Initializing Memory and Memory Management in xv6

Chester Rebeiro

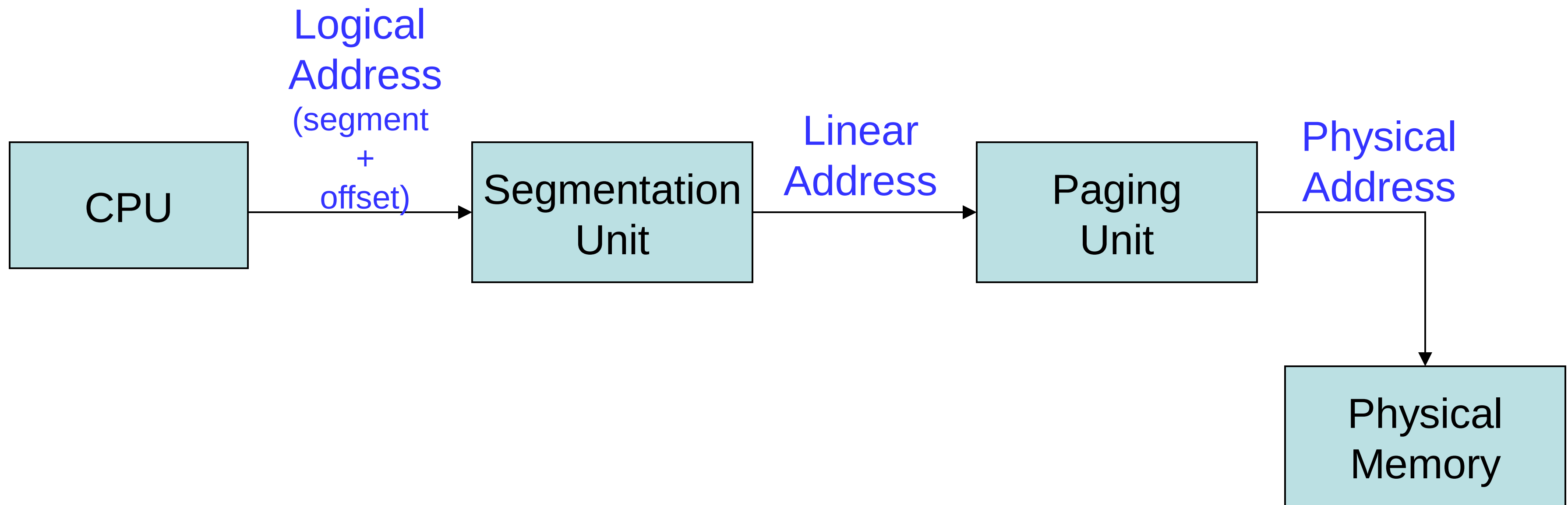
IIT Madras

Outline

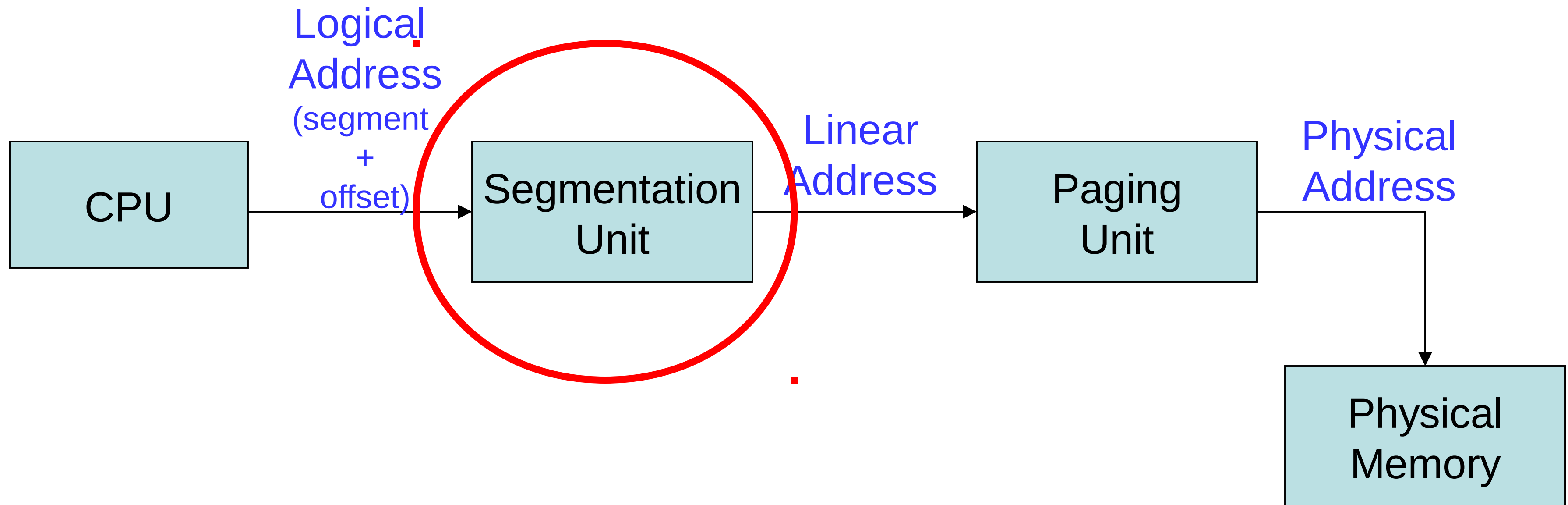
- Memory Management in x86
  - Segmentation
  - Virtual Memory
- Initializing memory in xv6
  - Initializing Pages
  - Initializing Segments
- Implementation of kalloc

CR

x86 address translation

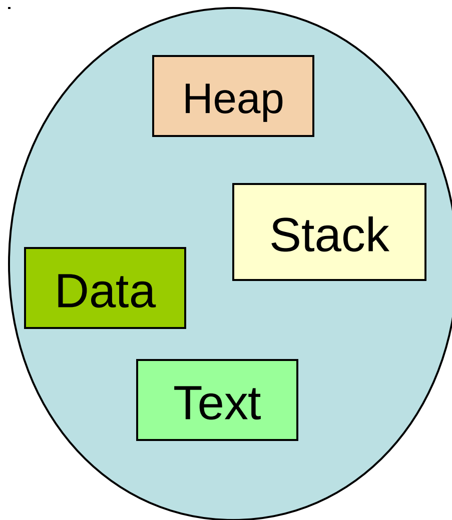


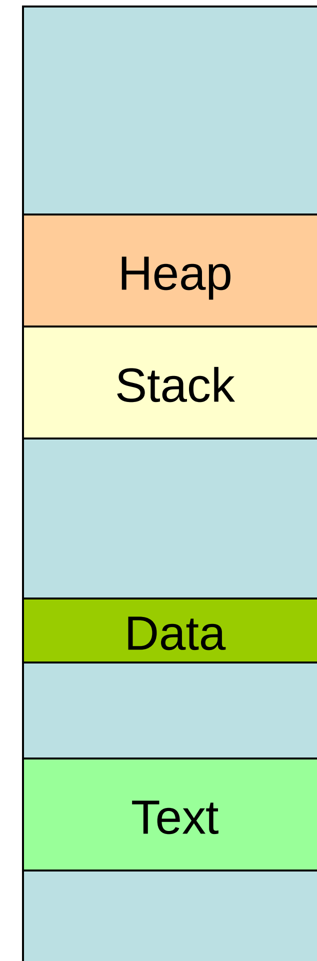
x86 address translation



Segmentation Unit

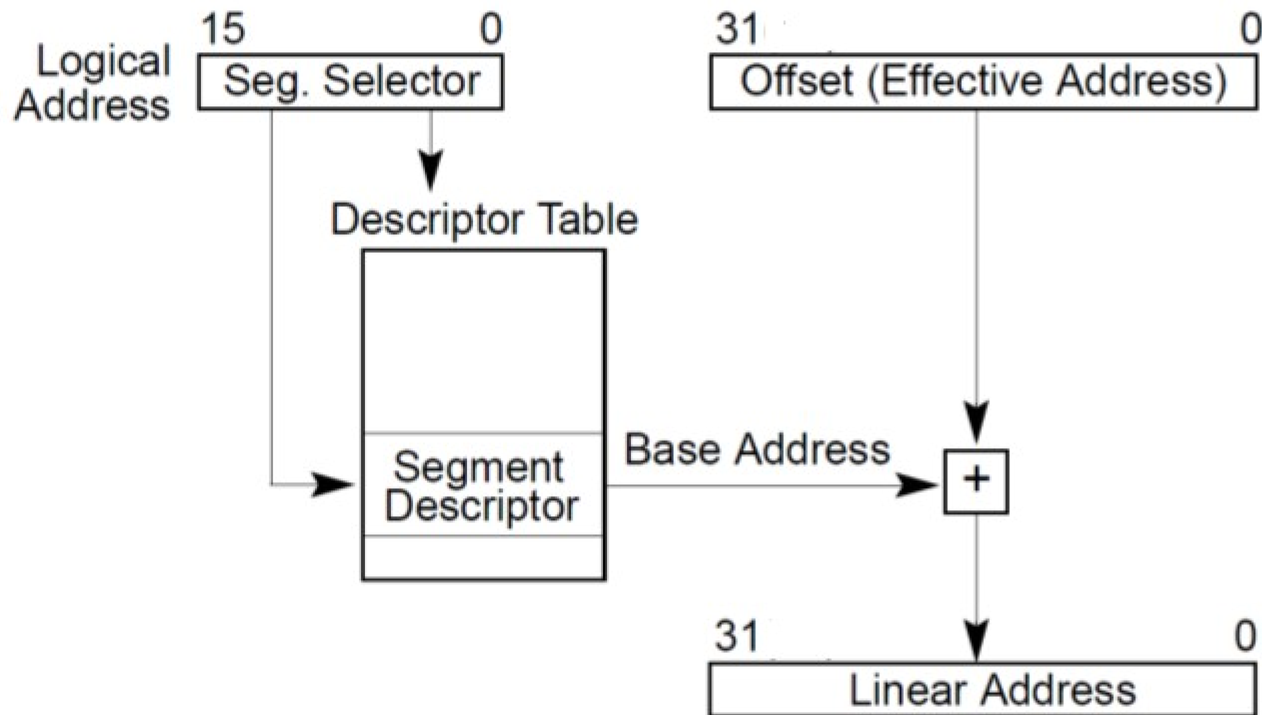
- Virtual address space of process divided into separate logical segments
- Each segment associated with a segment selector and offset



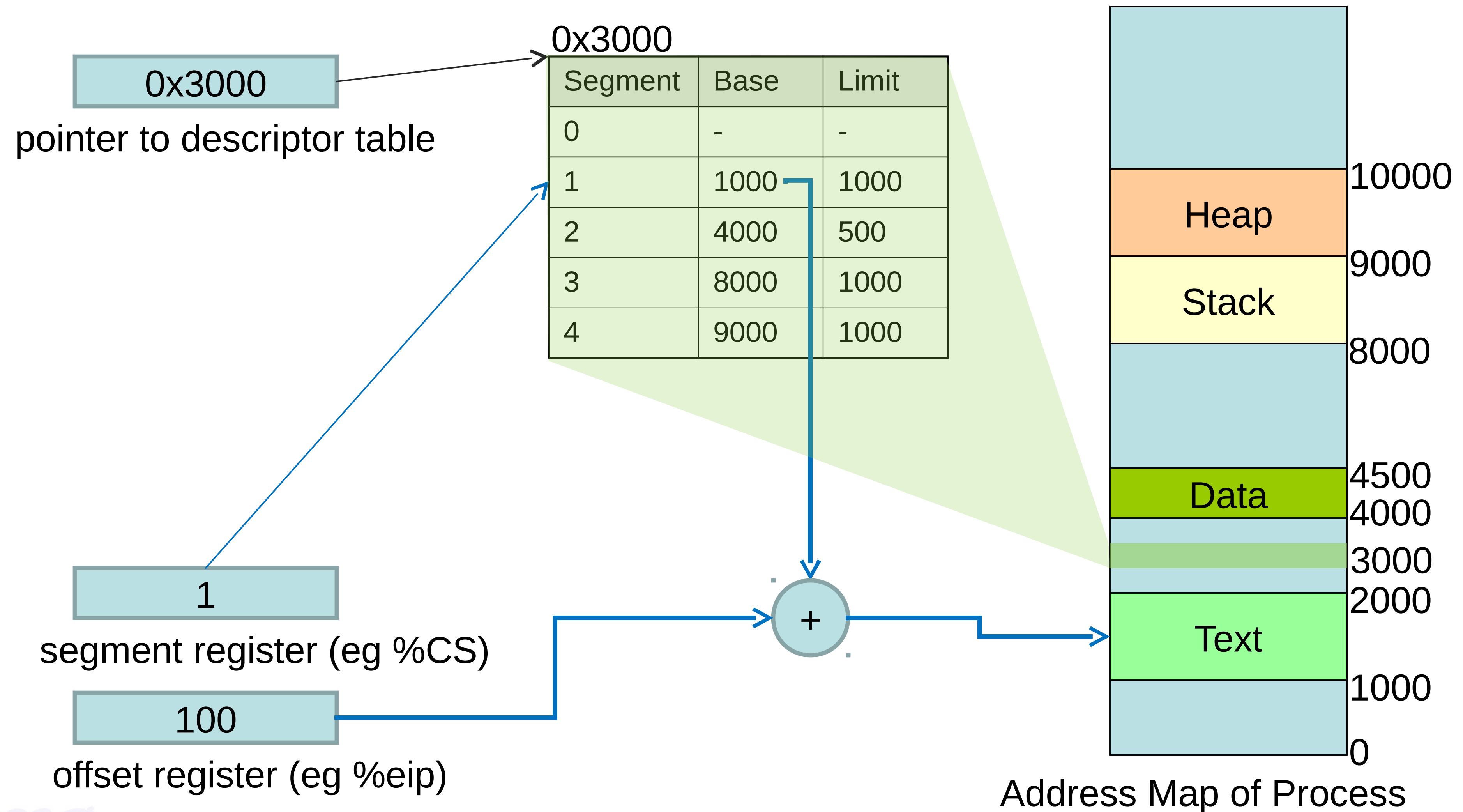


Address Map of Process

Segmentation
(*logical* to *linear* address)



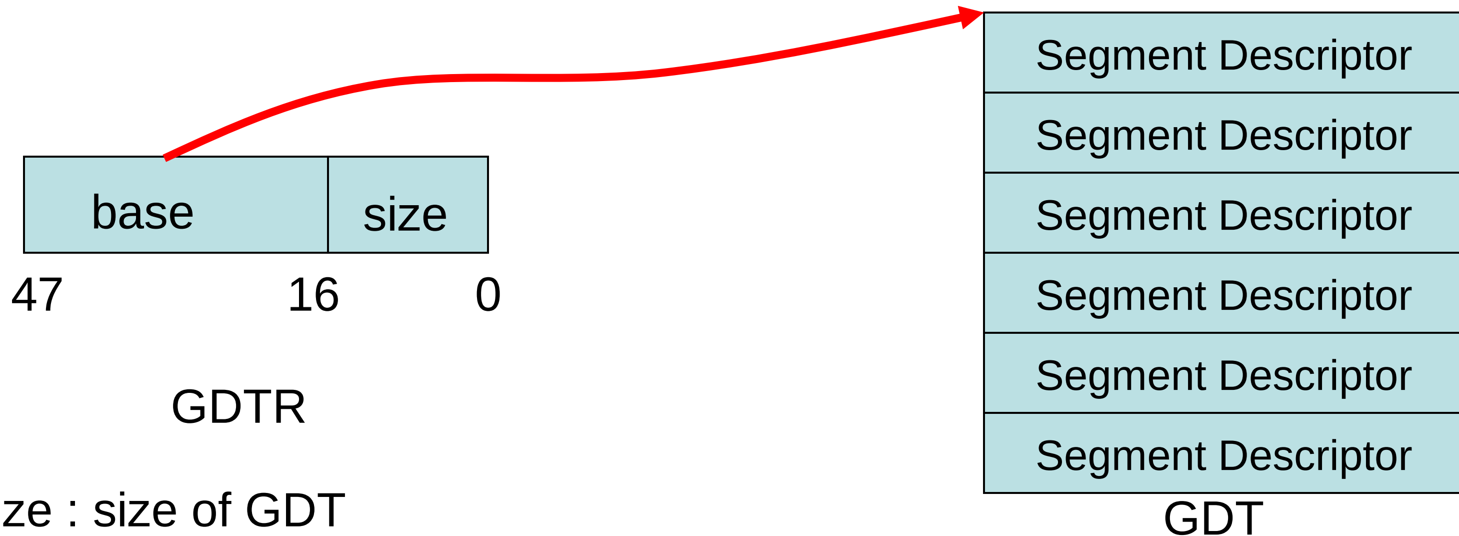
Example



Segment	Base	Limit
0	-	-
1	1000	1000
2	4000	500
3	8000	1000
4	9000	1000

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)
- Similar table called LDT present in x86 (not used by xv6!)



Segment Descriptor

- Base Address
  - 0 to 4GB
- Limit
  - 0 to 4GB
- Access Rights
  - Execute, Read, Write
  - Privilege Level (0-3)

Access	Limit
Base Address	

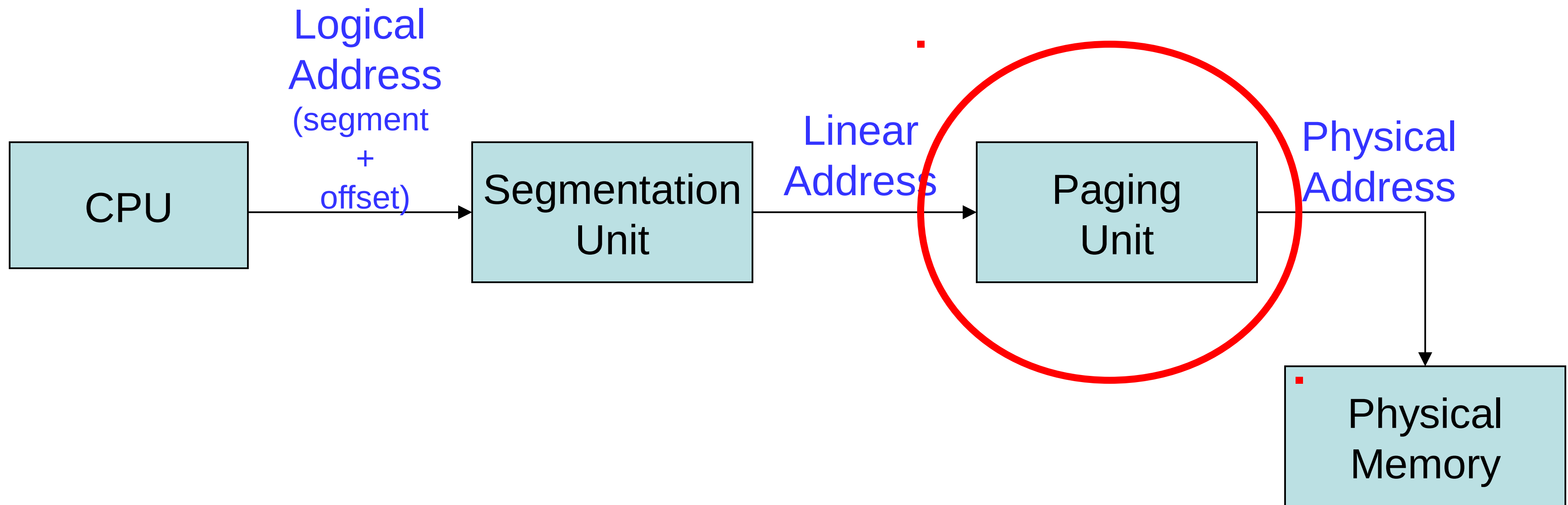
Segment Registers

- Holds 16 bit segment selectors
  - Points to offsets in GDT
- Segments associated with one of three types of storage
  - Code
    - %CS register holds segment selector
    - %EIP register holds offset
  - Data
    - %DS, %ES, %FS, %GS registers hold segment selector
  - Stack
    - %SS register holds segment selector
    - %SP register holds stack pointer

    (Note: Only one code segment and stack segment can be accessible at a time. But 4 data segments can be accessed simultaneously)

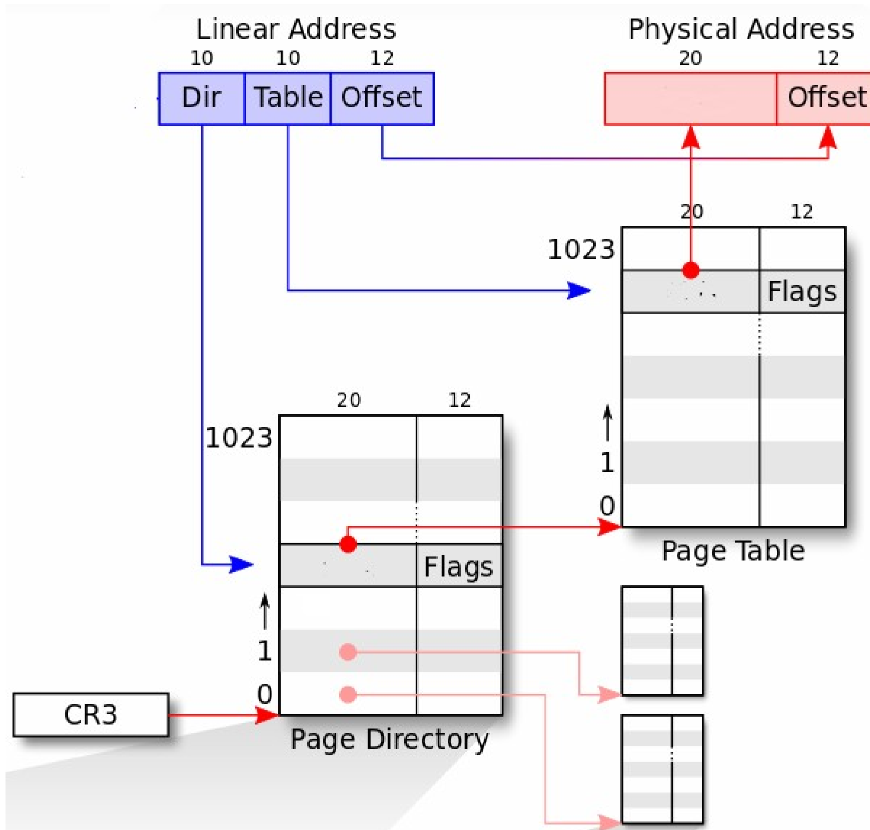
CR

x86 address translation



Linear to Physical Address

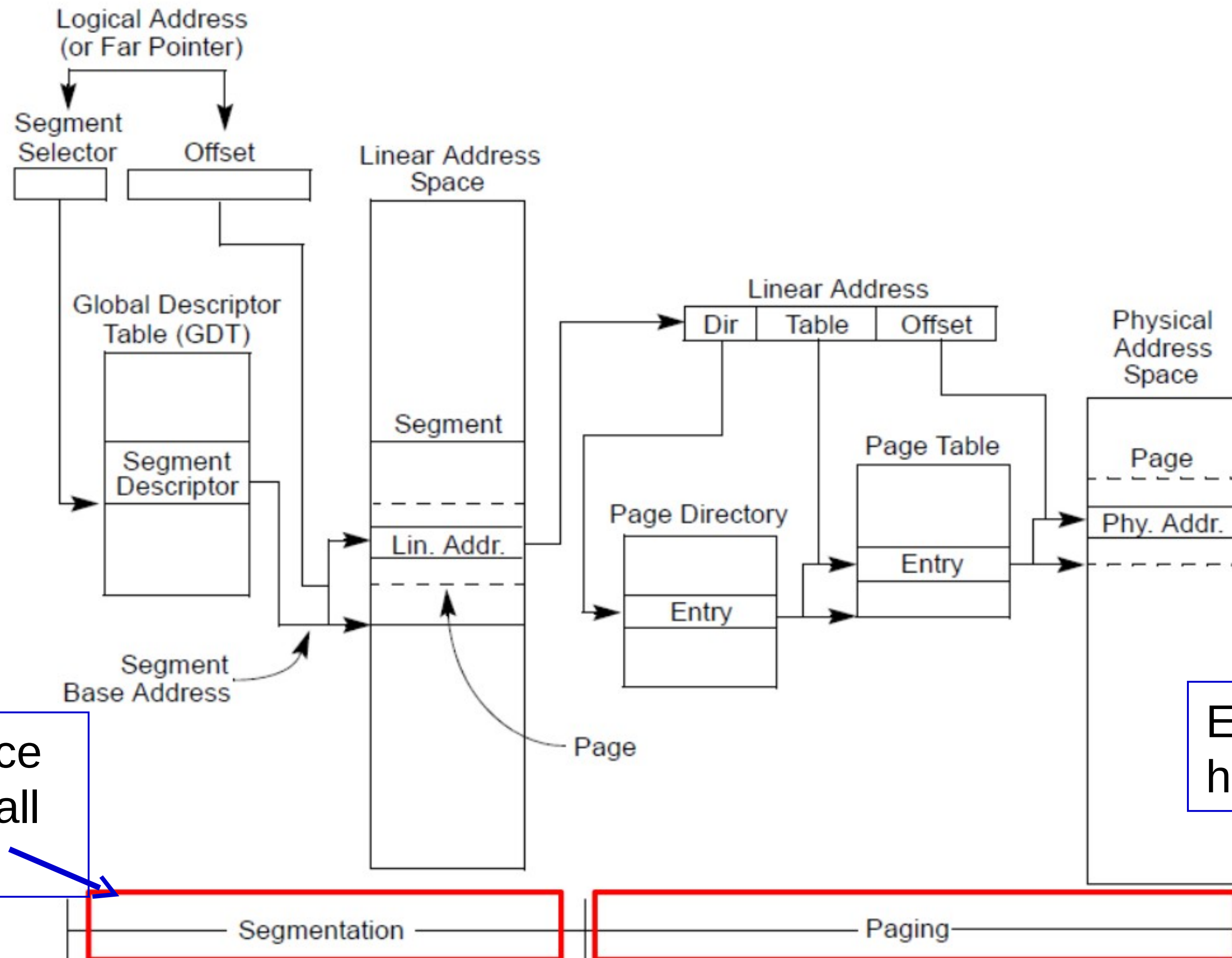
- 2 level page translation



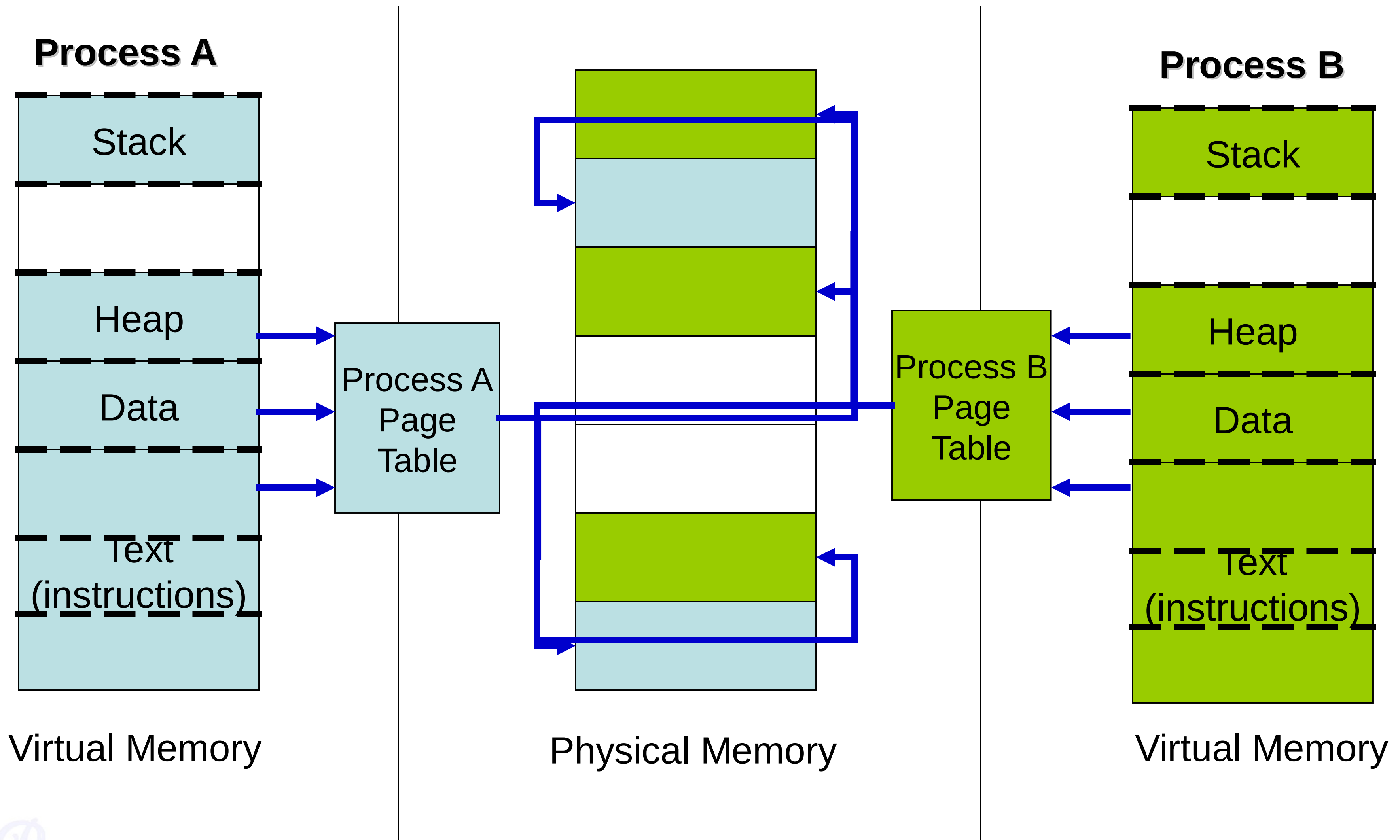
- How many page tables are present?
- What is the maximum size of the process' address space?
  - 4G

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

The full Picture

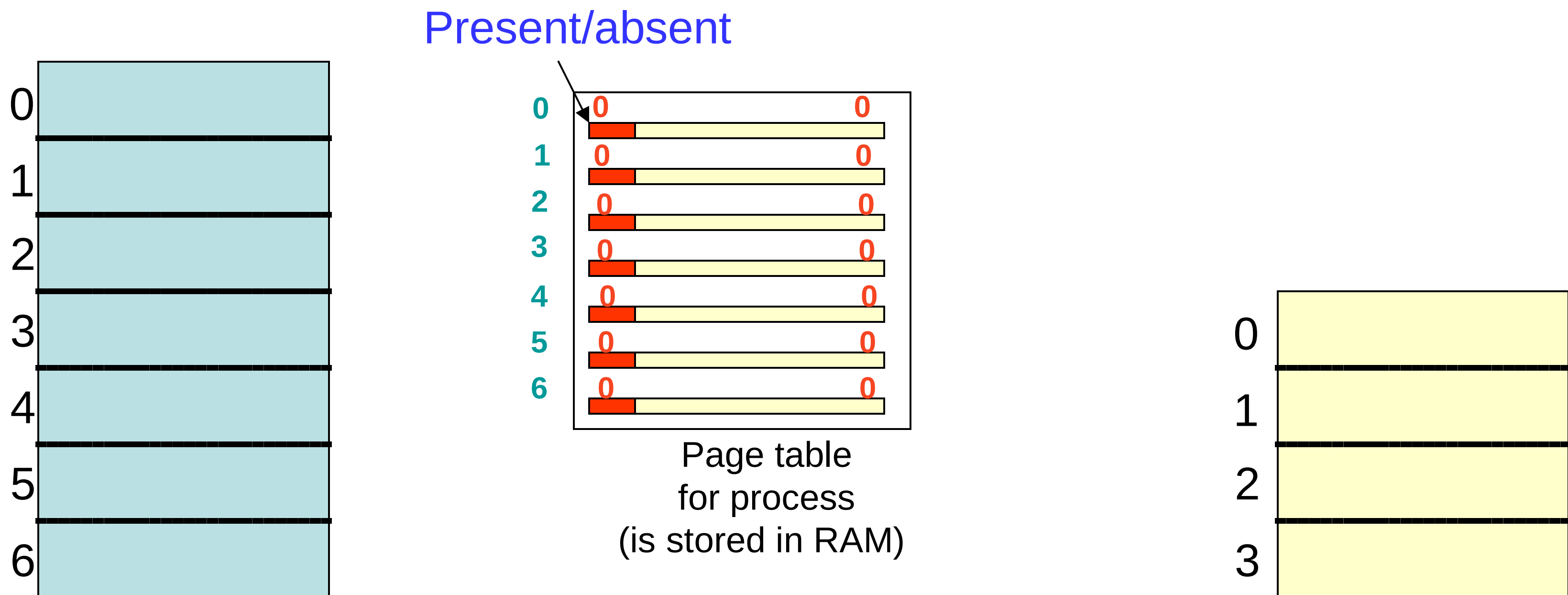


Virtual Address Advantages (Isolation between Processes)



Virtual Addressing Advantages Paging on Demand

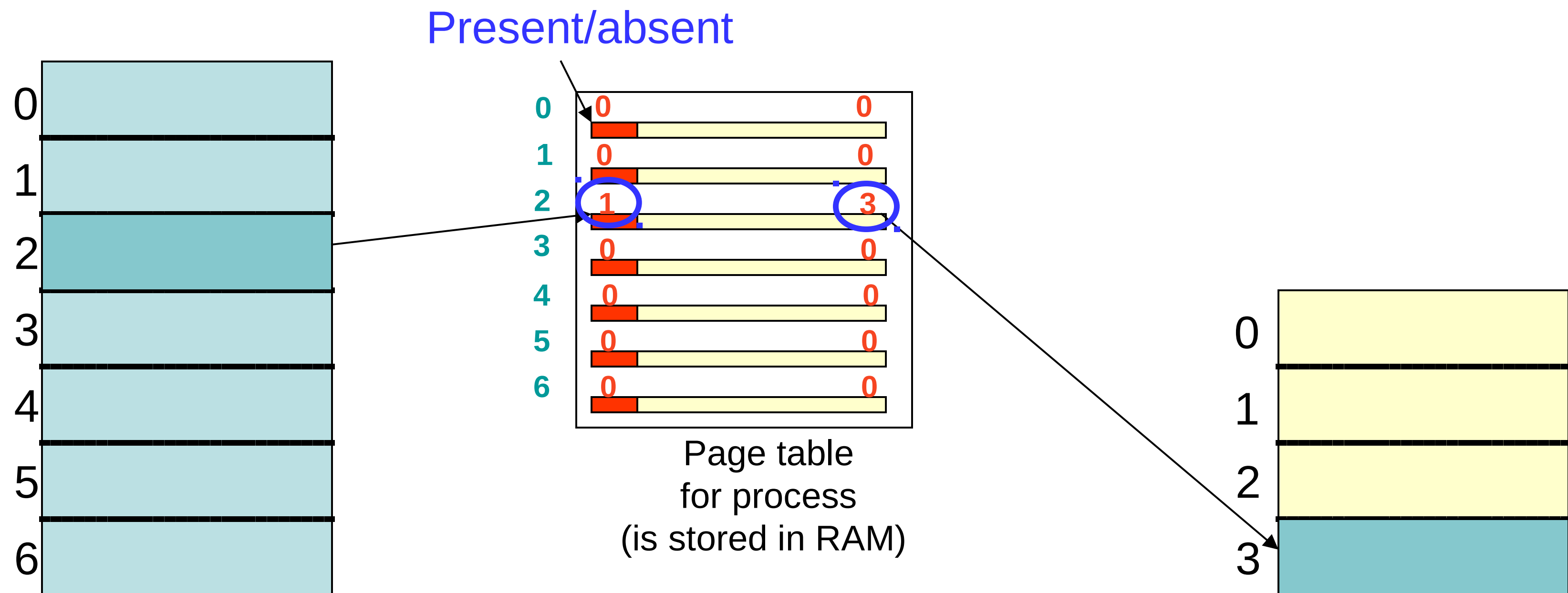
- RAM only loads pages into memory whenever needed
- When new program is executed page table is empty



Present/Absent bit : 1 if entry in page table is valid, 0 if invalid
In x86 : PTE_P

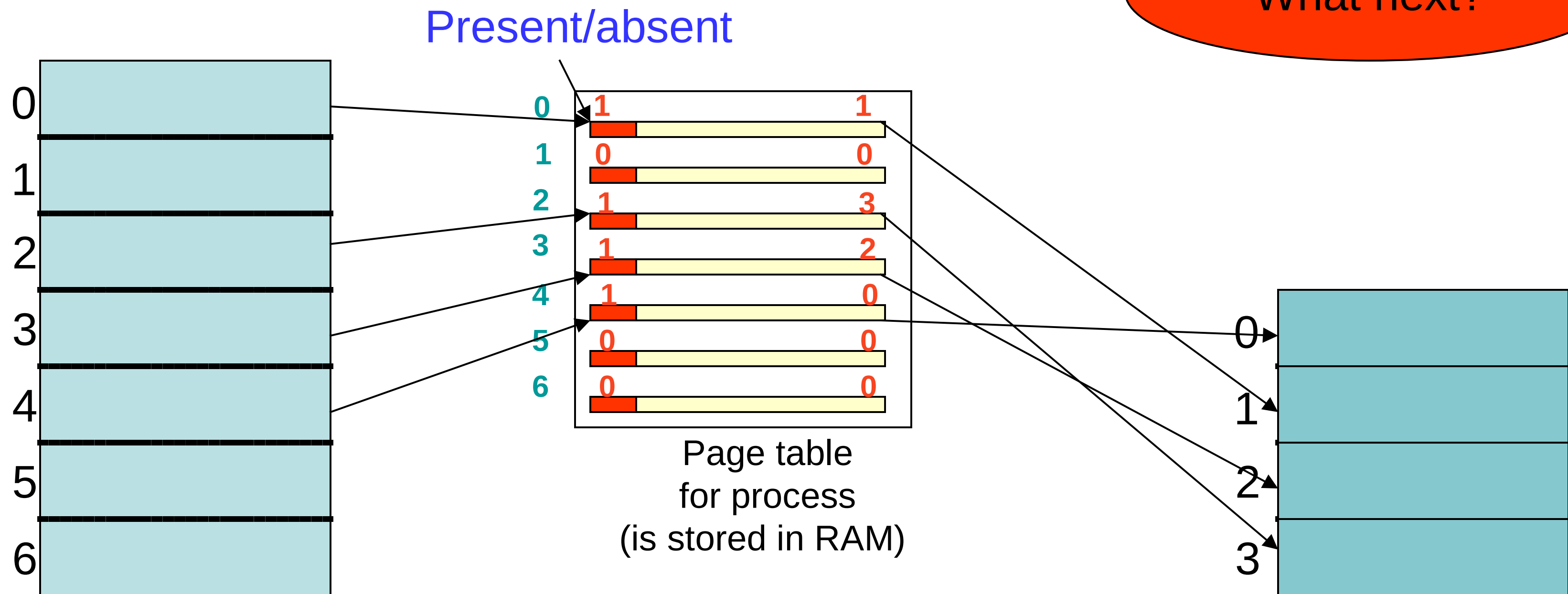
Paging on Demand (2)

- As data gets referenced, page table gets filled up.
- Page frame loaded from hard disk



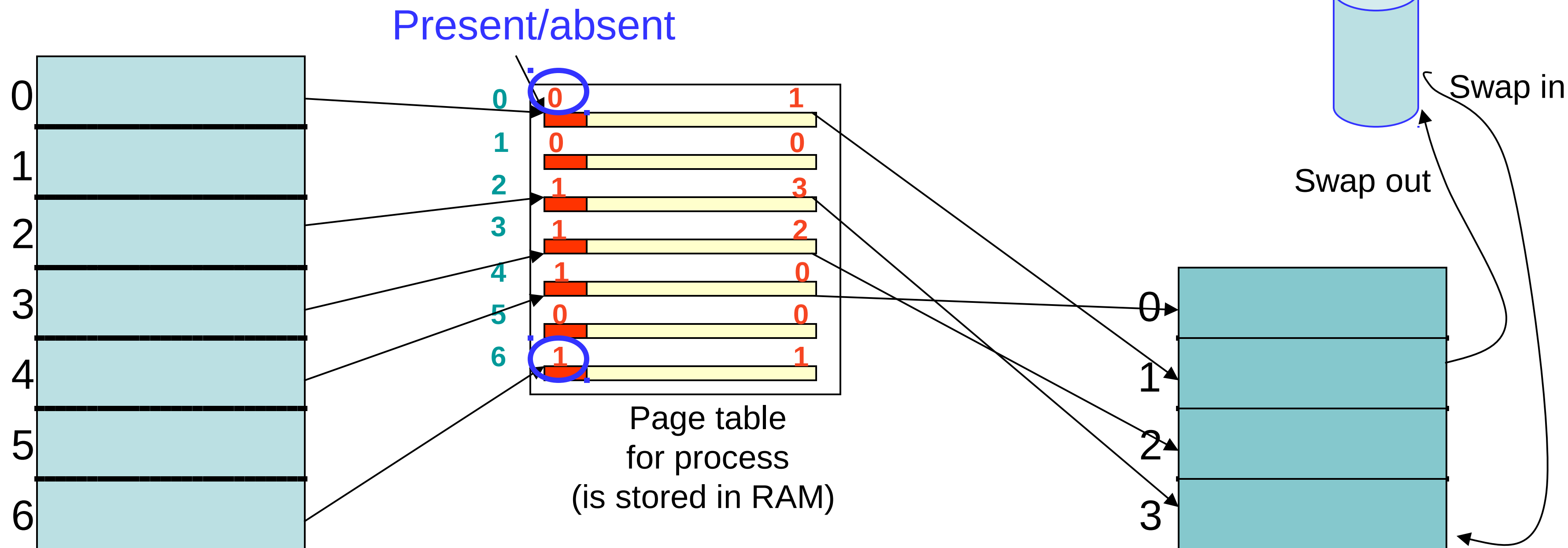
Paging on Demand(3)

- As execution progresses, more entries in page table get filled.
- Similarly, more frames in RAM get used
- Eventually, entire RAM is filled



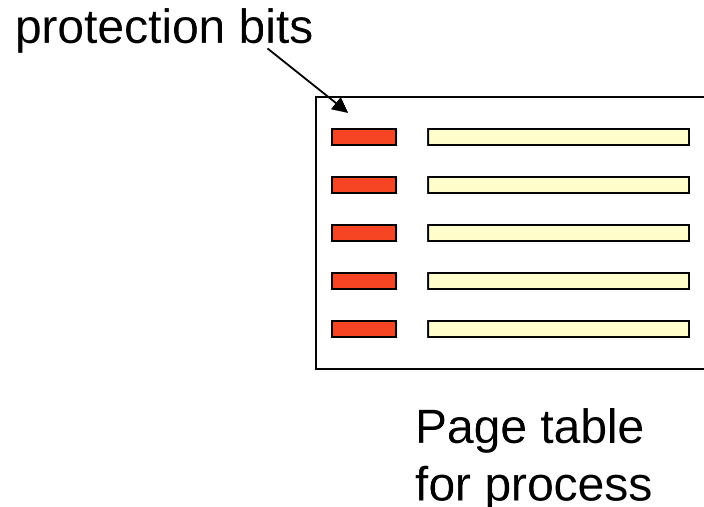
Paging on Demand (4)

- A particular frame is selected and swapped out into disk
- A new page swapped in
- Page table is updated



Virtual Memory achieves **Security**

- Security
  - Page tables augmented by protection bits



Page table
for process

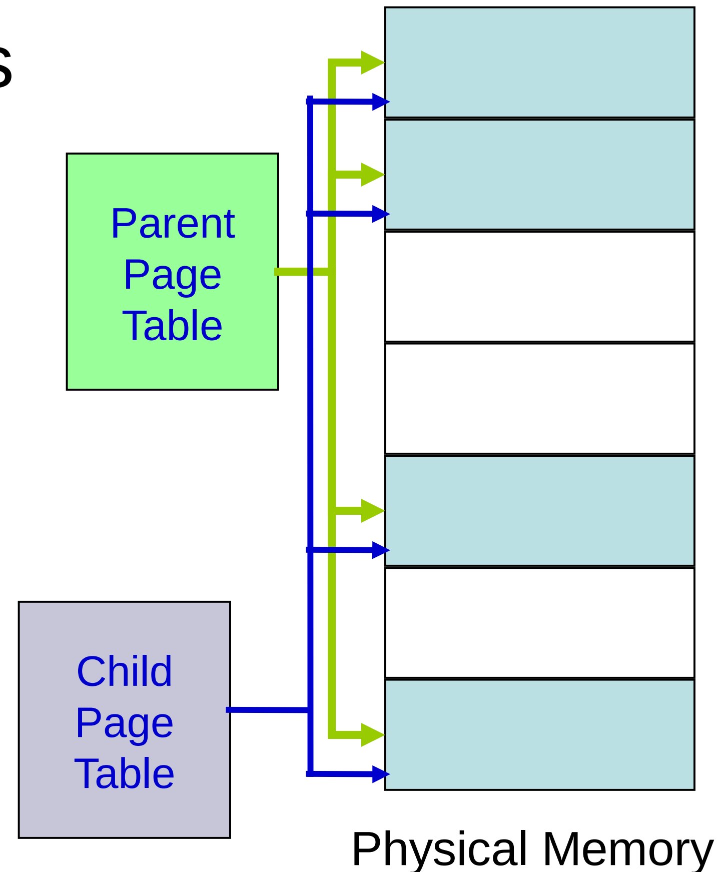
Protection Bits in x86

- PTE_W : controls if instructions are allowed to write to the page
- PTE_U : controls if user process can use the page. If not only kernel can use the page

These are checked by the MMU for each memory access!

Virtual Addressing Advantages (easy to make copies of a process)

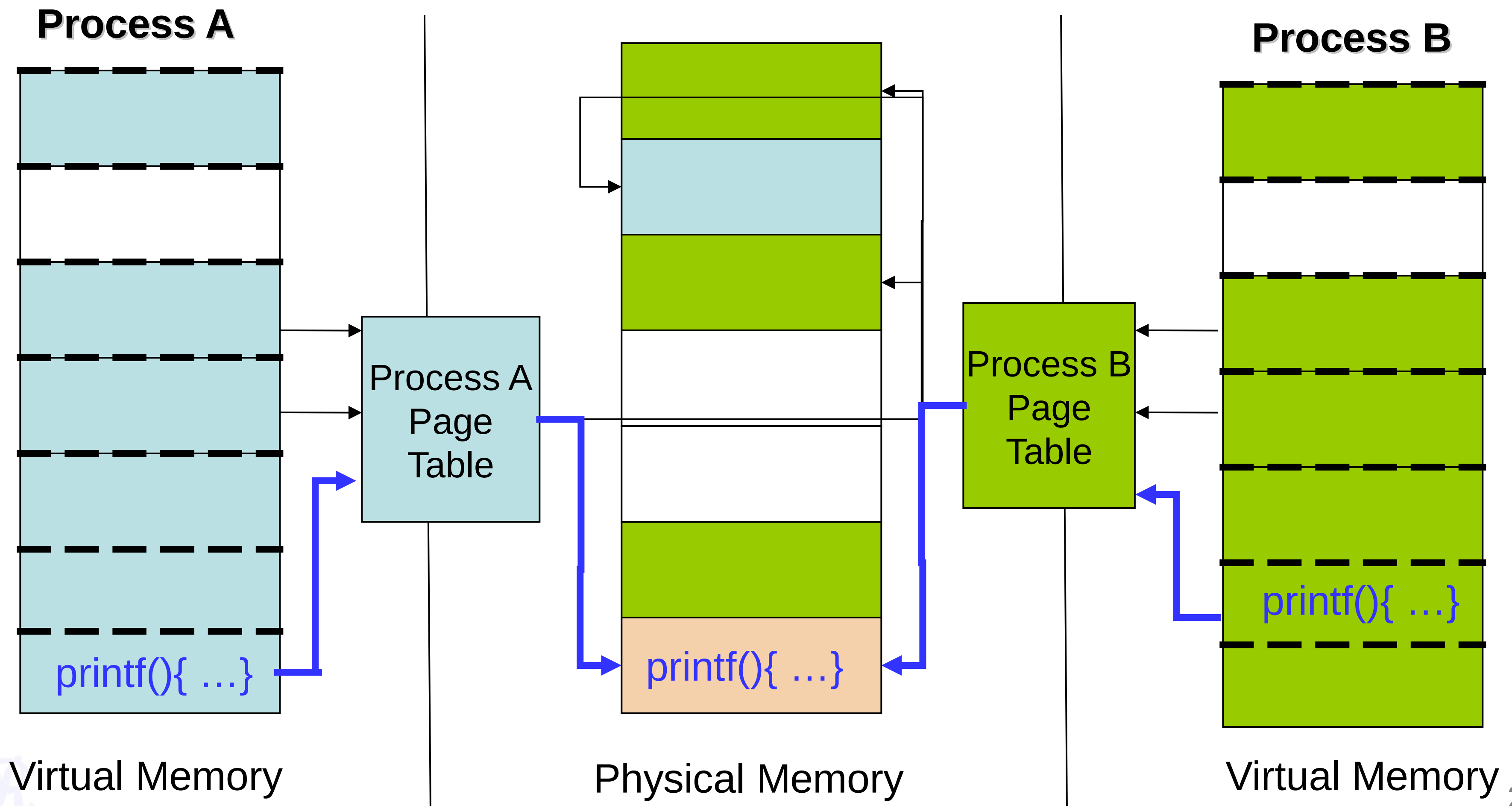
- Making a copy of a process is called forking.
  - Parent (is the original)
  - child (is the new process)
- When fork is invoked,
  - child is an exact copy of parent
    - When fork is called all pages are shared between parent and child
    - Easily done by copying the parent's page tables



CR

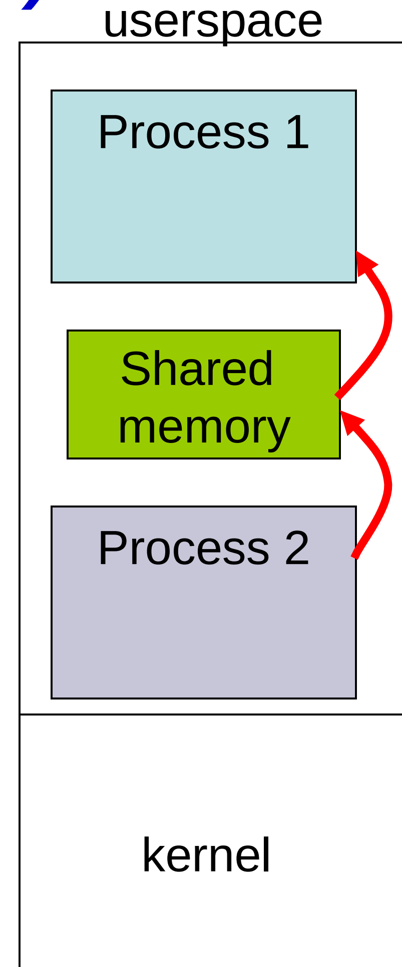
Virtual Addressing Advantages (Shared libraries)

- Many common functions such as *printf* implemented in shared libraries
- Pages from shared libraries, shared between processes



Virtual Addressing Advantages (Shared Memory)

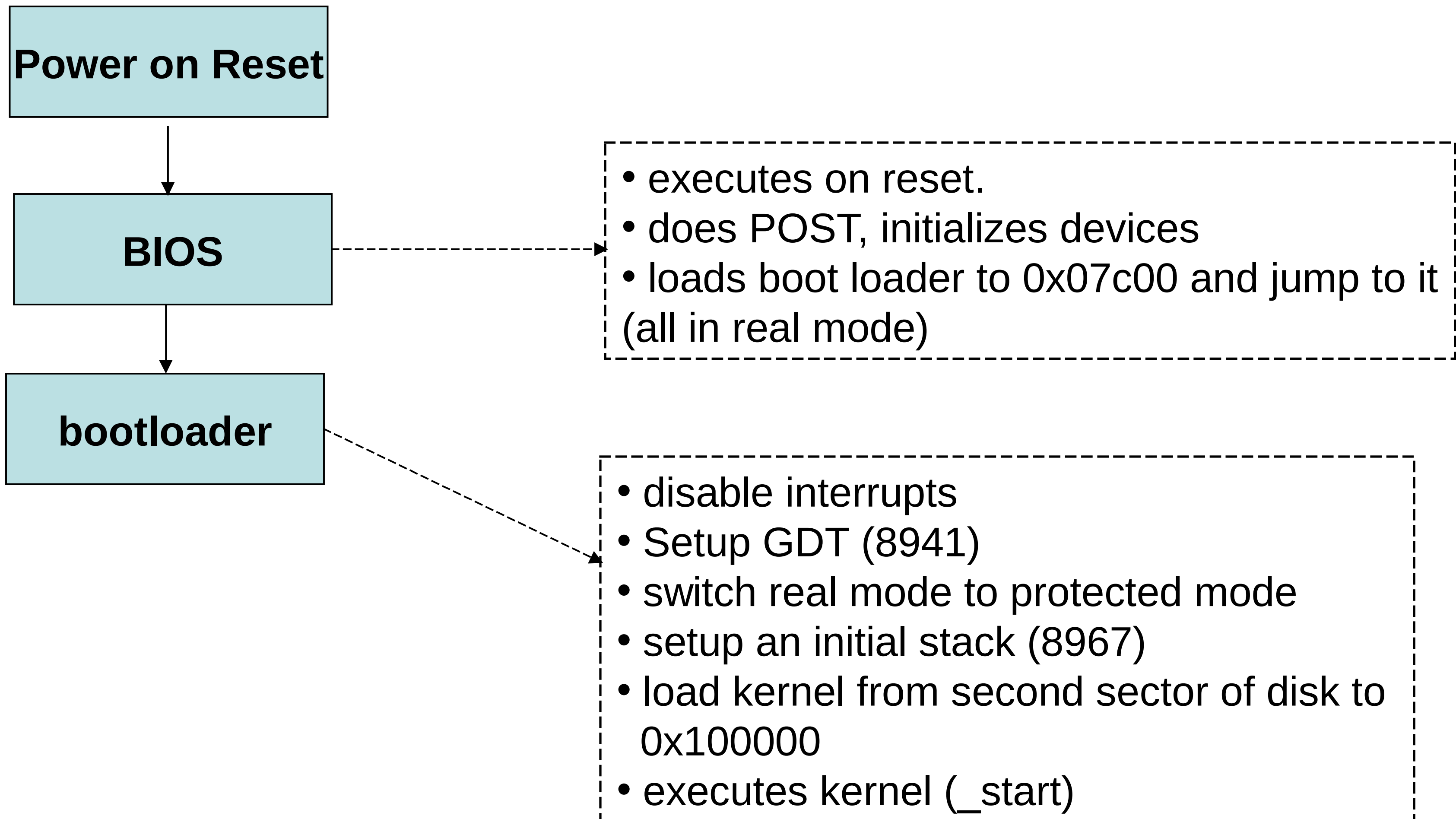
- Shared memory between processes easily implemented using virtual memories
  - Shared memory mapped to the same page
  - Writes from one process visible to another process



back to booting…

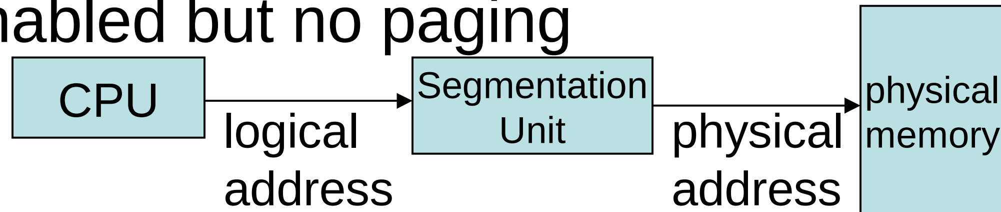
CR

so far…

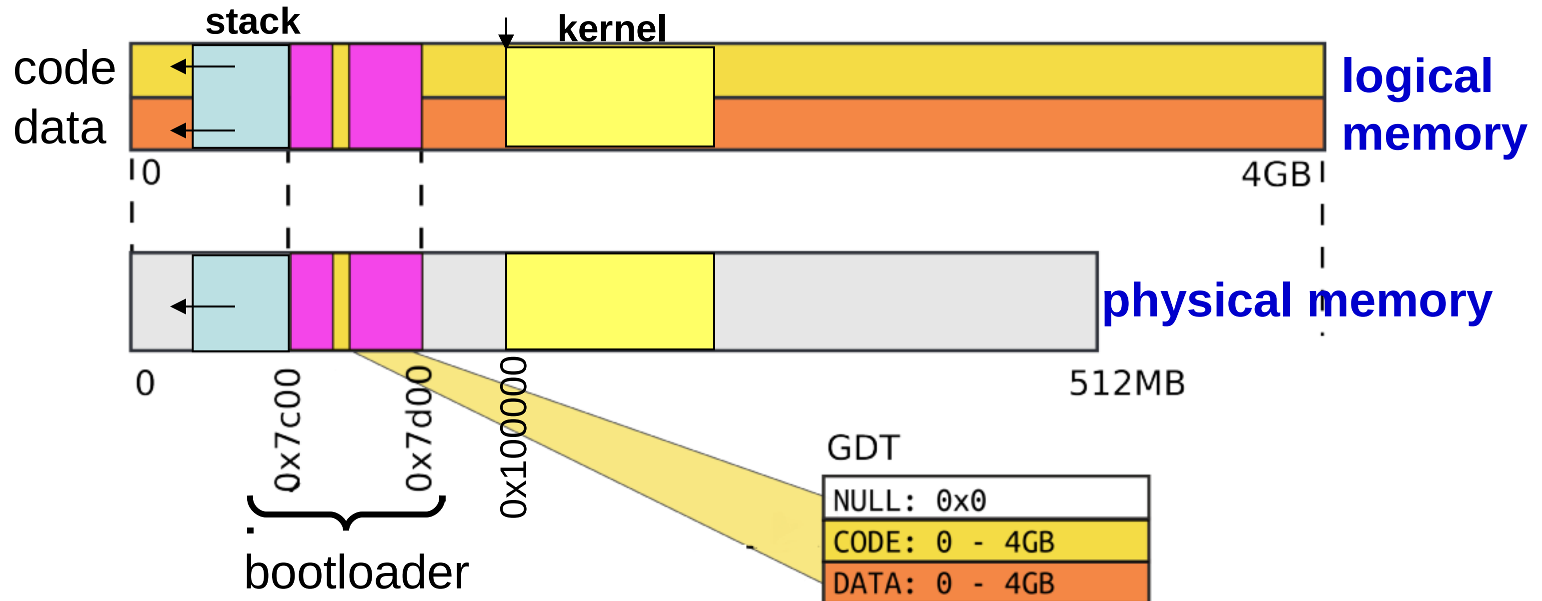


Memory when kernel is invoked
(just after the bootloader)

- Segmentation enabled but no paging

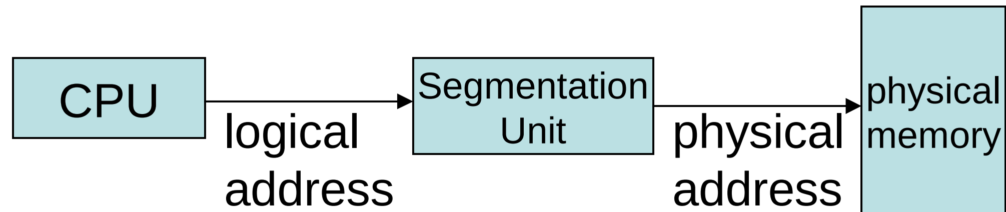


- Memory map



Slide taken from Anton Burtsev, Univ. of Utah

Memory Management Analysis



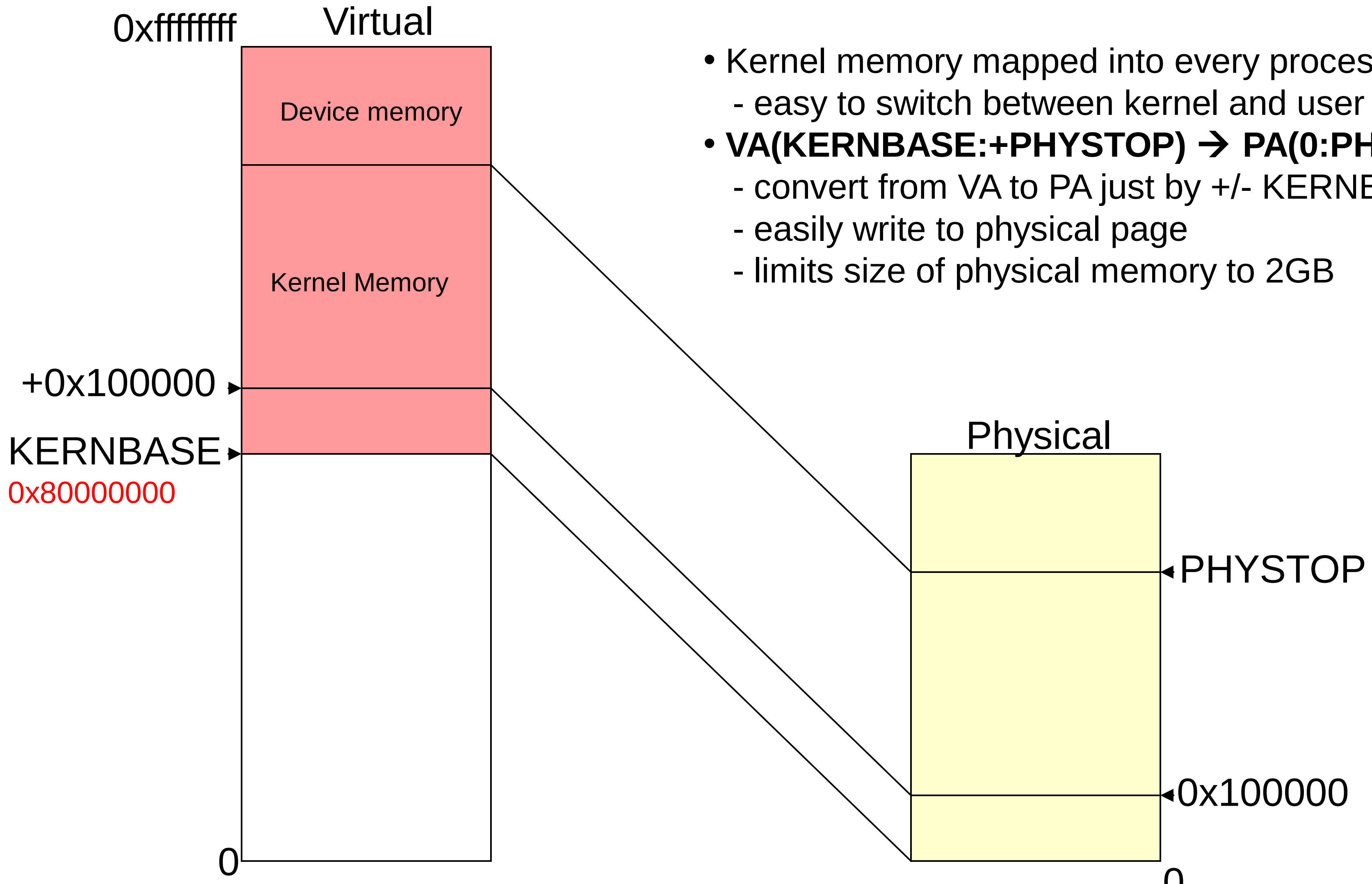
- Advantages
  - Got the kernel into protected mode (32 bit code) with minimum trouble
- Disadvantages
  - Protection of kernel memory from user writes
  - Protection between user processes
  - User space restricted by physical memory
- The plan ahead
  - Need to get paging up and running

OS code are not Relocatable

- 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_WO(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

ref : memlayout.h (0200)

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 V2P, but without casts
```

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

Early Kernel Paging Initialization

- Kernel entry point : _start (*1036*)

Turn on Page size extension

```
1042    movl    %cr4, %eax
1043    orl     $(CR4_PSE), %eax
1044    movl    %eax, %cr4
```

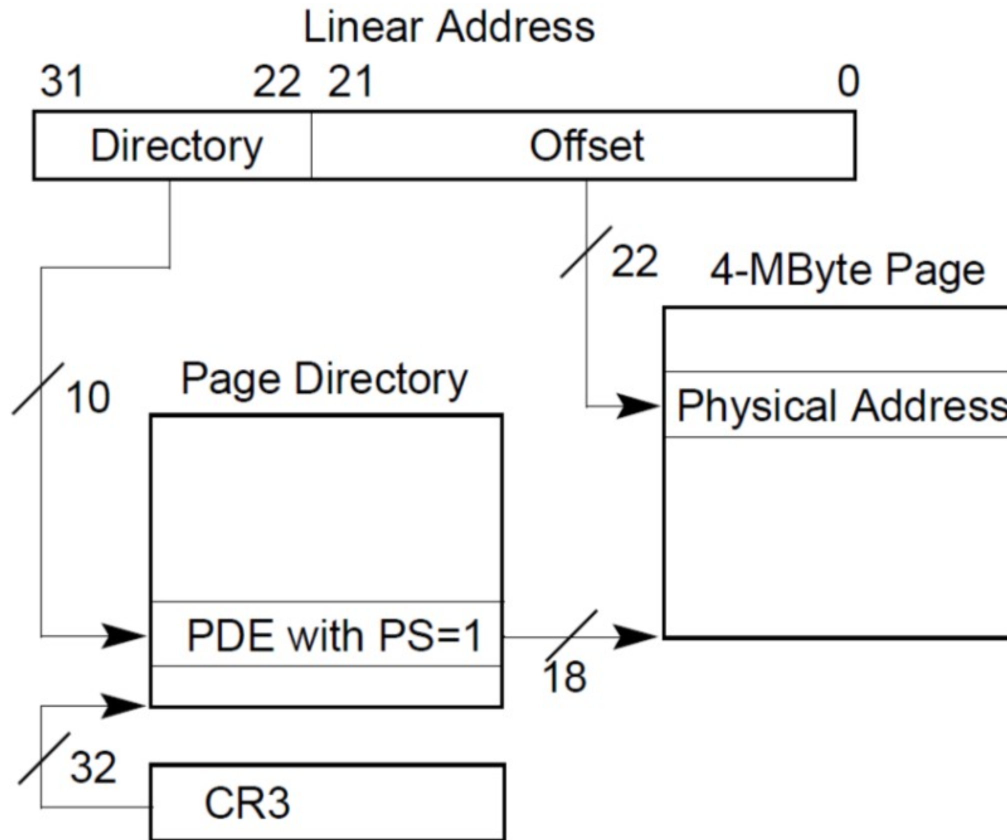
Why 4MB pages? Simplicity
(We just want 2 pages)

Set Page Directory

```
1046    movl    $(V2P_WO(entrypgdir)), %eax
1047    movl    %eax, %cr3
```

```
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 };
```

4MB Pages



Kernel memory setup

- First setup two 4MB pages
  - Entry 0:

    Virtual addresses 0 to 0x04000000 →Physical addresses 0 to 4MB
  - Entry 512:

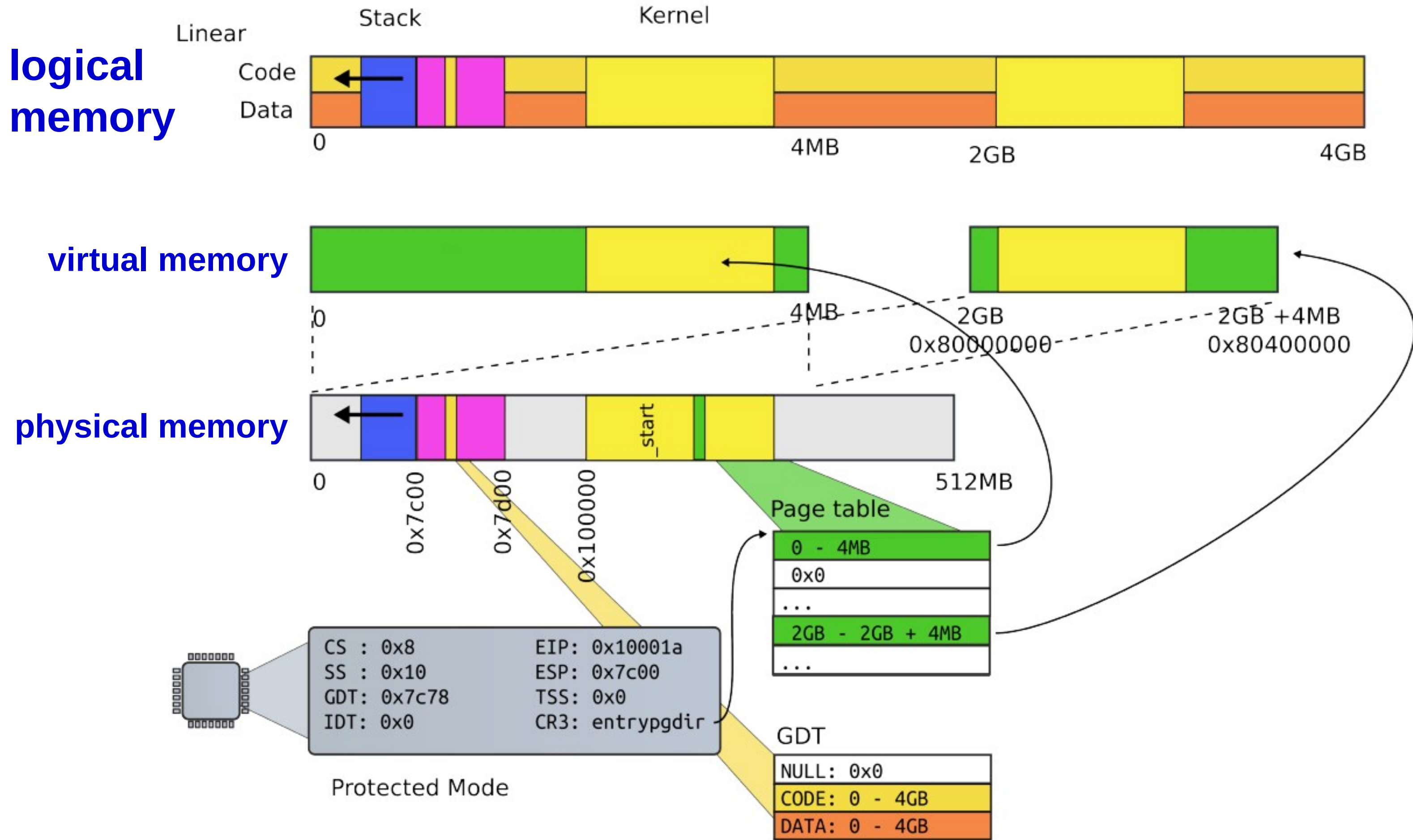
    Virtual addresses 0x80000000 to 0x84000000 →

    Physical addresses 0 to 4MB

Why do we need to map this twice?

CR

First Page Table



courtesy Anton Burtsev, Univ. of Utah

Enable Paging

- Entry point : _start (*1036*)

Turn on Page size extension

Set Page Directory

Enable Paging

```
1048    # Turn on paging.
1049    movl      %cr0, %eax
1050    orl       $(CR0_PG|CR0_WP), %eax
1051    movl      %eax, %cr0
```

Stack setup

- Entry point : _start (*1036*)

Turn on Page size extension

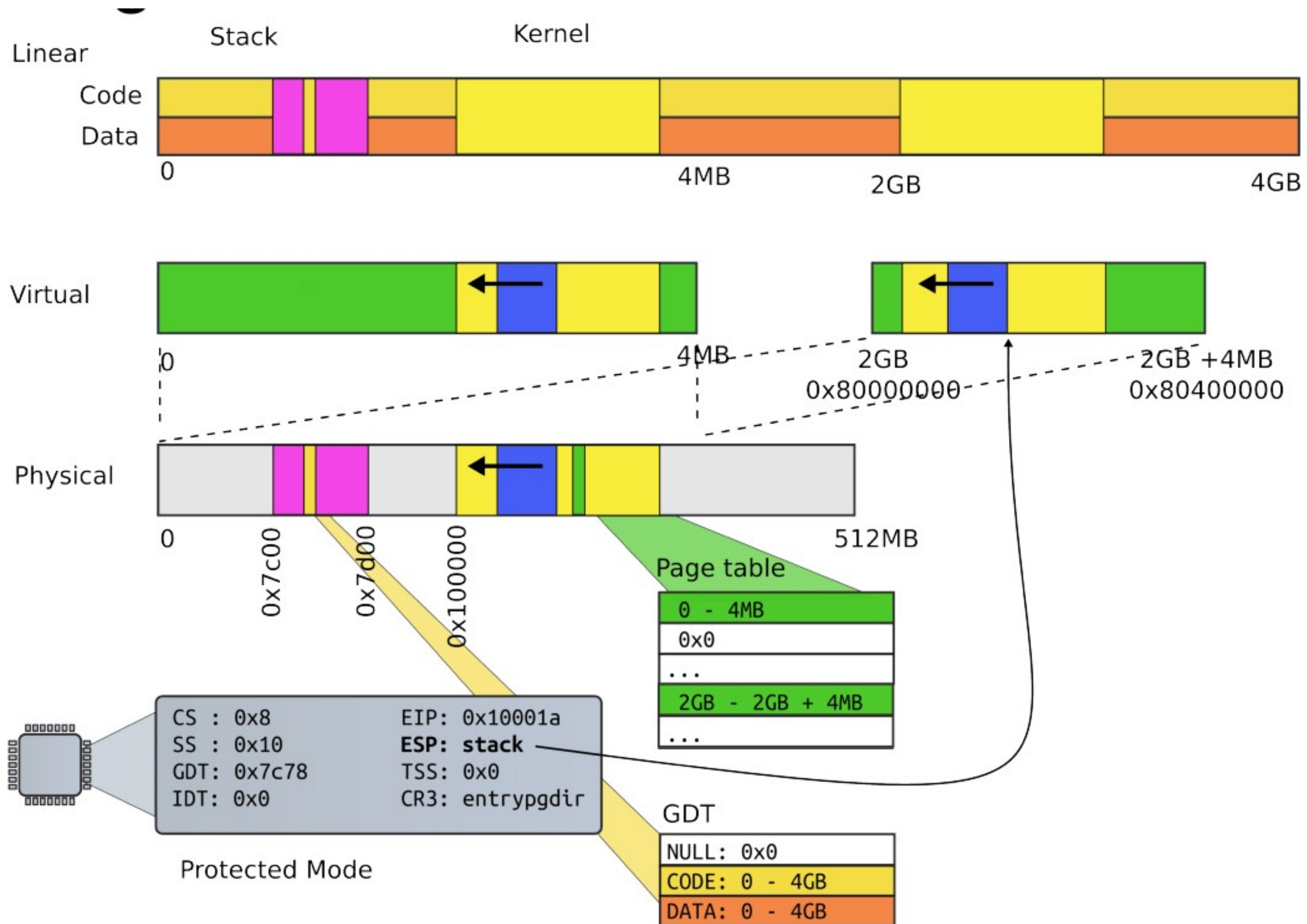
Set Page Directory

Enable Paging

Stack setup

```
1053    # Set up the stack pointer.
1054    movl $(stack + KSTACKSIZE), %esp
```

Stack



courtesy Anton Burtsev, Univ. of Utah

Execute main

- entry point : _start (*1036*)

Turn on Page size extension

Set Page Directory

Enable Paging

Stack setup

Jump to main

```
1053    # Set up the stack pointer.
1054    movl $(stack + KSTACKSIZE), %esp
```

```
1060    mov $main, %eax
1061    jmp *%eax
```

New Address Scheme Analysis

Scheme : enable paging with 2 pages of 4MB each

- Advantages,
  - Useful for initializing the rest of memory
    - (issues with kmalloc …. later!!!)
- Disadvantages
  - Kernel mapped twice, reducing user space area
  - Only 4MB of physical memory is mapped. Remaining is unutilized

xv6 next goes into the final addressing scheme

(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

CR

Virtual Address Space

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
};
```

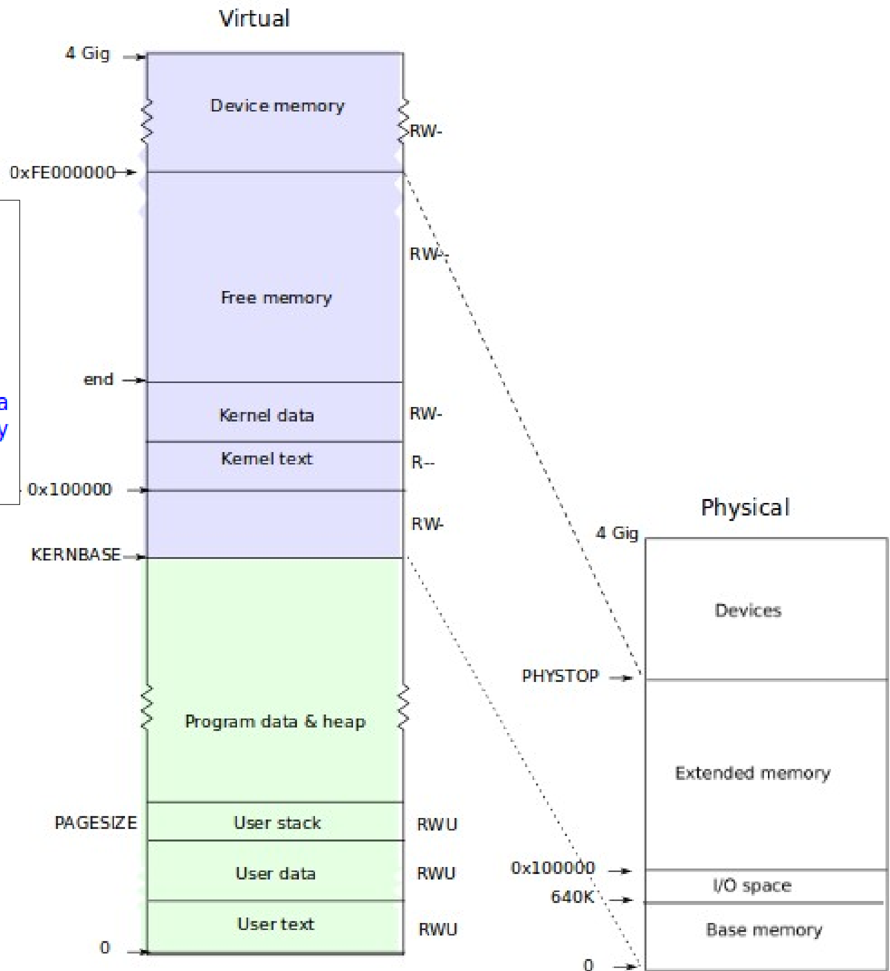
Setting Up kernel pages (vm.c)

1. stuct kmap

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

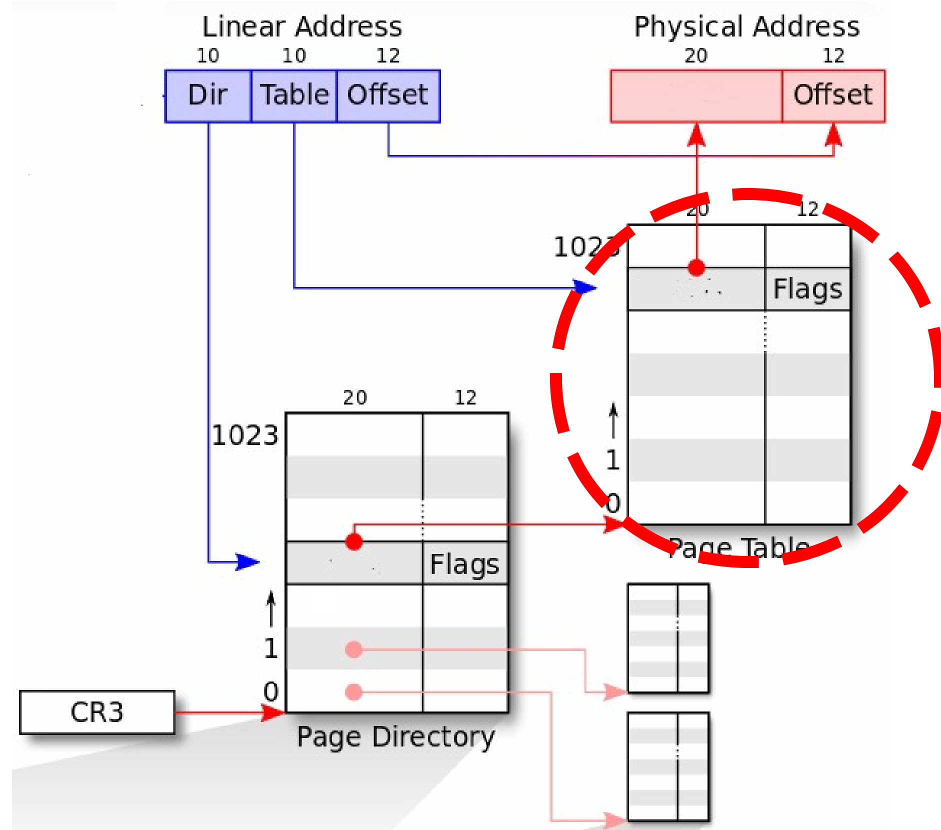
2. Kernel page tables set up in **kvmalloc()** (*1857*)

   (invoked from main)



mappages (1779)

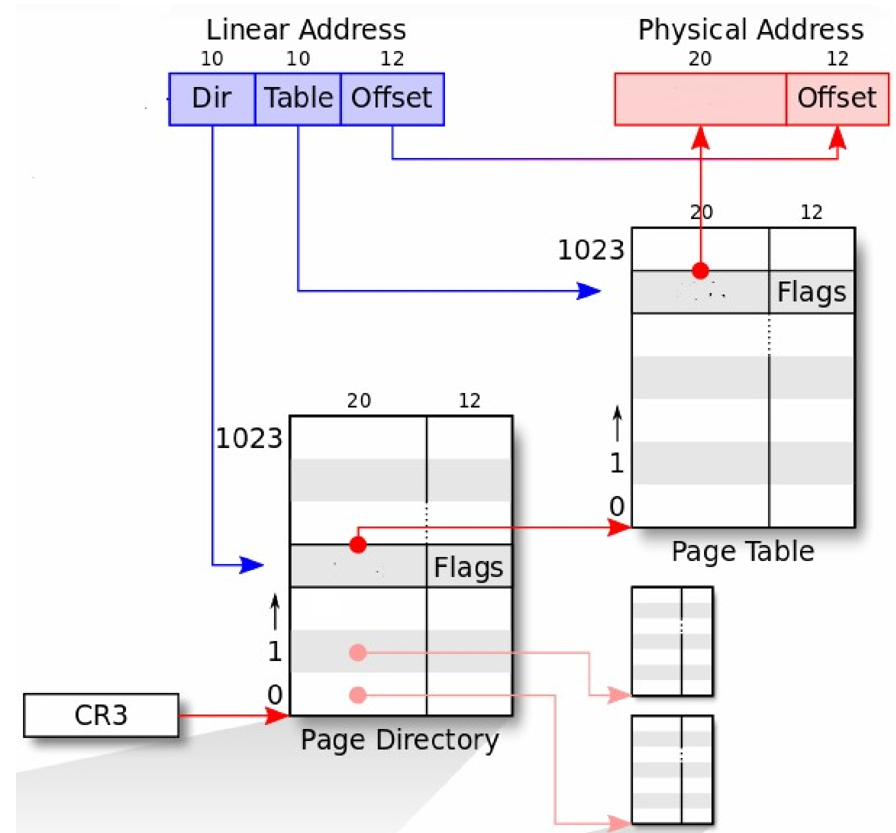
- Fill page table entries mapping virtual addresses to physical addresses
- Which page table entry?
  - obtained from walkpgdir
- What are the contents?
  - Physical address
  - Permissions
  - present



walkpgdir (1754)

- Create a page table 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

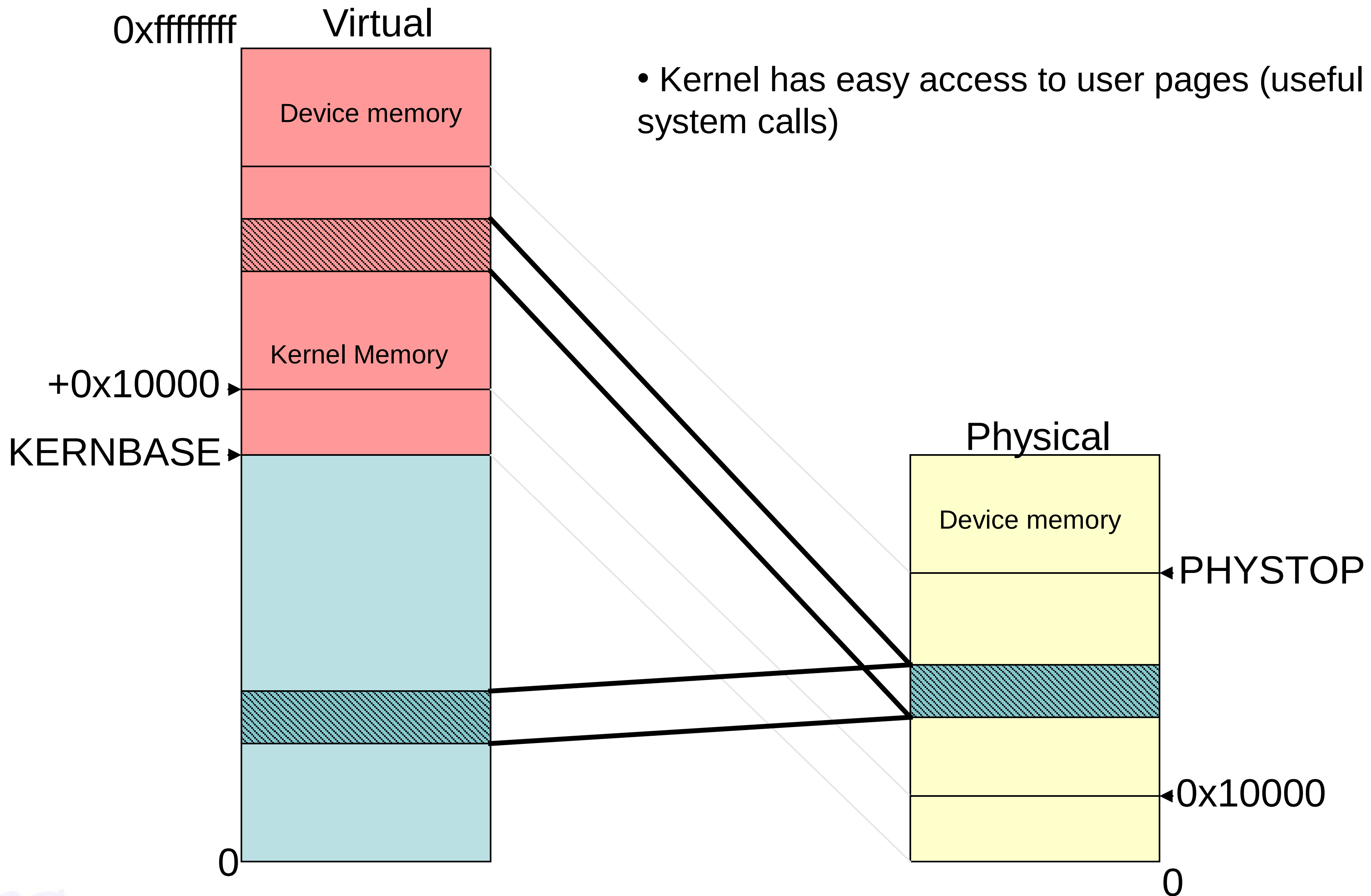


Using Page Tables (in OS)

- Functions available
  - mappages (*1779*) : create page table entries mapping virtual addresses to physical addresses
  - copyuvm (*2053*): copy a process's page table into another
  - walkpgdir *(1754)* : return page table entry corresponding to a virtual address

CR

User Pages mapped twice



- Kernel has easy access to user pages (useful for system calls)

(Re)Initializing Segmentation

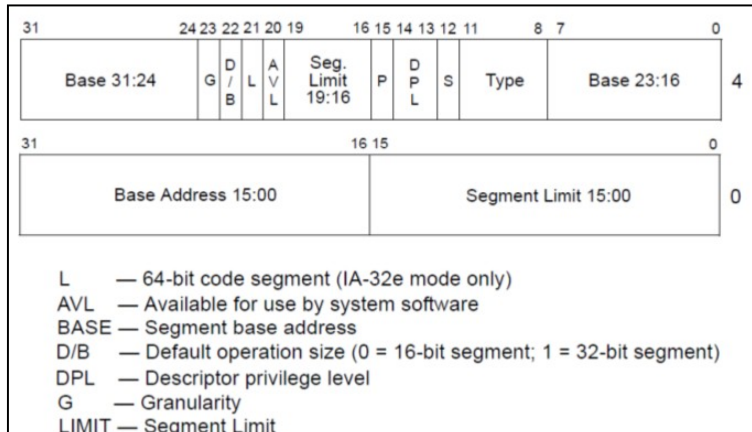
- Segments
  - Kernel code
  - Kernel data
  - User code
  - User data
  - Per CPU data

```
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);
```

```
c->gdt[SEG_KCPU] = SEG(STA_W, &c->cpu, 8, 0);
```

ref : seginit (1716)

Segment Descriptor in xv6




```c
// 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 }
```

```c
// 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)

Loading the GDTR

- Instruction LGDT
- Each CPU has its own GDTR

```
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);

// Map cpu, and curproc
c->gdt[SEG_KCPU] = SEG(STA_W, &c->cpu, 8, 0);

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

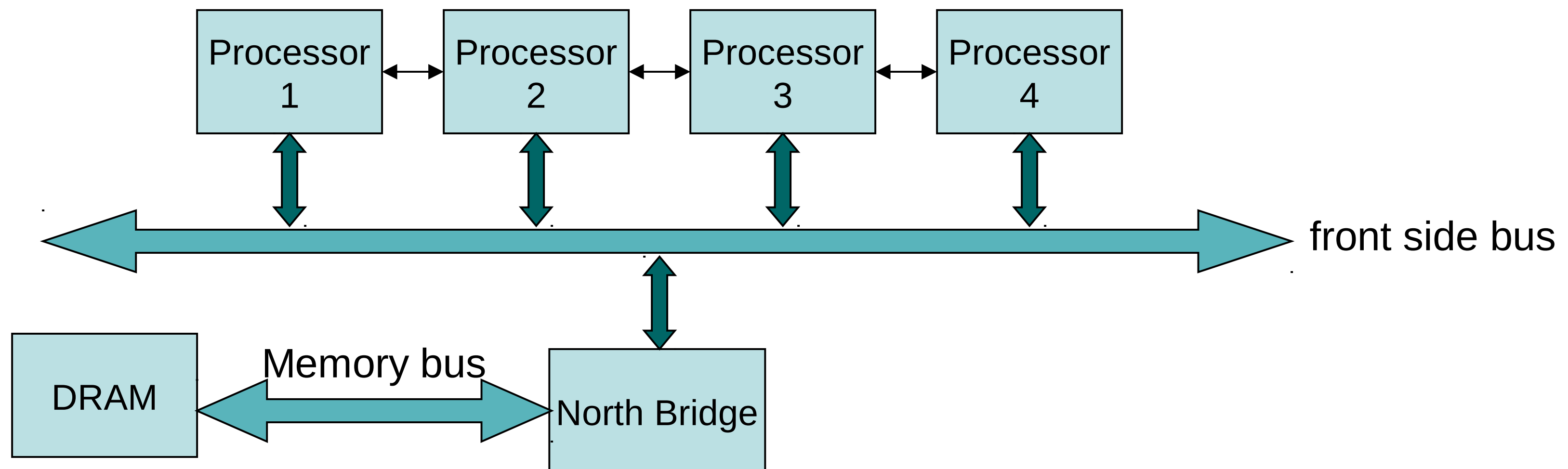
```
static inline void
lgdt(struct segdesc *p, int size)
{
  volatile ushort pd[3];

  pd[0] = size-1;
  pd[1] = (uint)p;
  pd[2] = (uint)p >> 16;

  asm volatile("lgdt (%0)" : : "r" (pd));
}
```

ref : x86.h

Per CPU Data

CR

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.

CR

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 per-cpu data
  - **slow (because locking can be tedious)!!!**

ref : proc.h [23]

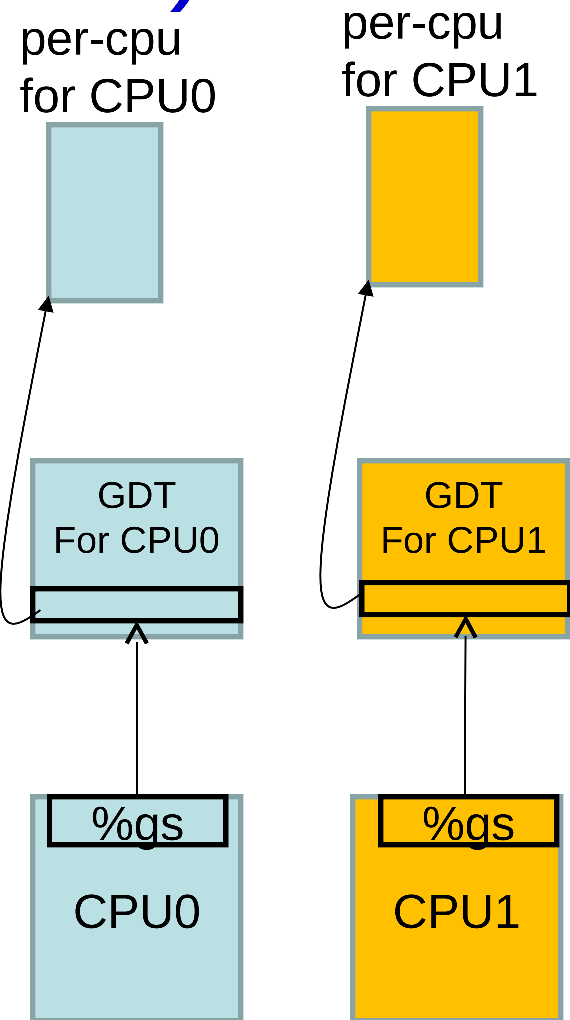
Alternate Solution
(using CPU registers)

- 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 ☹

Content borrowed from Carmi Merimovich (http://www2.mta.ac.il/~carmi/)

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



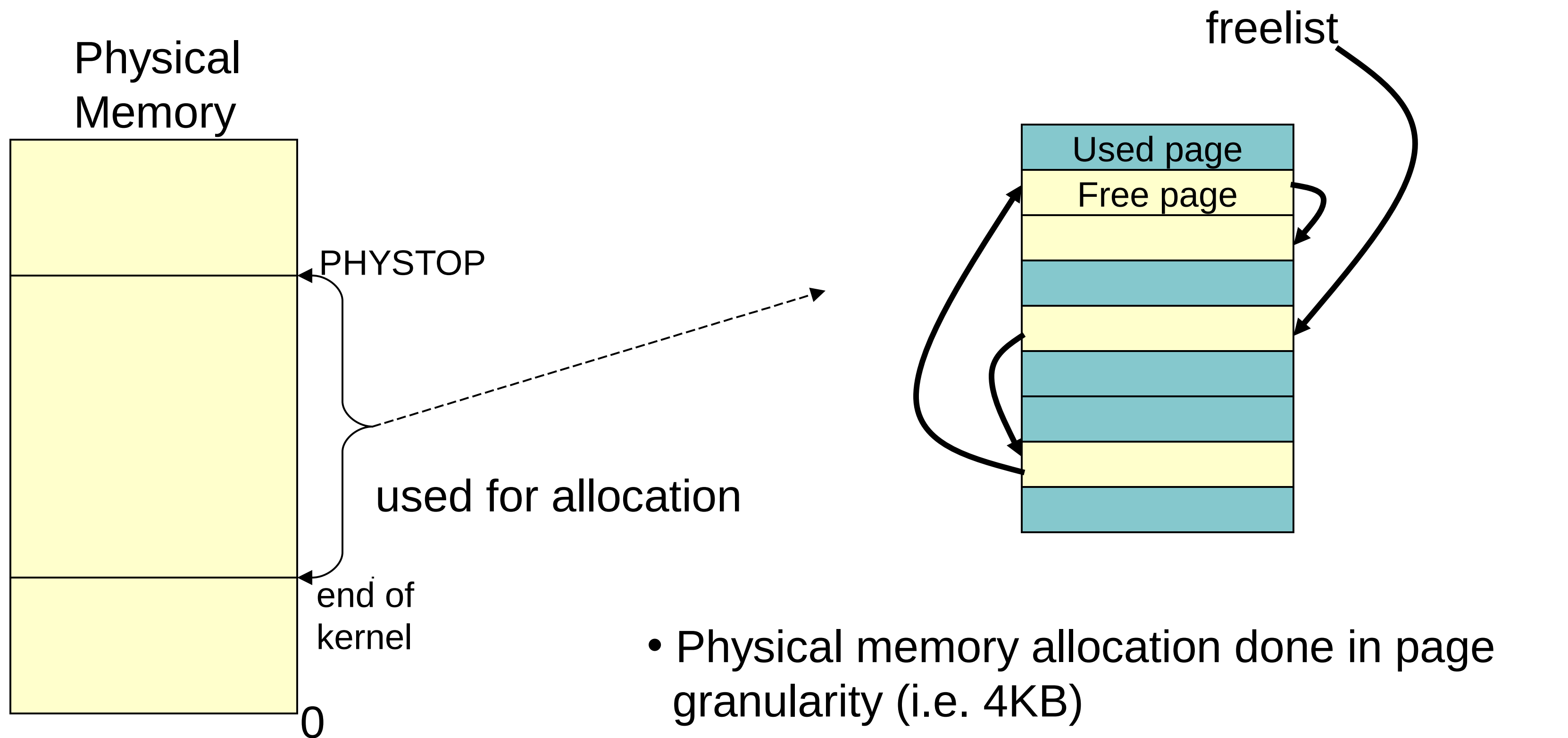
Using %gs

- 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.

Allocating Memory

CR

Allocating Pages (kalloc)

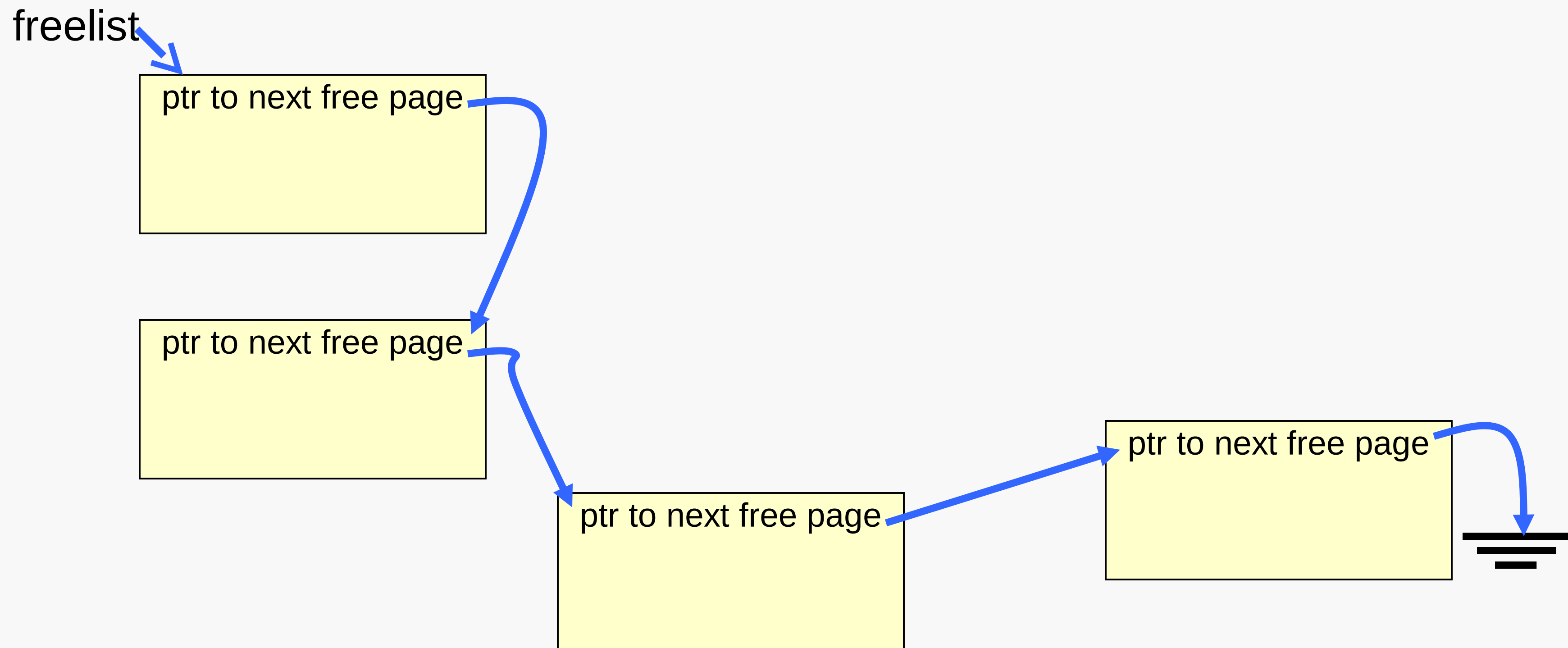


- Physical memory allocation done in page granularity (i.e. 4KB)
- Free physical pages in a list
- Page Allocation removes from list head (see function kalloc)
- Page free adds to list head (see kfree)

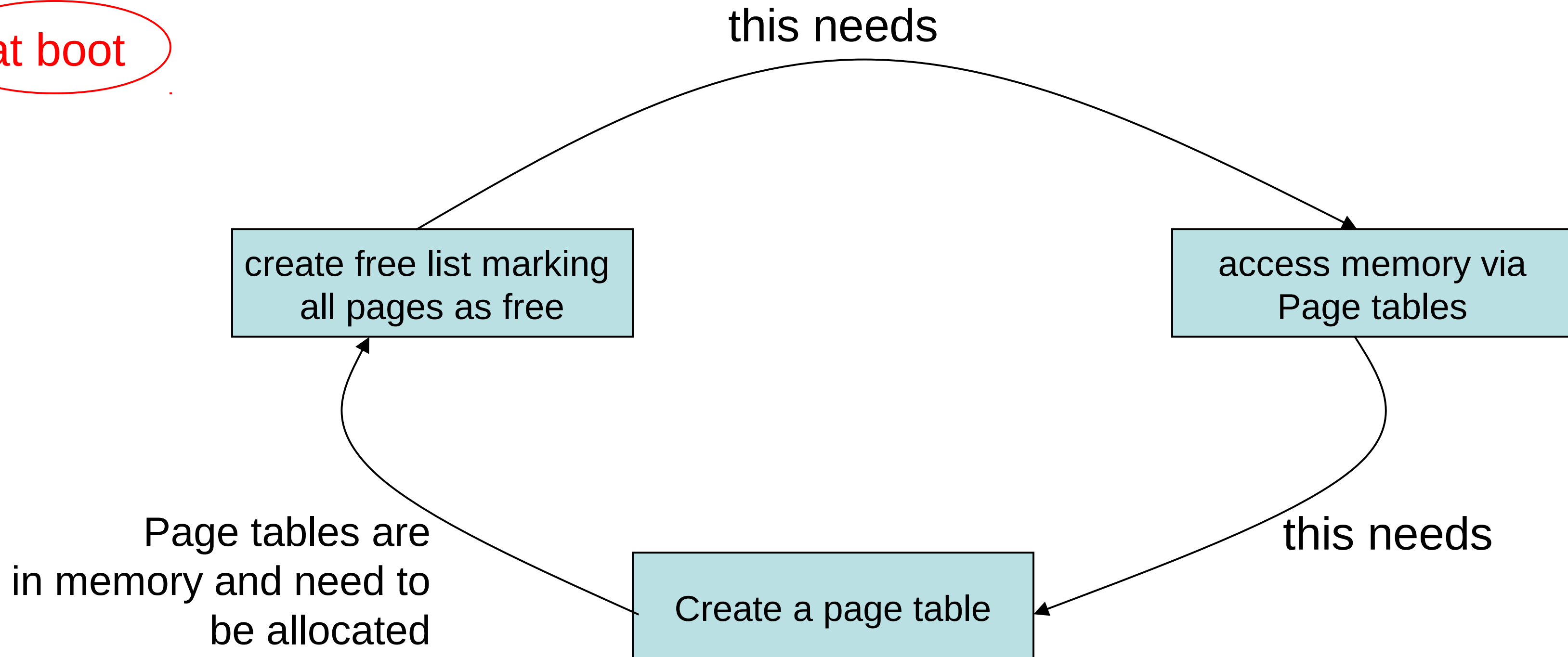
ref : kalloc.c [30]

Freelist Implementation

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



Initializing the list (chicken & egg problem)



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)

For next class…

- revise / learn interrupt handling in x86 processors

CR