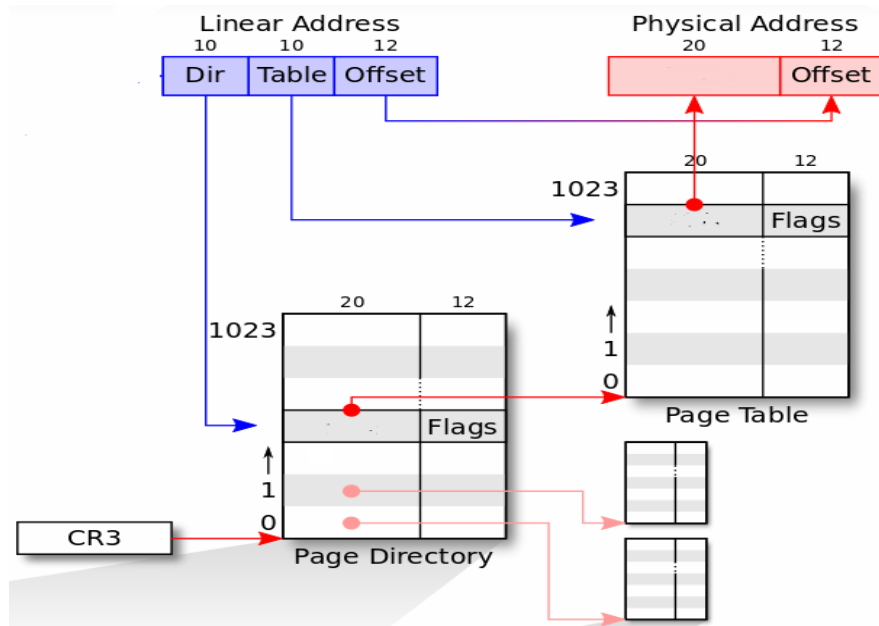
← protection bits

2 Level Page Translation



Linear to Physical Address

- 2 level page translation



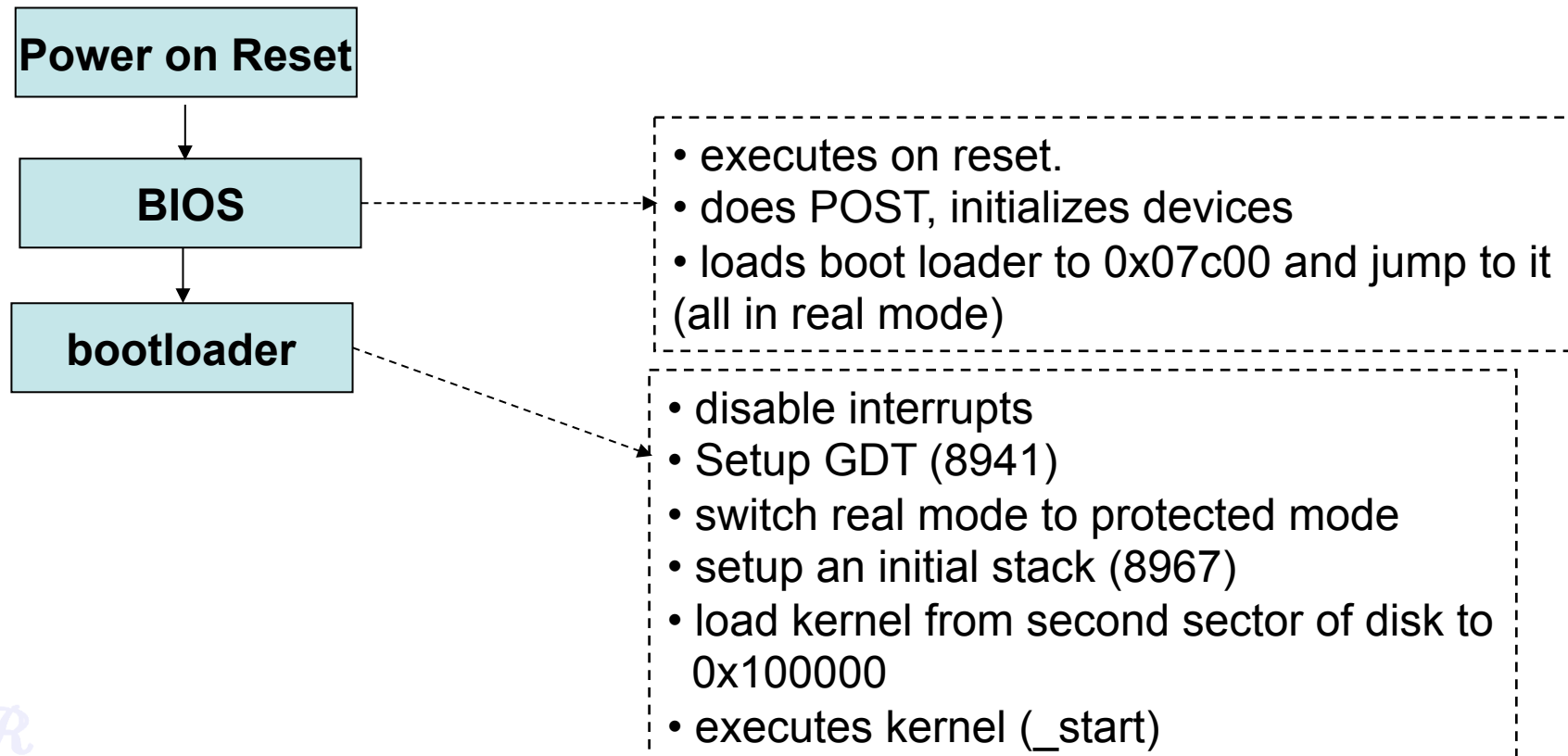
- How many page tables are present?
- What is the maximum size of the process' address space?

CR

ref : mmu.h (PGADDR, NPENTRIES, NPTENTRIES, PGSIZE)

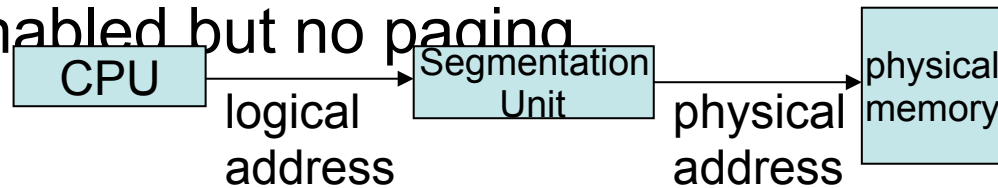
back to booting...

so far...

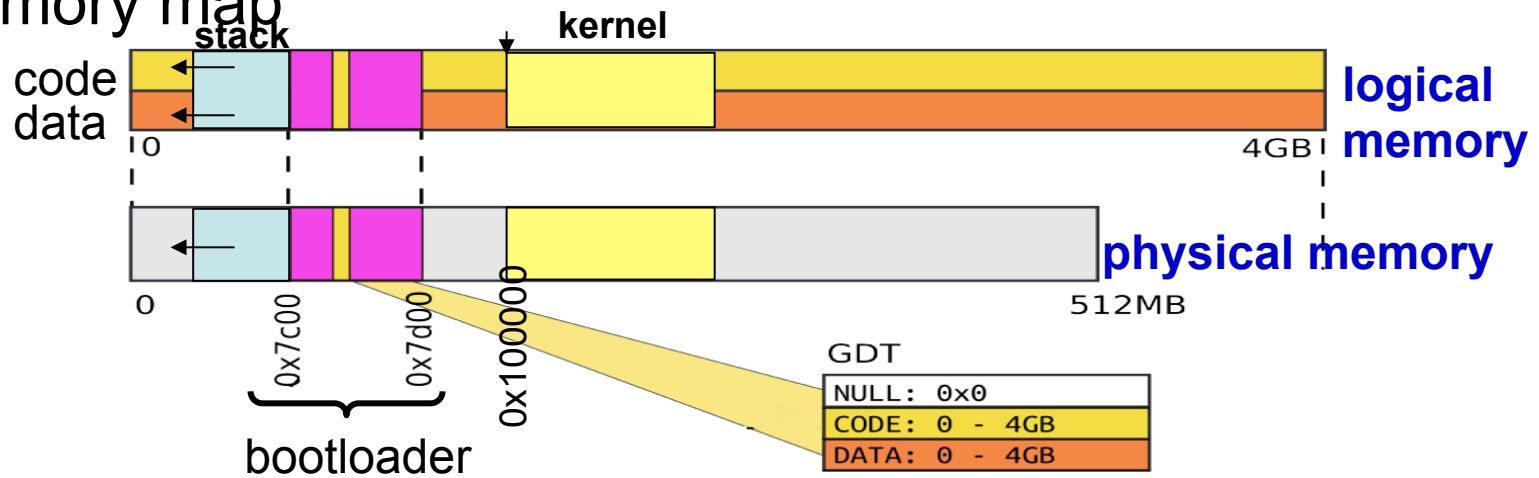


Memory when kernel is invoked (just after the bootloader)

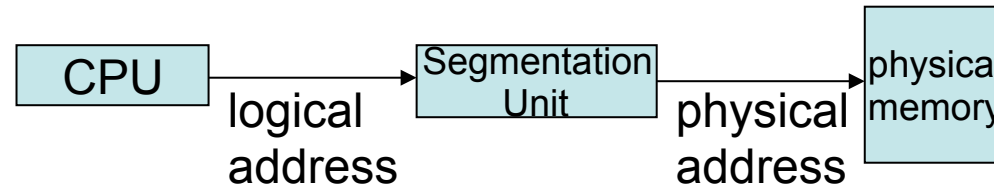
- Segmentation enabled but no padding



- Memory map



Memory Management Analysis



- Advantages
 - Got the kernel into protected mode (32 bit code) with minimum trouble
- Disadvantages
 - Protection of kernel memory from user writes
 - Protection between user processes
 - User space restricted by physical memory
- The plan ahead
 - Need to get paging up and running

entry

```
1035 .globl _start
1036 _start = V2P_WO(entry)
1037
1038 # Entering xv6 on boot processor, with paging off.
1039 .globl entry
1040 entry:
1041 # Turn on page size extension for 4Mbyte pages
1042 movl    %cr4, %eax
1043 orl     $(CR4_PSE), %eax
1044 movl    %eax, %cr4
1045 # Set page directory
1046 movl    $(V2P_WO(entrypgdir)), %eax
1047 movl    %eax, %cr3
1048 # Turn on paging.
1049 movl    %cr0, %eax
```

The kernel executes
from here

OS code Linker address

- kernel.asm (xv6)
- The linker sets the executable so that the kernel starts from 0x80100000
- 0x80100000 is a virtual address and not a physical address

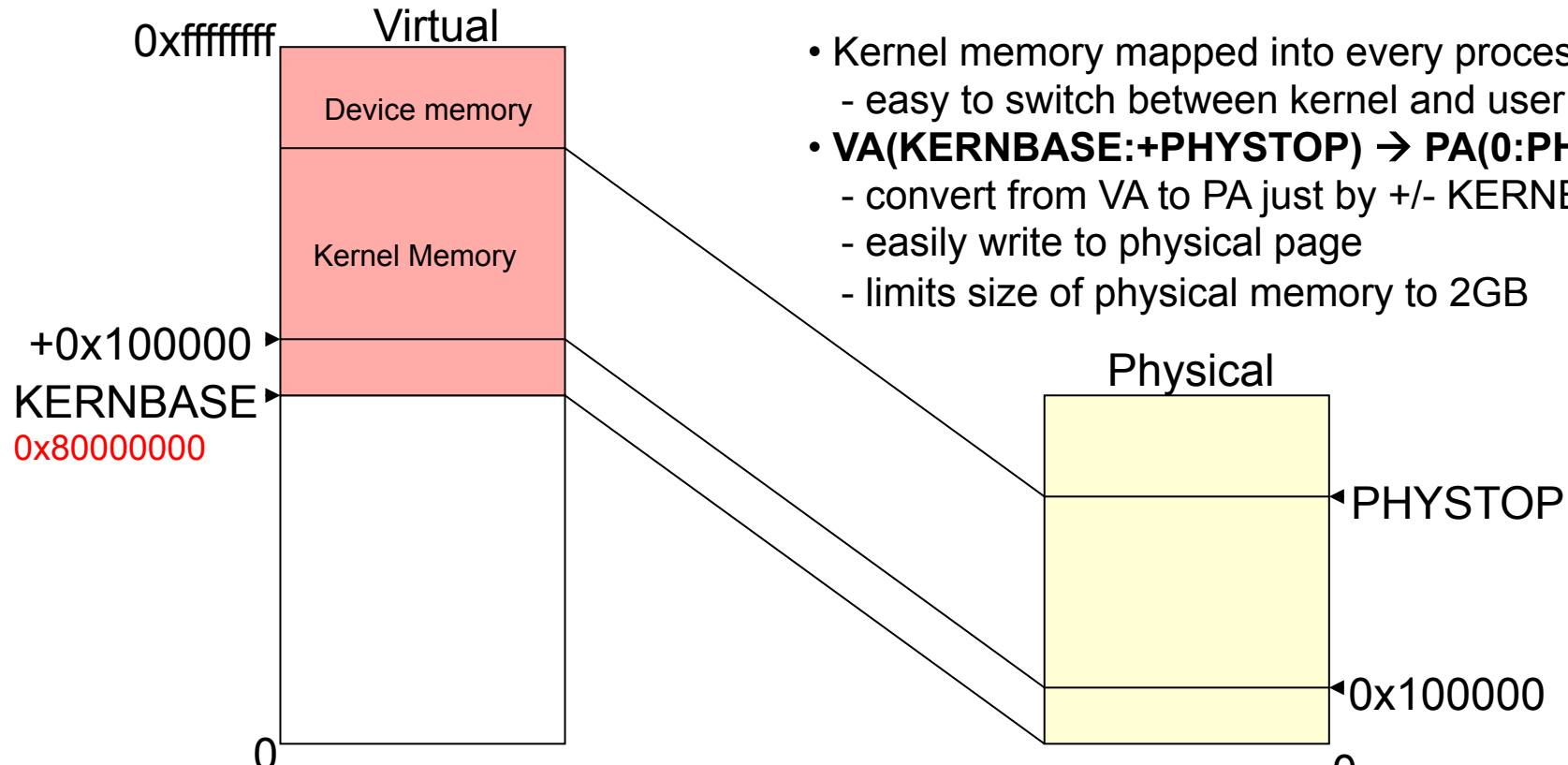
```
Disassembly of section .text:
80100000 <multiboot_header>:
80100000:    02 b0 ad 1b 00 00    add    0x1bad(%eax),%dh
80100006:    00 00                add    %al,(%eax)
80100008:    fe 4f 52            decb   0x52(%edi)
8010000b:    e4 0f                in     $0xf,%al

8010000c <entry>:

# Entering xv6 on boot processor, with paging off.
.globl entry
entry:
# Turn on page size extension for 4Mbyte pages
movl   %cr4, %eax
8010000c:    0f 20 e0            mov    %cr4,%eax
orl    $(CR4_PSE), %eax
8010000f:    83 c8 10            or     $0x10,%eax
movl   %eax, %cr4
80100012:    0f 22 e0            mov    %eax,%cr4
# Set page directory
movl   $(V2P_W0(entrypgdir)), %eax
80100015:    b8 00 a0 10 00     mov    $0x10a000,%eax
movl   %eax, %cr3
8010001a:    0f 22 d8            mov    %eax,%cr3
# Turn on paging.
movl   %cr0, %eax
8010001d:    0f 20 c0            mov    %cr0,%eax
orl    $(CR0_PG|CR0_WP), %eax
80100020:    0d 00 00 01 80     or     $0x80010000,%eax
movl   %eax, %cr0
80100025:    0f 22 c0            mov    %eax,%cr0

# Set up the stack pointer.
movl   $(stack + KSTACKSIZE), %esp
80100028:    bc 50 c6 10 80     mov    $0x8010c650,%esp
```

Virtual Address Space

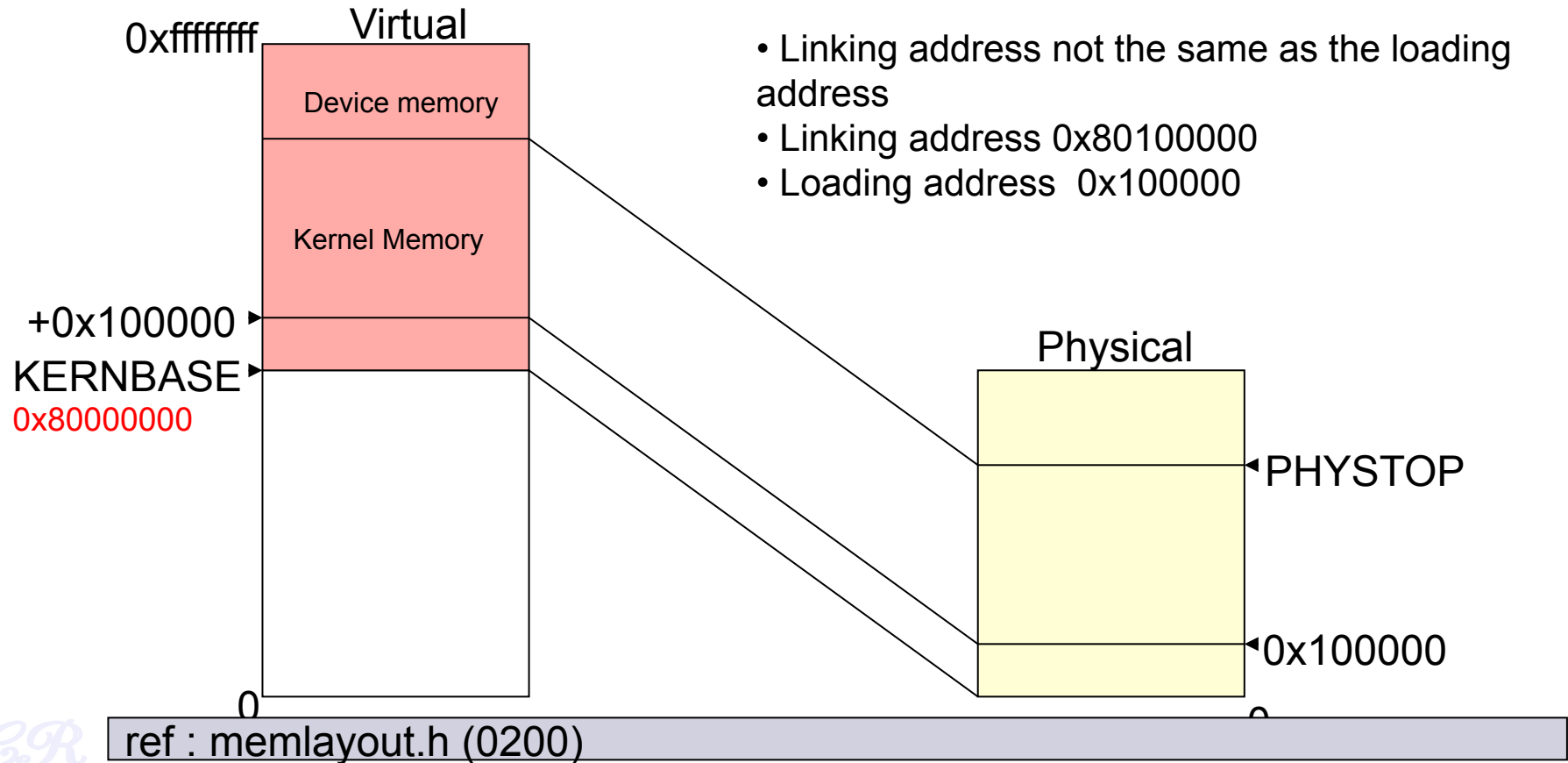


- Kernel memory mapped into every process
 - easy to switch between kernel and user modes
- **VA(KERNBASE:+PHYSTOP) → PA(0:PHYSTOP)**
 - convert from VA to PA just by +/- KERNBASE
 - easily write to physical page
 - limits size of physical memory to 2GB

BR

ref : memlayout.h (0200)

Virtual Address Space



Converting virtual to physical in kernel space

```
#define V2P(a) (((uint) (a)) - KERNBASE)
#define P2V(a) (((void *) (a)) + KERNBASE)

#define V2P_WO(x) ((x) - KERNBASE) // same as V2P, but without casts
#define P2V_WO(x) ((x) + KERNBASE) // same as P2V, but without casts
```

Enable Paging

1. Start of with a quick solution

Aim is get the kernel running with paging enabled

-- create a minimal paging environment

---- two pages of 4MB size (just sufficient to hold the OS)

2. Have an elaborate paging mechanism

Create pages for each 4KB RAM block

Allocate and manage free memory

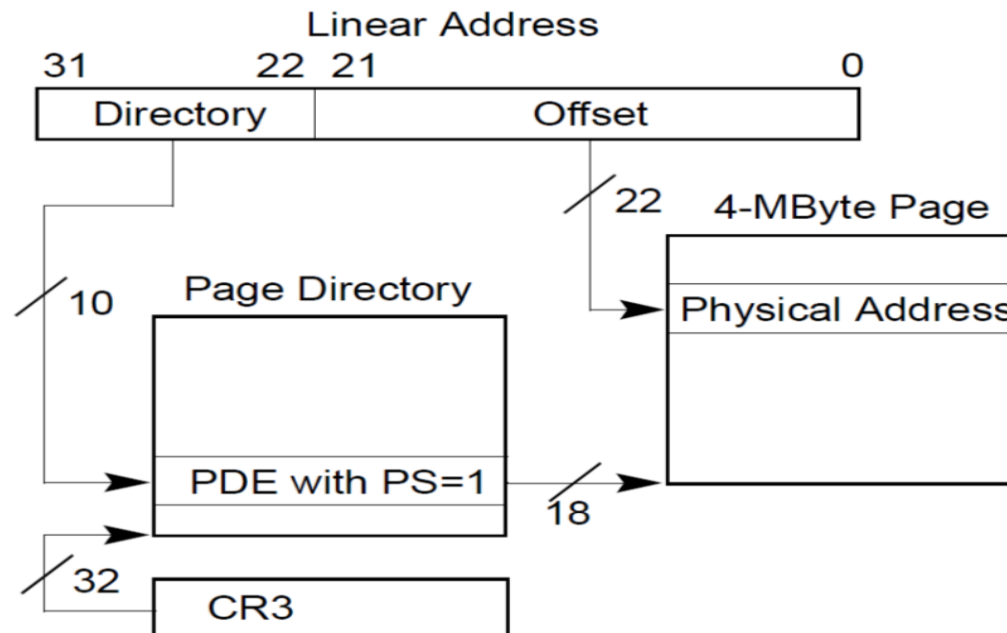
Enable Paging

```
1035 .globl _start
1036 _start = V2P_WO(entry)
1037
1038 # Entering xv6 on boot processor, with paging off.
1039 .globl entry
1040 entry:
1041     # Turn on page size extension for 4Mbyte pages
1042     movl    %cr4, %eax
1043     orl    $(CR4_PSE), %eax
1044     movl    %eax, %cr4
1045     # Set page directory
1046     movl    $(V2P_WO(entrypgdir)), %eax
1047     movl    %eax, %cr3
1048     # Turn on paging.
1049     movl    %cr0, %eax
```

Aim is to create two
4MB pages

Turn on Page size
extension

4MB Pages



Enable Paging

```
1035 .globl _start
1036 _start = V2P_WO(entry)
1037
1038 # Entering xv6 on boot processor, with paging off.
1039 .globl entry
1040 entry:
1041 # Turn on page size extension for 4Mbyte pages
1042 movl    %cr4, %eax
1043 orl    $(CR4_PSE), %eax
1044 movl    %eax, %cr4
1045 # Set page directory
1046 movl    $(V2P_WO(entrypgdir)), %eax
1047 movl    %eax, %cr3
1048 # Turn on paging.
1049 movl    %cr0, %eax
```

Set Page Directory

2

```
1316 __attribute__((__aligned__(PGSIZE)))
1317 pde_t entrypgdir[NPDENTRIES] = {
1318 // Map VA's [0, 4MB) to PA's [0, 4MB)
1319 [0] = (0) + PTE_P + PTE_W + PTE_PS,
1320 // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
1321 [KERNBASE>>PDXSHIFT] = (0) + PTE_P + PTE_W + PTE_PS,
1322 };
```

Kernel memory setup

- First setup two 4MB pages

- Entry 0:

Virtual addresses 0 to 0x04000000 → Physical addresses 0 to 4MB

- Entry 512:

Virtual addresses 0x80000000 to 0x84000000 →

Physical addresses 0 to 4MB

```
1316 __attribute__((__aligned__(PGSIZE)))
1317 pde_t entrypgdir[NPDENTRIES] = {
1318     // Map VA's [0, 4MB) to PA's [0, 4MB)
1319     [0] = (0) + PTE_P + PTE_W + PTE_PS,
1320     // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
1321     [KERNBASE >> PDXSHIFT] = (0) + PTE_P + PTE_W + PTE_PS,
1322 };
```

What would be the address generated before and immediately after paging is enabled?

before : 0x001000xx

Immediately after : 0x8001000xx

So the OS needs to be present at two memory ranges

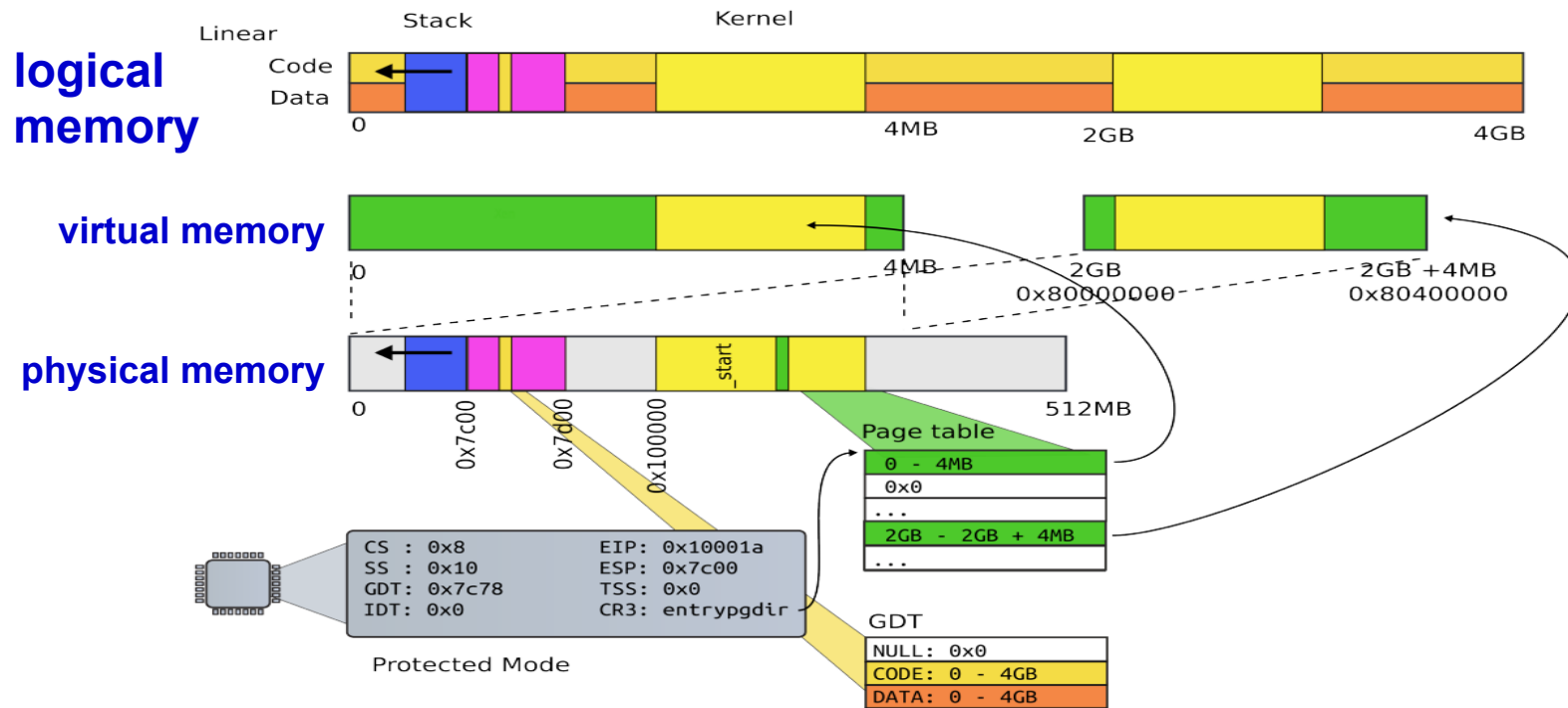


Enable Paging

```
1035 .globl _start
1036 _start = V2P_WO(entry)
1037
1038 # Entering xv6 on boot processor, with paging off.
1039 .globl entry
1040 entry:
1041     # Turn on page size extension for 4Mbyte pages
1042     movl    %cr4, %eax
1 1043     orl    $(CR4_PSE), %eax
1044     movl    %eax, %cr4
1045     # Set page directory
2 1046     movl    $(V2P_WO(entrypgdir)), %eax
1047     movl    %eax, %cr3
3 1048     # Turn on paging.
1049     movl    %cr0, %eax
```

Turn on Paging

First Page Table



courtesy Anton Burtsev, Univ. of Utah

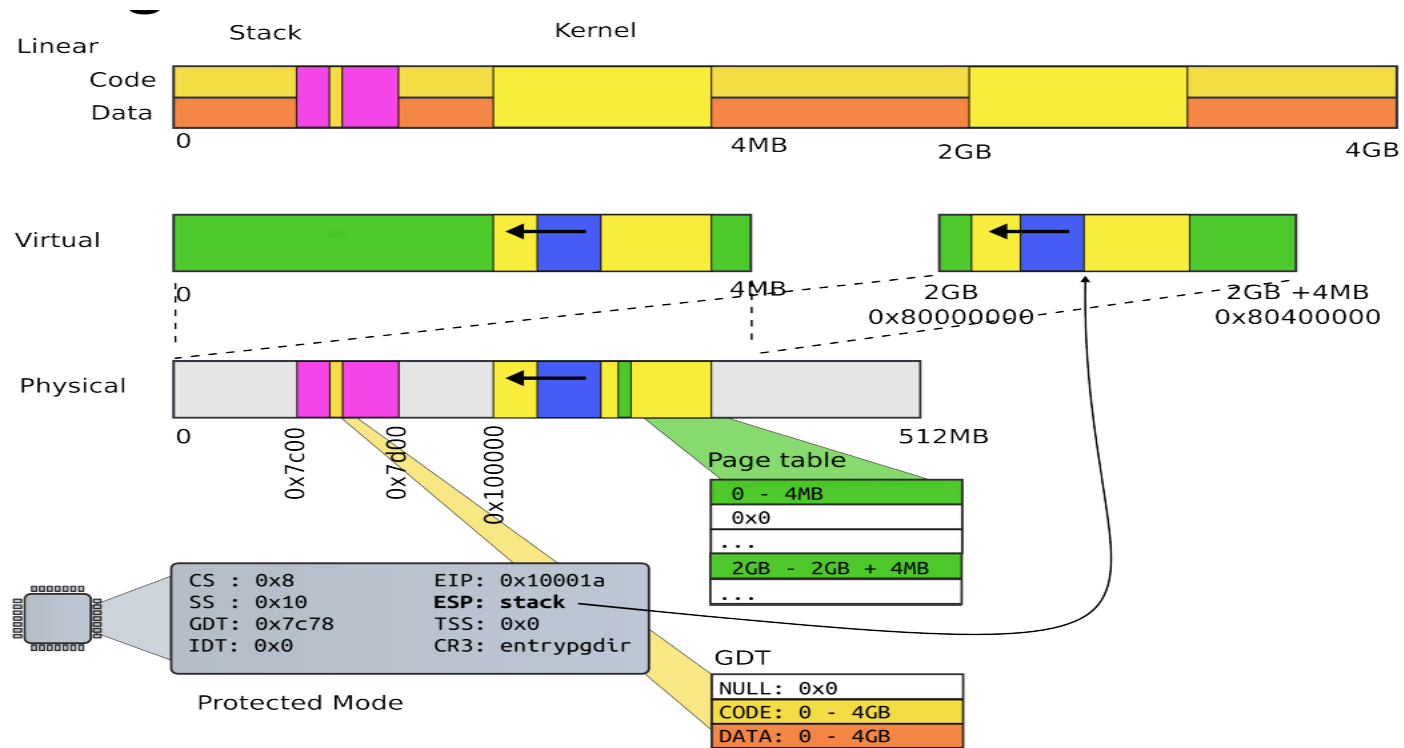
Execute main

```
1053 # Set up the stack pointer.
1054 movl $(stack + KSTACKSIZE), %esp
1055
1056 # Jump to main(), and switch to executing at
1057 # high addresses. The indirect call is needed because
1058 # the assembler produces a PC-relative instruction
1059 # for a direct jump.
1060 mov $main, %eax
1061 jmp *%eax
1062
1063 .comm stack, KSTACKSIZE
```

Set up stack

Jump to main

Stack



courtesy Anton Burtsev, Univ. of Utah

(Re)Initializing Paging

- Configure another page table
 - Map kernel only once making space for other user level processes
 - Map more physical memory, not just the first 4MB
 - Use 4KB pages instead of 4MB pages
 - 4MB pages very wasteful if processes are small
 - Xv6 programs are a few dozen kilobytes

main → kvmalloc → setupkvm



Setup kernel vm

```

KERNBASE = 0x80000000
KERNLINK = KERNBASE + 0x100000
PHYSTOP = 0xE000000
EXTMEM = 0x100000
    
```

1823

```

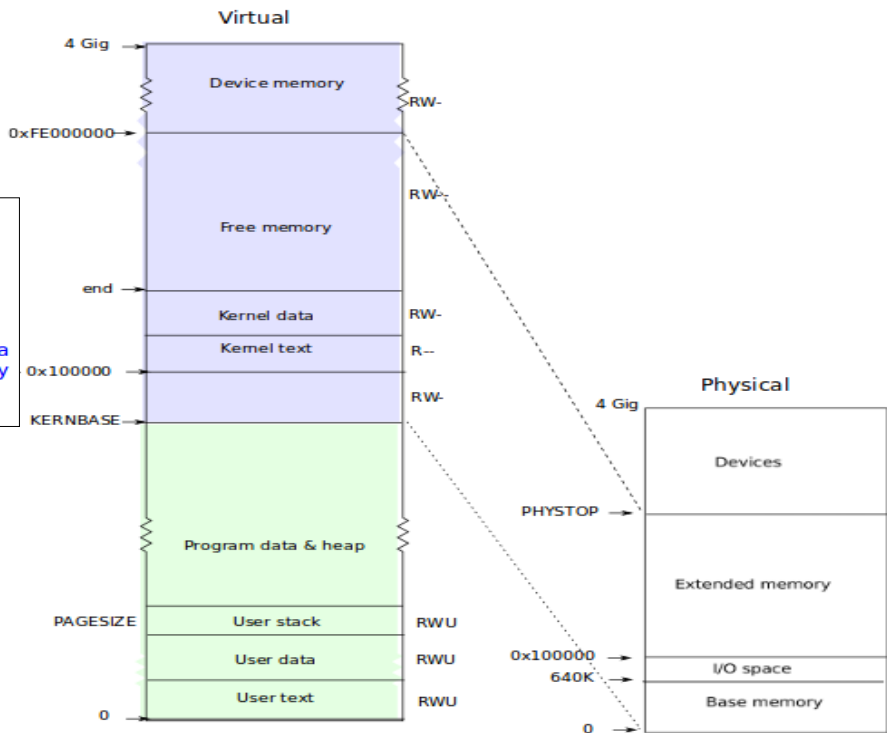
static struct kmap {
    void *virt;
    uint phys_start;
    uint phys_end;
    int perm;
} kmap[] = {
    { (void*)KERNBASE, 0,          EXTMEM,  PTE_W}, // I/O space
    { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0}, // kern text+rodata
    { (void*)data, V2P(data), PHYSTOP, PTE_W}, // kern data+memory
    { (void*)DEVSPACE, DEVSPACE, 0, PTE_W}, // more devices
};
    
```

Setting Up kernel pages (vm.c)

1. struct kmap

data obtained from linker script, which determines size of code+readonly data

- Kernel page tables set up in `kvmalloc()` (1857) (invoked from main)



Creating the Page Table Mapping for the kernel


- Enable paging
- Create/Fill page directory
- Create/Fill page tables
- Load CR3 register

Creating the Page Table Mapping for the kernel

- Enable paging
- Create/Fill page directory
- Create/Fill page tables
- Load CR3 register

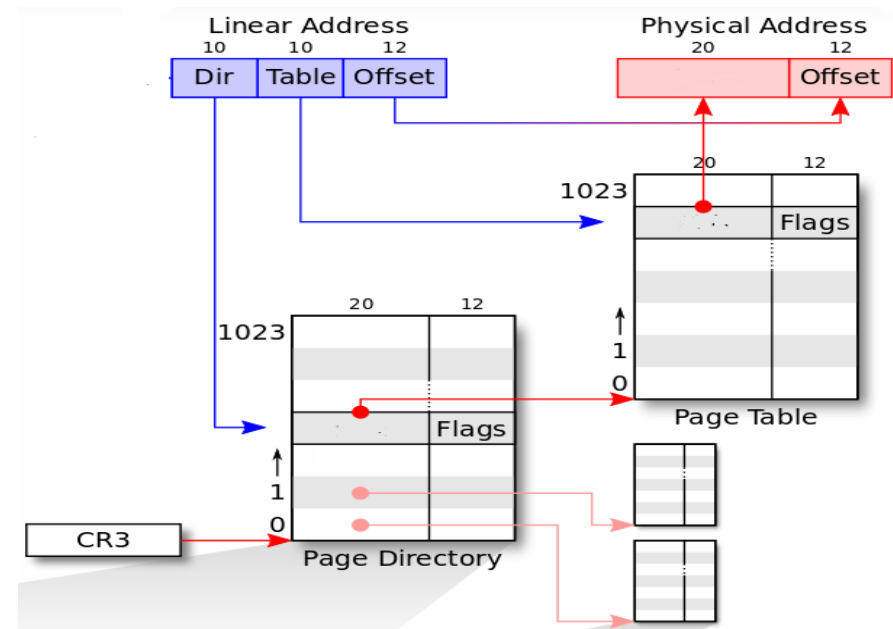
Setting paging enable bit in CR0 register (1049)

Creating the Page Table Mapping for the kernel

- Enable paging
- Create/Fill page directory 
- Create/Fill page tables
- Load CR3 register

walkpgdir (1754)

- Create a page directory entry corresponding to a virtual address.
- If page table is not present, then allocate it.
- $PDX(va)$: page directory index
- $PTE_ADDR(*pde)$: page directory entry
- $PTX(va)$: page table entry



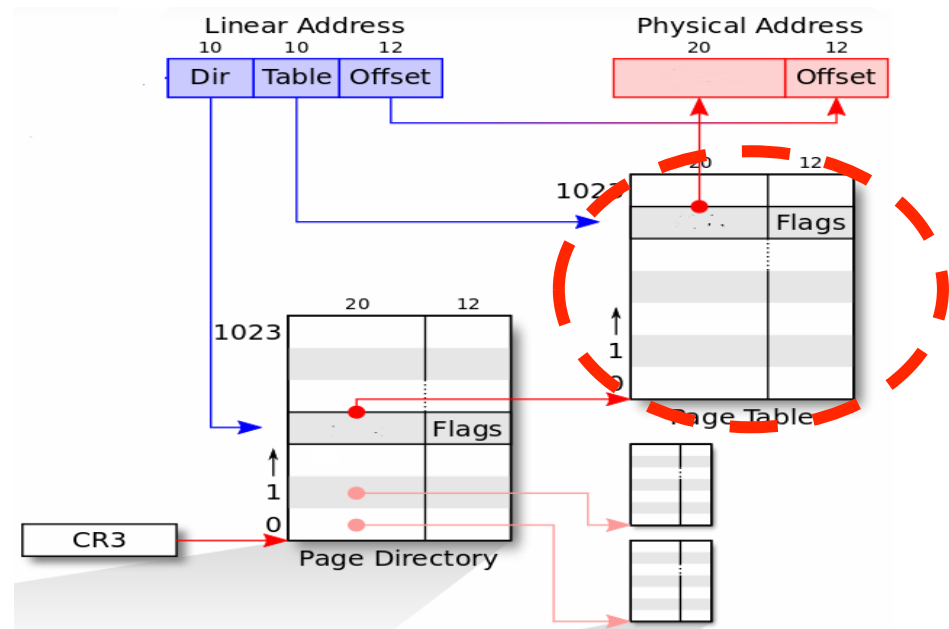
Creating the Page Table Mapping for the kernel

- Enable paging
- Create/Fill page directory
- Create/Fill page tables
- Load CR3 register

done in function mappages (1779)

mappages (1779)

- Fill page table entries mapping virtual addresses to physical addresses
- What are the contents?
 - Physical address
 - Permissions
 - Present bit

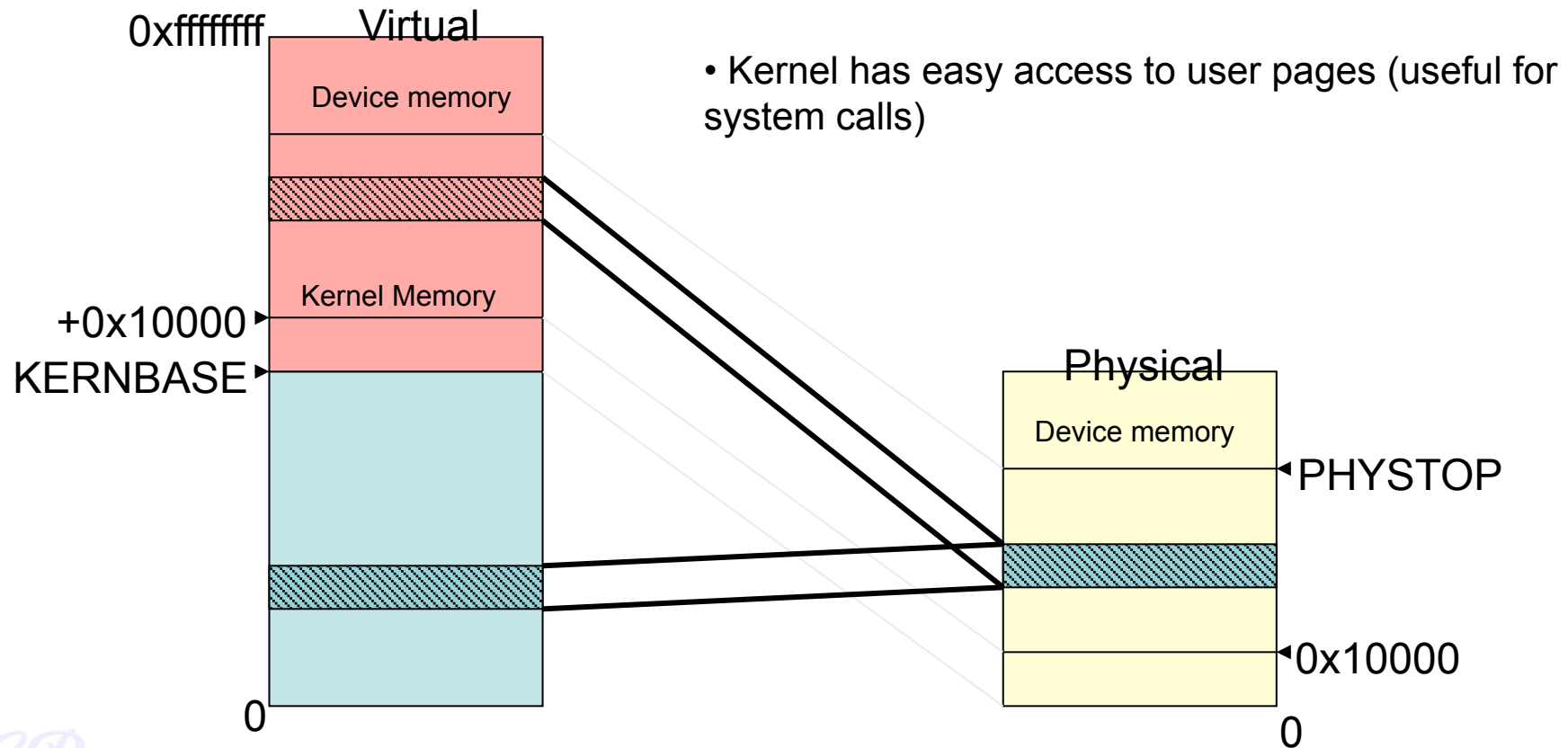


Creating the Page Table Mapping for the kernel

- Enable paging
- Create/Fill page directory
- Create/Fill page tables
- Load CR3 register

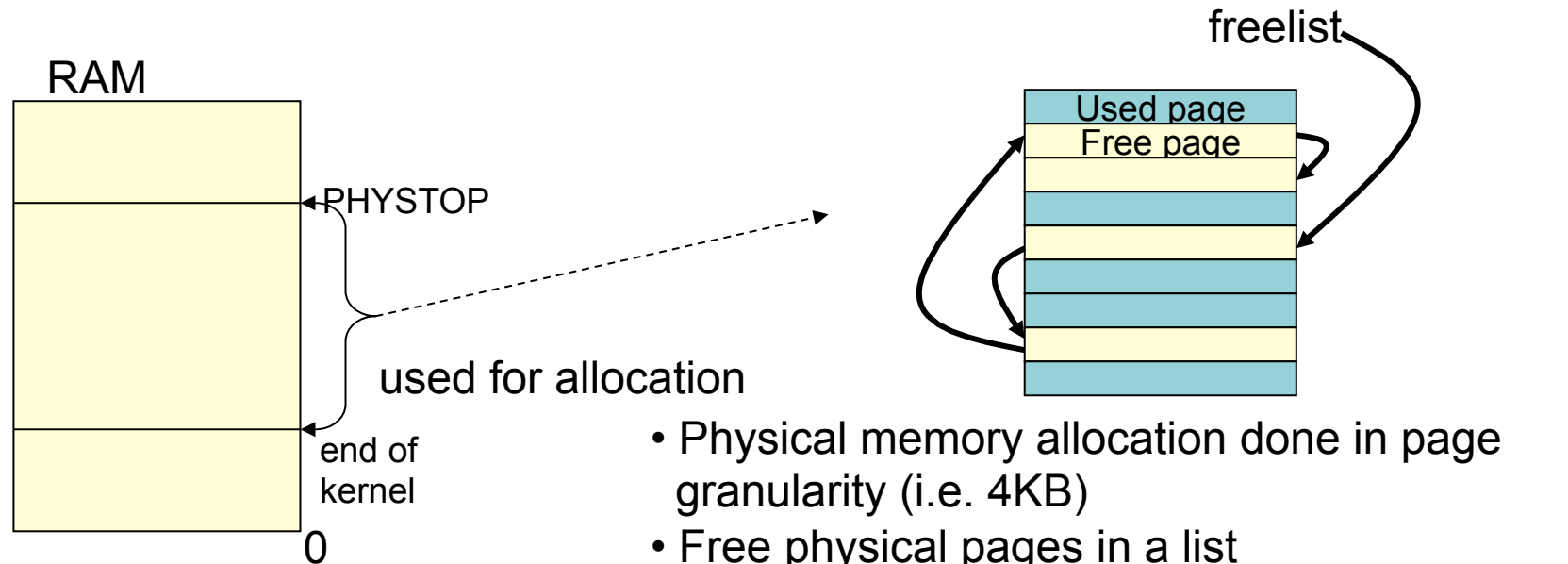
Load the CR3 register to point to the page directory.

User Pages mapped twice



Allocating Memory

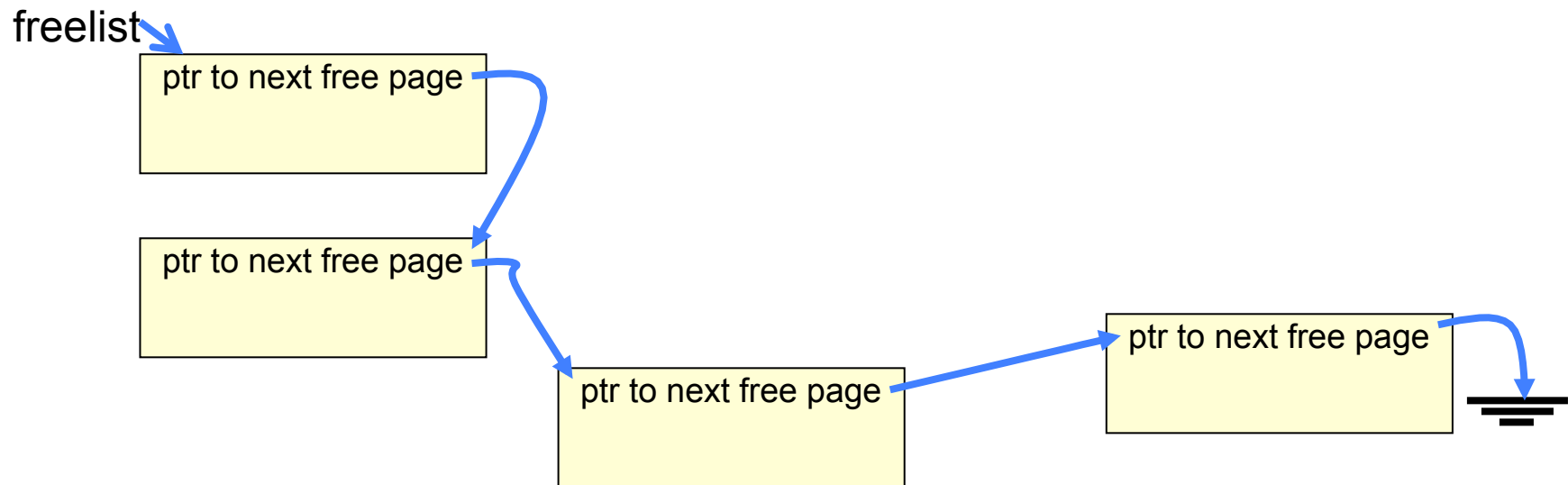
Allocating Pages (kalloc)



- Physical memory allocation done in page granularity (i.e. 4KB)
- Free physical pages in a list
- Page Allocation removes from list head (see function [kalloc](#))
- Page free adds to list head (see [kfree](#))

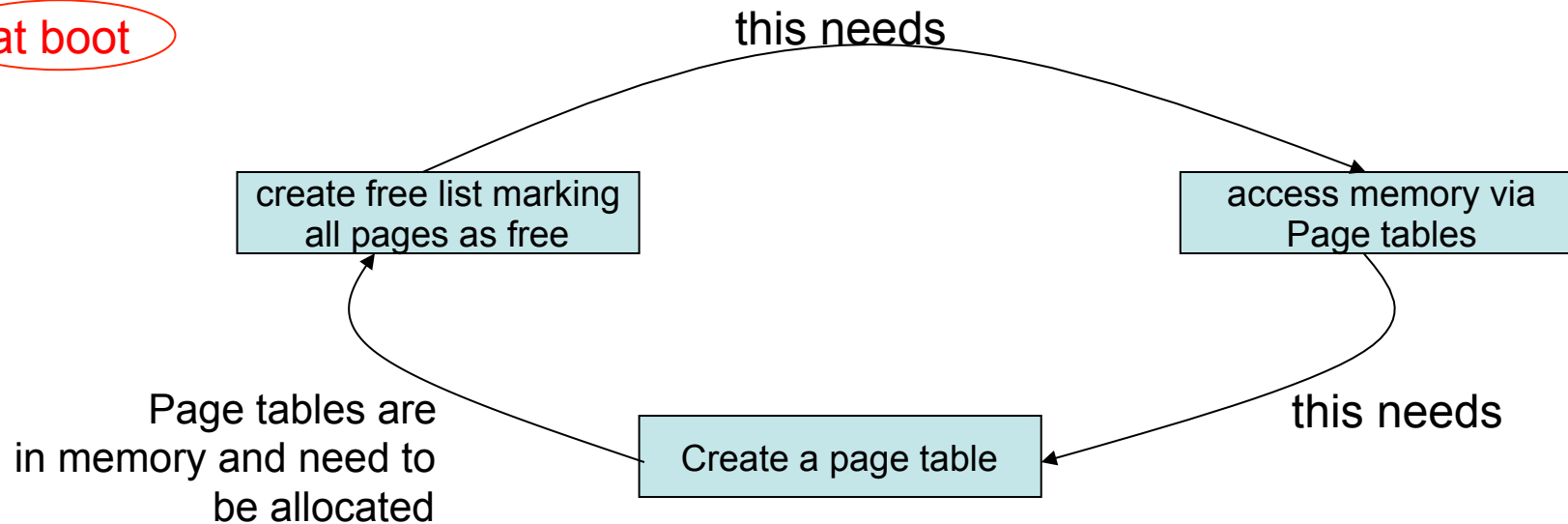
Freelist Implementation

- How is the freelist implemented?
 - No exclusive memory to store links (3014)



Initializing the list (chicken & egg problem)

at boot



Resolved by a separate page allocator during boot up, which allocates 4MB memory just after the kernel's data segment (see kinit1 and kinit2).

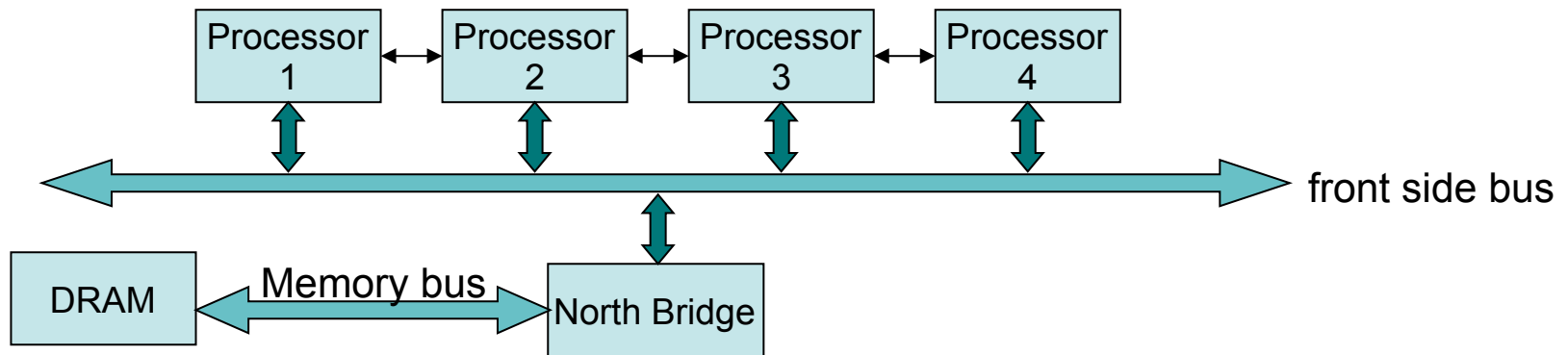


ref : kalloc.c (kinit1 and kinit2)

Per CPU Data

Recall

Memory is Symmetric Across Processors



- **Memory Symmetry**
 - All processors in the system share the same memory space
 - Advantage : Common operating system code
- However there are certain data which have to be unique to each processor
 - This is the **per-cpu data**
 - example, cpu id, scheduler context, taskstate, gdt, etc.

Naïve implementation of per-cpu data

```
// Per-CPU state
struct cpu {
    uchar id;                    // Local APIC ID; index into cpus[] below
    struct context *scheduler;  // swtch() here to enter scheduler
    struct taskstate ts;        // Used by x86 to find stack for interrupt
    struct segdesc gdt[NSEGS];  // x86 global descriptor table
    volatile uint started;      // Has the CPU started?
    int ncli;                    // Depth of pushcli nesting.
    int intena;                  // Were interrupts enabled before pushcli?

    // Cpu-local storage variables; see below
    struct cpu *cpu;
    struct proc *proc;          // The currently-running process.
};
```

```
struct cpu cpus[NCPU];
```

- An array of structures, each element in array corresponding to a processor
- Access to a per-cpu data, example : *cpu[cpunum()].ncli*
- This requires locking every time the cpu structure is accessed
 - eg. Consider process migrating from one processor to another while updating a

CR

ref : proc.h [23]

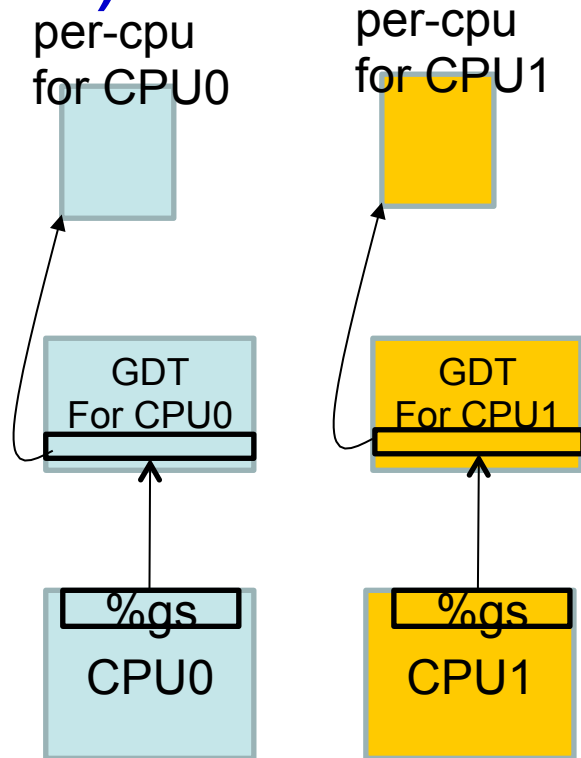
Alternate Solution (using CPU registers)

- CPU has several general purpose registers
 - The registers are unique to each processor (not shared)
- Use CPU registers to store per-cpu data
 - Must ensure the gcc does not use these registers for other purposes
- Fastest solution to our problem, but we do not have so many registers ☹️



Next best solution (xv6 implementation)

- In `seginit()`, which is run on each CPU initialization, the following is done.
 - GDTR will point upon cpu initialization to `cpus[cpunum()].gdt`.
 - (Thus, each processor will have its own private GDT in struct `cpu`).
- Have an entry which is unique for each processor
 - The base address field of `SEG_KCPU` entry in GDT is `&cpus[cpunum()].cpu (1731)`
 - `%gs` register loaded with `SEG_KCPU << 3`.
- Lock free access to per-cpu data
 - `%gs` indexes into the `SEG_KCPU` offset in GDT
 - This is unique for each processor



Using %gs

- Without locking or cpunum() overhead we have:
 - %gs:0 is cpus[cpunum()].cpu.
 - %gs:4 is cpus[cpunum()].proc.
- If we are interrupting user mode code then %gs might contain irrelevant value. Hence
 - In alltraps %gs is loaded with SEG_KCPU << 3.
 - (The interrupted code %gs is already on the trapframe.)
- gcc not aware of the existence of %gs, so it will no generate code messing up gs.

