SOS Internals

1 Simple Operating System

- SOS is a **S**imple **O**perating **S**ystem designed for the 32-bit x86 architecture

- Its purpose is to understand basic concepts of operating system design
  - how is an OS loaded?
  - how are programs run?
  - how is memory managed?
  - …

IA-32 x86 Registers

- General purpose: 32-bit EAX, EBX, ECX, EDX, ESI, EDI, EBP and ESP
- Segment registers: 16-bit CS, DS, SS, ES, FS and GS
- Instruction pointer: 32-bit EIP
- Program status: 32-bit EFLAGS
- And many others…

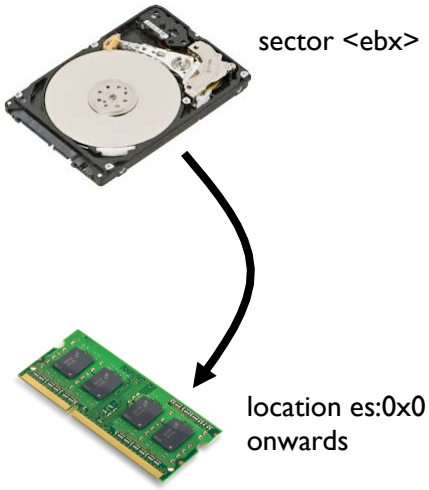
BIOS Routines

Display a character

```
sub %bh, %bh      # Page 0
mov $0x0e, %ah    # Teletype output service
mov $0x41, %al    # the character A
int $0x10
```

Read sector(s) from disk

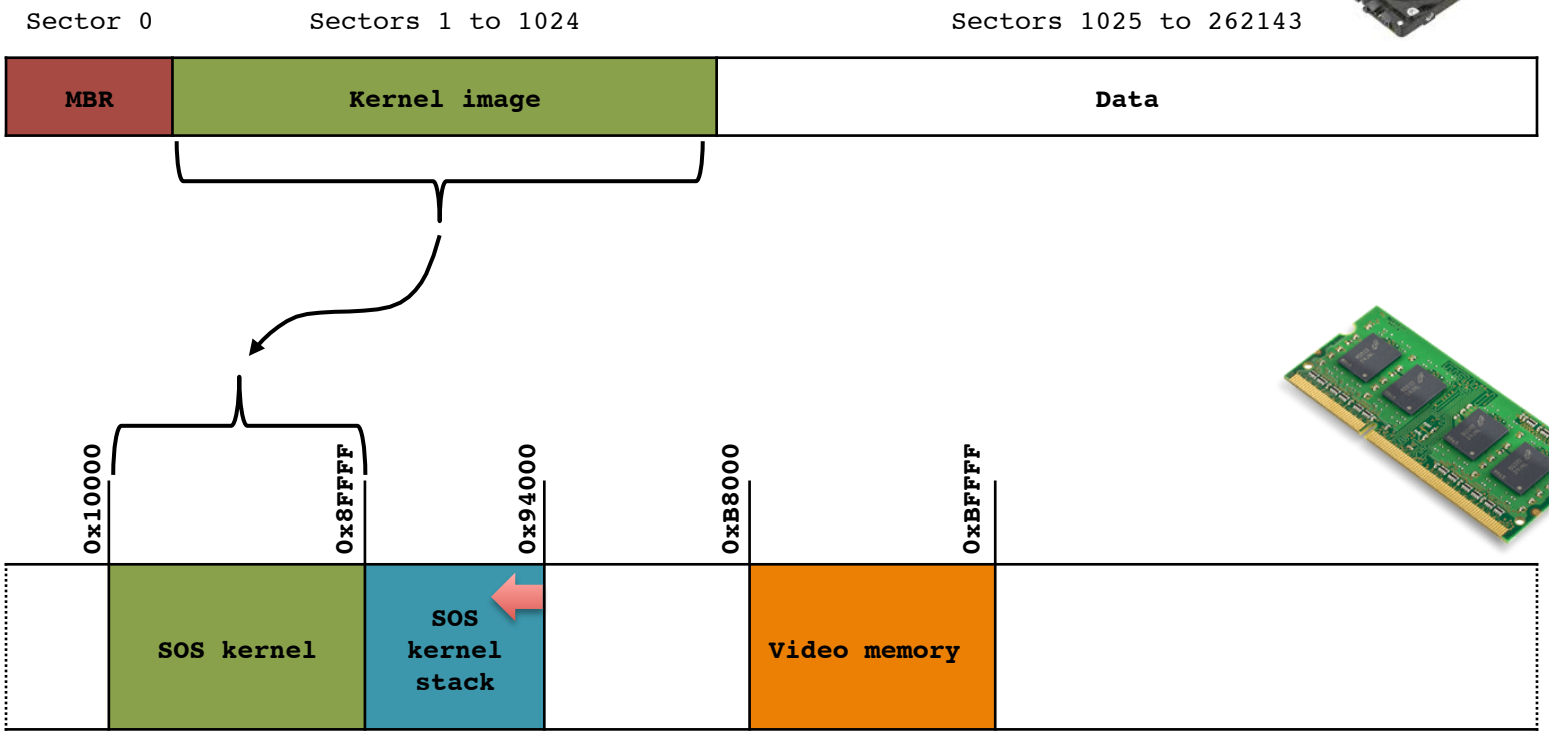
```
sub %ax, %ax
push %ax          # LBA sector number [48:63]
push %ax          # LBA sector number [32:47]
push %ebx         # LBA sector number [0:31]
push %es          # Buffer segment
push %ax          # Buffer offset (always 0)
push $1           # Number of sectors to read
push $16          # Packet size
mov $0x42, %ah    # Extended read disk service
mov %sp, %si      # DS:SI -> packet
int $0x13
```



Real Mode Addressing

```
0xF000:0xFFFF = 0xFFFFF

   0xF000<<4      1111 0000 0000 0000 0000
 + 0xFFFF              1111 1111 1111 1111
   ------------------------------------
   0xFFFFF        1111 1111 1111 1111 1111

0xF800:0x7FFF = 0xFFFFF

0xF800:0x8000 = 0x00000 (20 bit address bus)
```

SOS on Disk and Memory

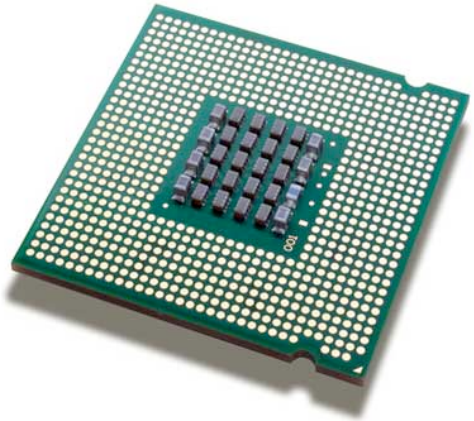


Master Boot Record

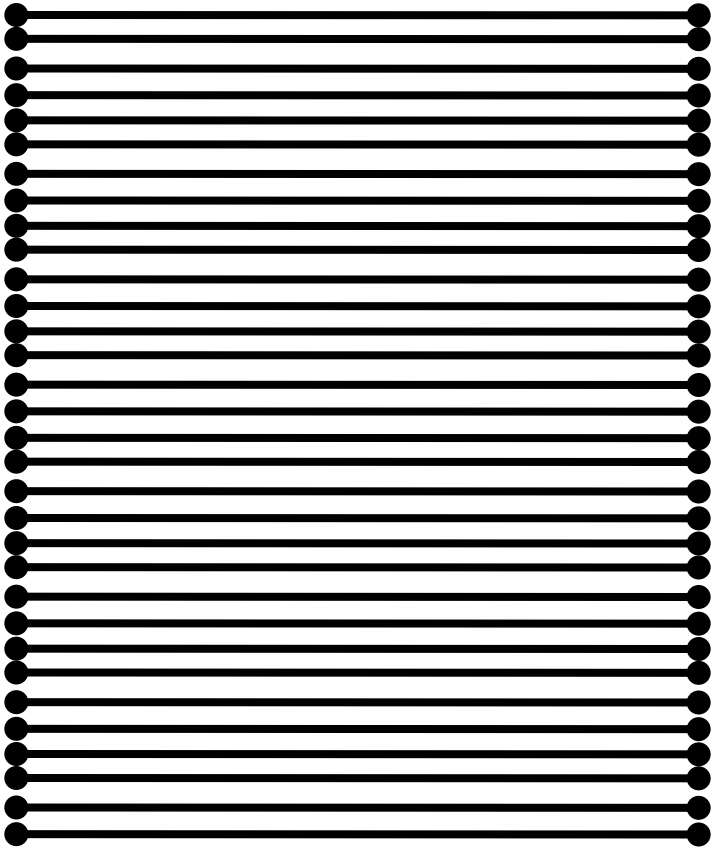
Byte	Content	Size
0 to 445	Code	446 bytes
446 to 461	Partition table entry 1	16 bytes
462 to 477	Partition table entry 2	16 bytes
478 to 493	Partition table entry 3	16 bytes
494 to 509	Partition table entry 4	16 bytes
510	0x55	1 byte
511	0xAA	1 byte

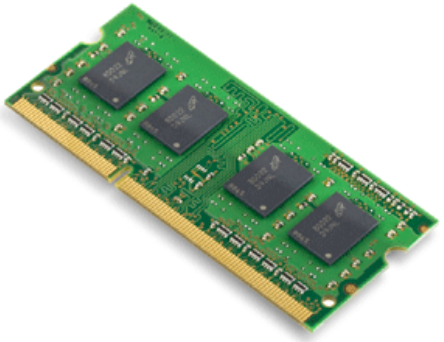
446 bytes of code to load the SOS kernel from disk to memory															
														80	00
00	00	**10**	00	00	00	00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00	00	00	00	00	**55**	**AA**

A20 Line



32-bit address bus



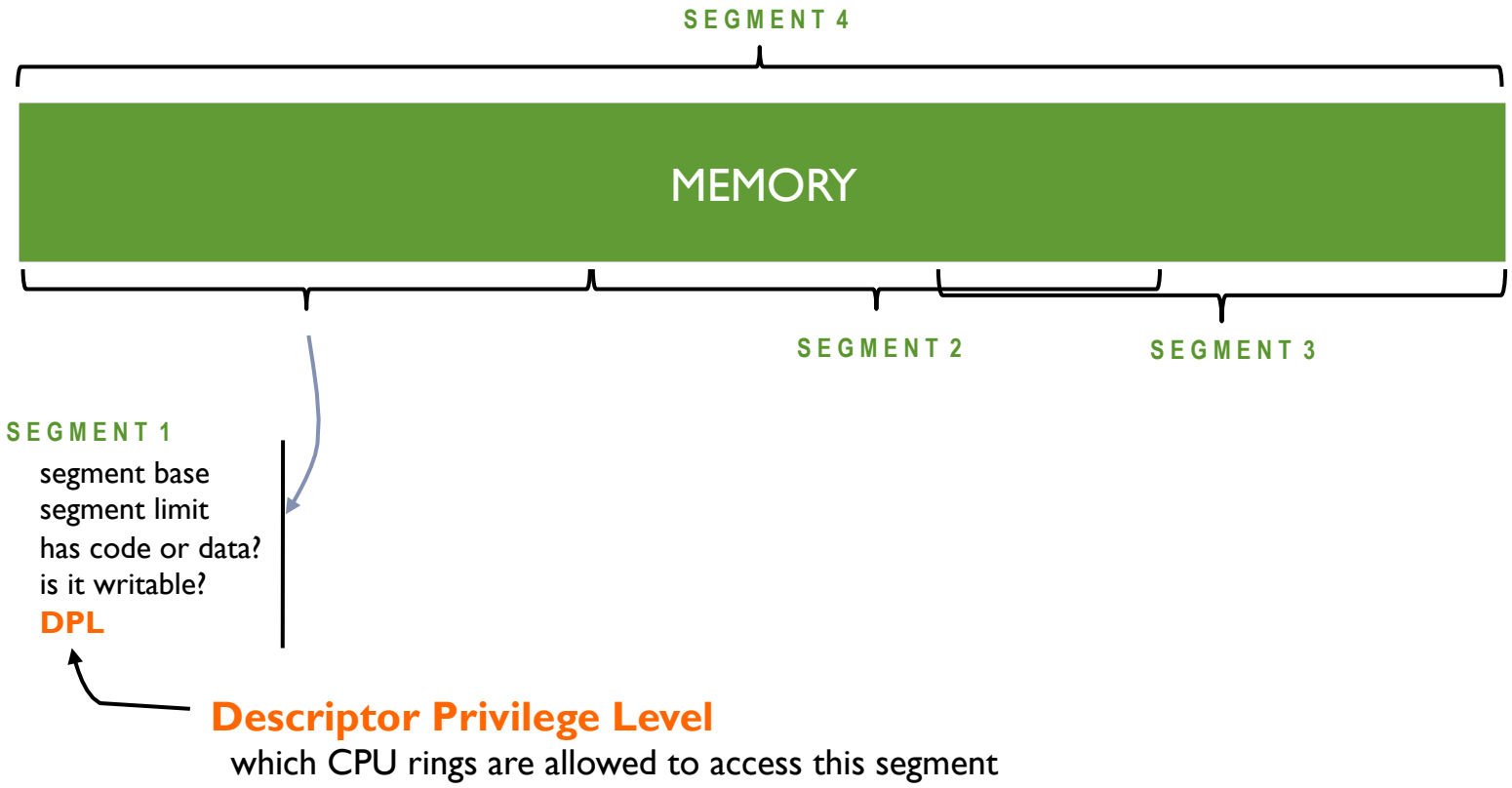


`0xF800:0x8000 = 0x100000`

(could become problematic for old programs)

Memory Segmentation

- Divide memory into segments (may overlap)



Global Descriptor Table (GDT)

- An array of entries
- Each entry describes one memory segment
- Each entry is 64 bits in size
  - segment base - 32 bits
  - segment limit - 20 bits
  - access byte - 8 bits
  - flag - 4 bits

Global Descriptor Table (GDT)

entry 0	0x0000000000000000	not used
entry 1	0x00cf9a000000ffff	base=0, limit=4GB, code, DPL=0
entry 2	0x00cf92000000ffff	base=0, limit=4GB, data, DPL=0

0x00cf9a000000ffff

0000 0000 1100 1111 1001 1010 0000 0000 0000 0000 0000 0000 1111 1111 1111 1111

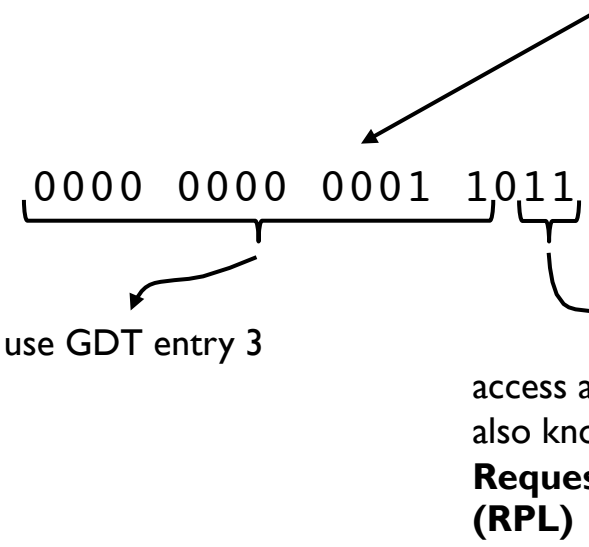
Descriptor Privilege Level (DPL)

```
typedef struct {
    uint16_t limit_0_15;
    uint16_t base_0_15;
    uint8_t  base_16_23;
    uint8_t  access_byte;
    uint8_t  limit_and_flag;
    uint8_t  base_24_31;
} __attribute__ ((packed)) GDT_DESCRIPTOR;
```

32-bit Protected Mode Addressing

16-bit Segment Selector	. .	32-bit Offset

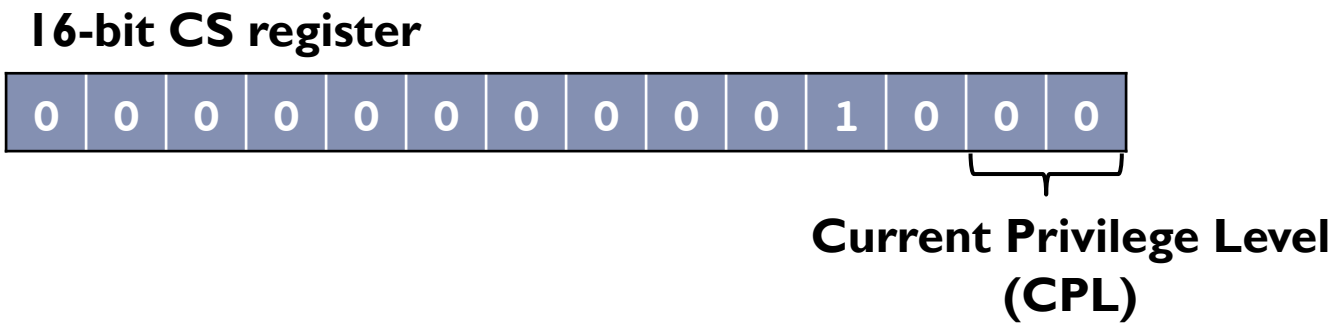
0x001B : 0x100 =



if access is granted,

segment base stored in GDT entry 3
+ 0x100

When is Memory Access Granted?



- **Offset** ≤ GDT entry **segment limit**
- **max(CPL, RPL)** ≤ **DPL**

Enabling Protected Mode

- Set up GDT somewhere in memory
- Use LGDT instruction to notify CPU about the location of the GDT in memory
- Disable interrupts
- Enable bit 0 in the CR0 register

- In protected mode
  - instructions are fetched from CS:EIP
  - the stack is at SS:ESP

The main() Function

- main() is called after protected mode is enabled

```
#include "kernel_only.h"

int main(void) {
        init_disk();
        init_display();
        init_interrupts();
        init_keyboard();

        start_console();

        return 0;
}
```

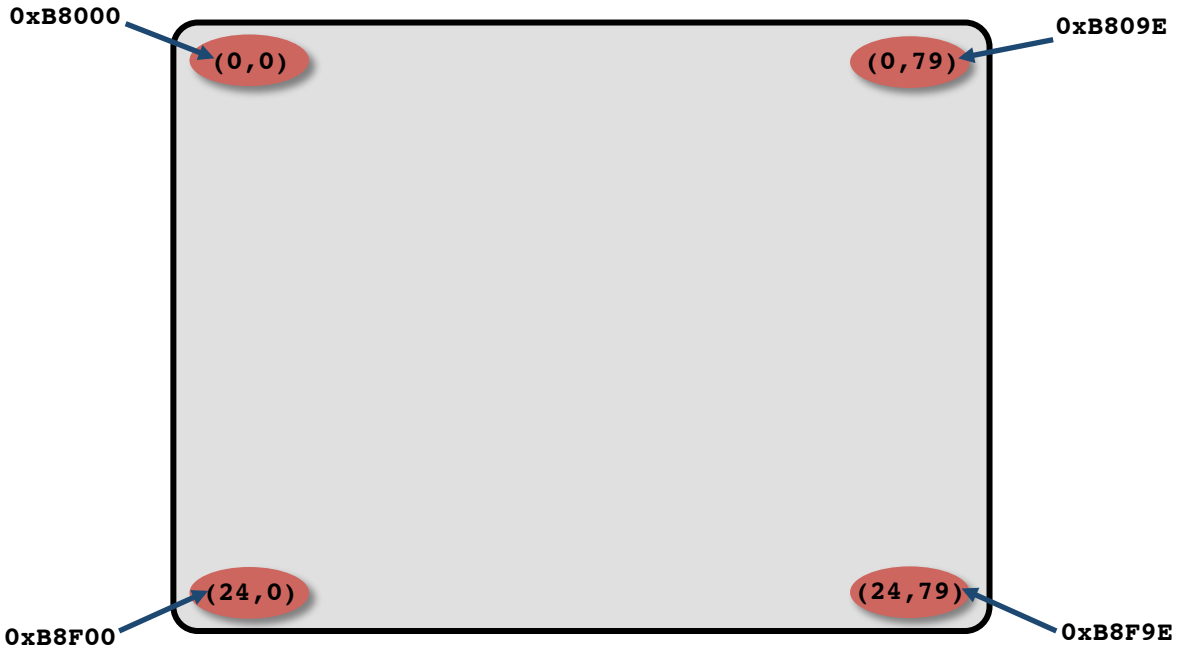
Disk Initialization

- `init_disk` in `disk.c`
- Queries hard drive for size
  - uses port I/O

- `uint8_t read_disk(uint32_t LBA, uint8_t n_sectors, uint8_t *buffer);`
  - reads sectors LBA, LBA+1, ..., LBA+n_sectors-1 from the disk and copies into buffer

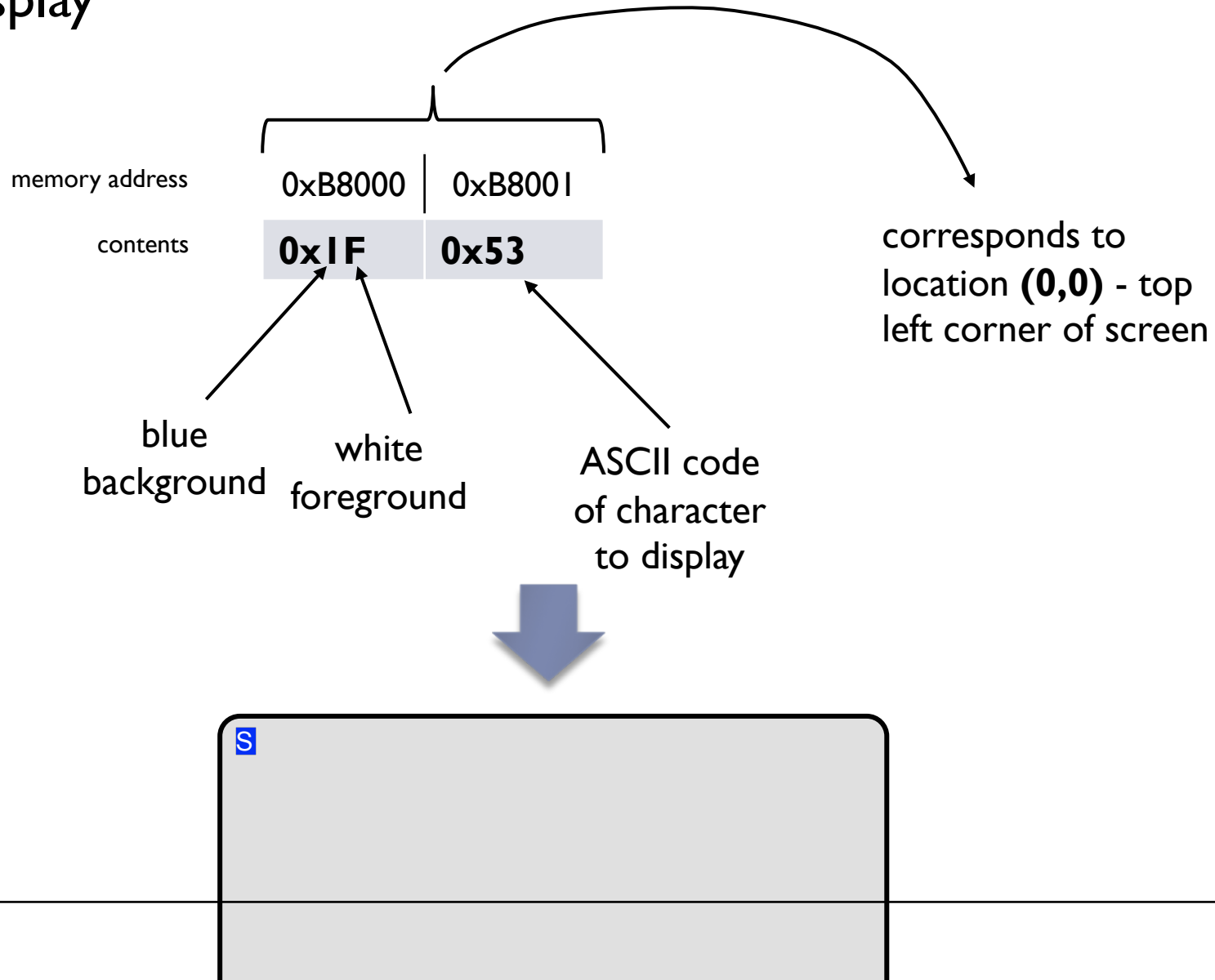
Display Initialization

- CGA text mode: 80 columns by 25 rows
- Memory range 0xB8000 to 0xBFFFF mapped to CGA display
  - 2 bytes per location



Writing to Display

- Writing to video memory is equivalent to writing to display



sys_printf

- Function wrapper that allows kernel to write to display
  - `void sys_printf(const char *, ...);`
- We will need a system call so that user programs can indirectly access this function

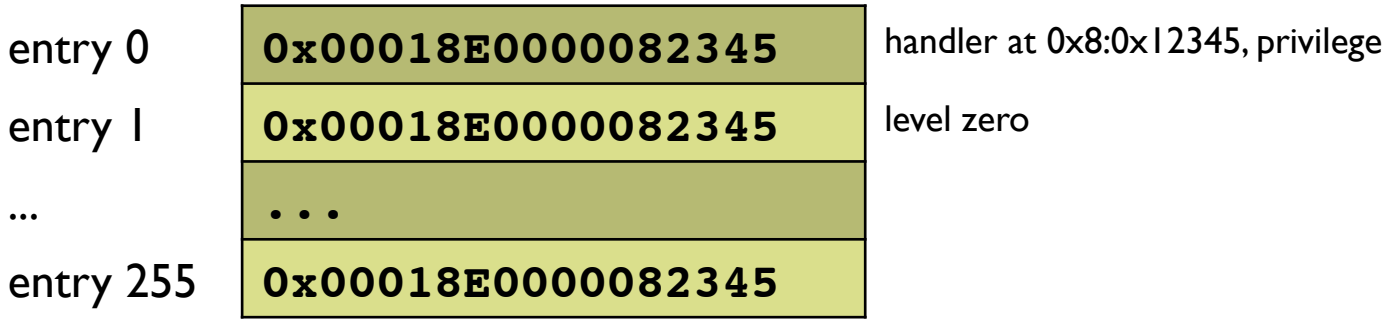
Interrupt Handlers

- 256 interrupts that we can program
- Interrupts 0 to 31 are fired on special events
  - interrupt 0: division by zero
  - interrupt 6: invalid opcode
- Interrupts 32 to 255 can be used by us to transfer control to OS
  - in system calls (using the INT instruction)
  - when user presses a key (using the PIC)
- Handlers are defined in the **Interrupt Descriptor Table (IDT)**

Interrupt Descriptor Table (IDT)

- An array of 256 entries
- Each entry describes the handler for one interrupt
- Each entry is 64 bits in size
  - protected mode address of handler function - 48 bits
  - flag - 8 bits
    - which privilege level can invoke interrupt using INT
  - reserved - 8 bits (always zero)

Interrupt Descriptor Table (IDT)




```
/*** An IDT entry ***/
typedef struct {
    uint16_t handler_lo;
    uint16_t sel;
    uint8_t  reserved;
    uint8_t  flags;
    uint16_t handler_hi;
} __attribute__ ((packed)) IDT_DESCRIPTOR;
```

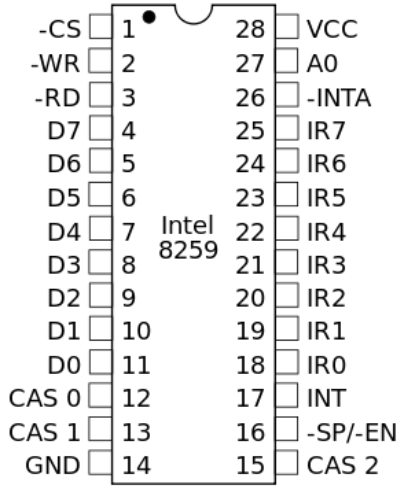
A Default Interrupt Handler

```
asm ("handler_default_entry: cli\n"
    "pushl %ds\n"
    "pushl %es\n"
    "pushl %fs\n"
    "pushl %gs\n"
    "pushal\n"
    "call default_interrupt_handler\n"
    "popal\n"
    "popl %gs\n"
    "popl %fs\n"
    "popl %es\n"
    "popl %ds\n"
    "sti\n"
    "iretl\n"
);

void default_interrupt_handler() {
    puts("Unhandled interrupt!");
}
```

```
IDT_DESCRIPTOR IDT[256];

for (i=0; i<256; i++) {
    install_interrupt_handler(i,
       handler_default_entry,
       0x0008,
       0x8E);
}
```

PIC

- PIC: Programmable Interrupt Controller
  - small piece of hardware that relays hardware interrupts to the CPU
- Intel 8259
  - 16 interrupt lines (also called IRQs)
  - some IRQ lines are hardwired
    - IRQ 0 : to the system timer
    - IRQ 1: to the keyboard controller



- Which interrupt routine will run when an IRQ fires?

PIC Configuration

- IRQ 0 → Interrupt 32
- IRQ 1→ Interrupt 33
- IRQ 2 → Interrupt 34
- ...
- IRQ 15 → Interrupt 47

```
port_write_byte (0x21, 0x20);
port_write_byte (0xA1, 0x28);
```

- For now, we will disable all IRQ lines except IRQ 1

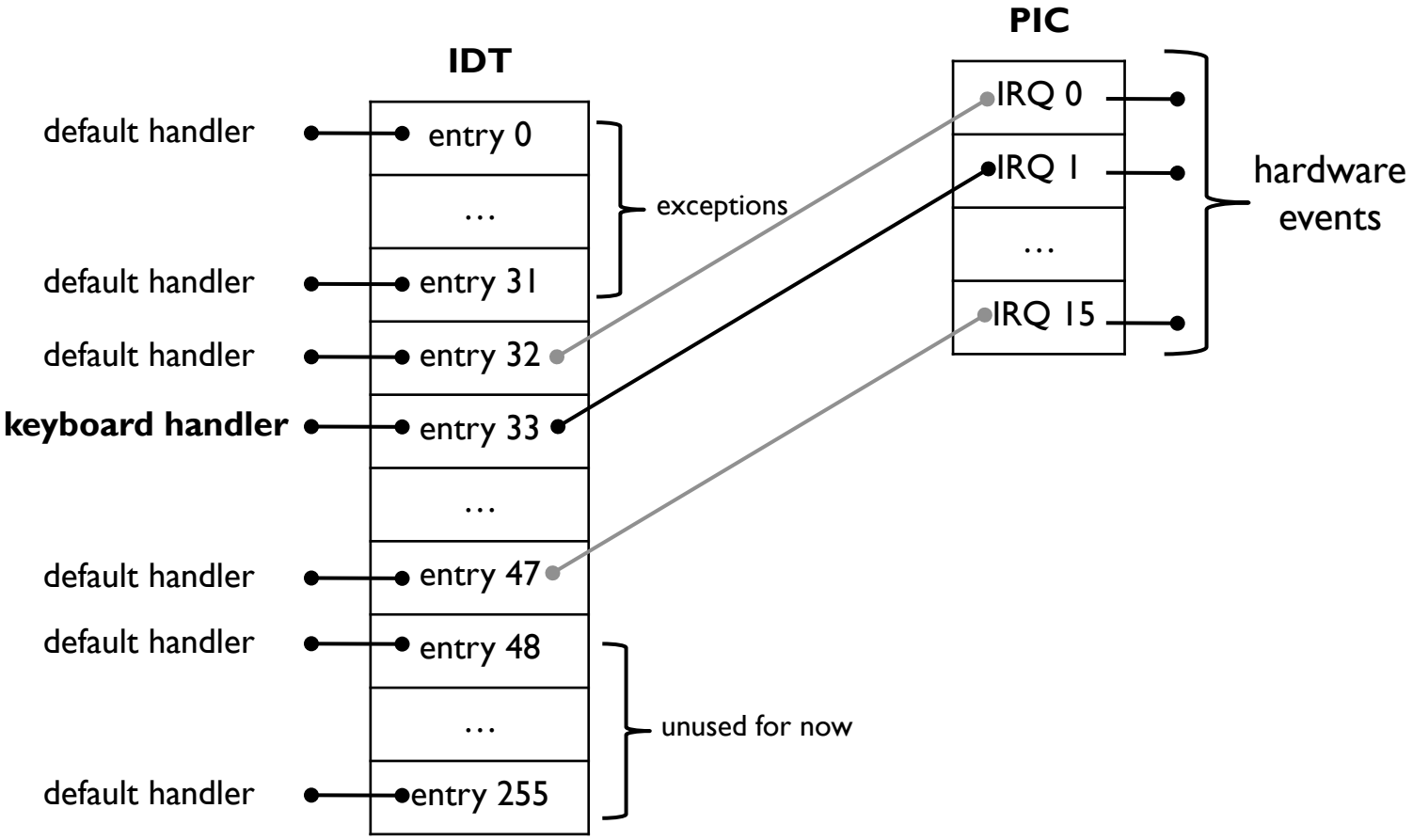
A Keyboard Handler

- IRQ 1 fired when user presses key
- IRQ 1 mapped to interrupt 33
- We need a keyboard handler for IDT[33]

```
asm("handler_keyboard_entry: cli\n"
    "pushl %ds\n"
    "pushl %es\n"
    "pushl %fs\n"
    "pushl %gs\n"
    "pushal\n"
    "call keyboard_interrupt_handler\n"
    "popal\n"
    "popl %gs\n"
    "popl %fs\n"
    "popl %es\n"
    "popl %ds\n"
    "sti\n"
    "iretl\n"
);
void keyboard_interrupt_handler() {
    current_key = KEY_UNKNOWN;
    ...
    ...
```

```
install_interrupt_handler(33,
        handler_keyboard_entry,
        0x0008,
        0x8E);
```

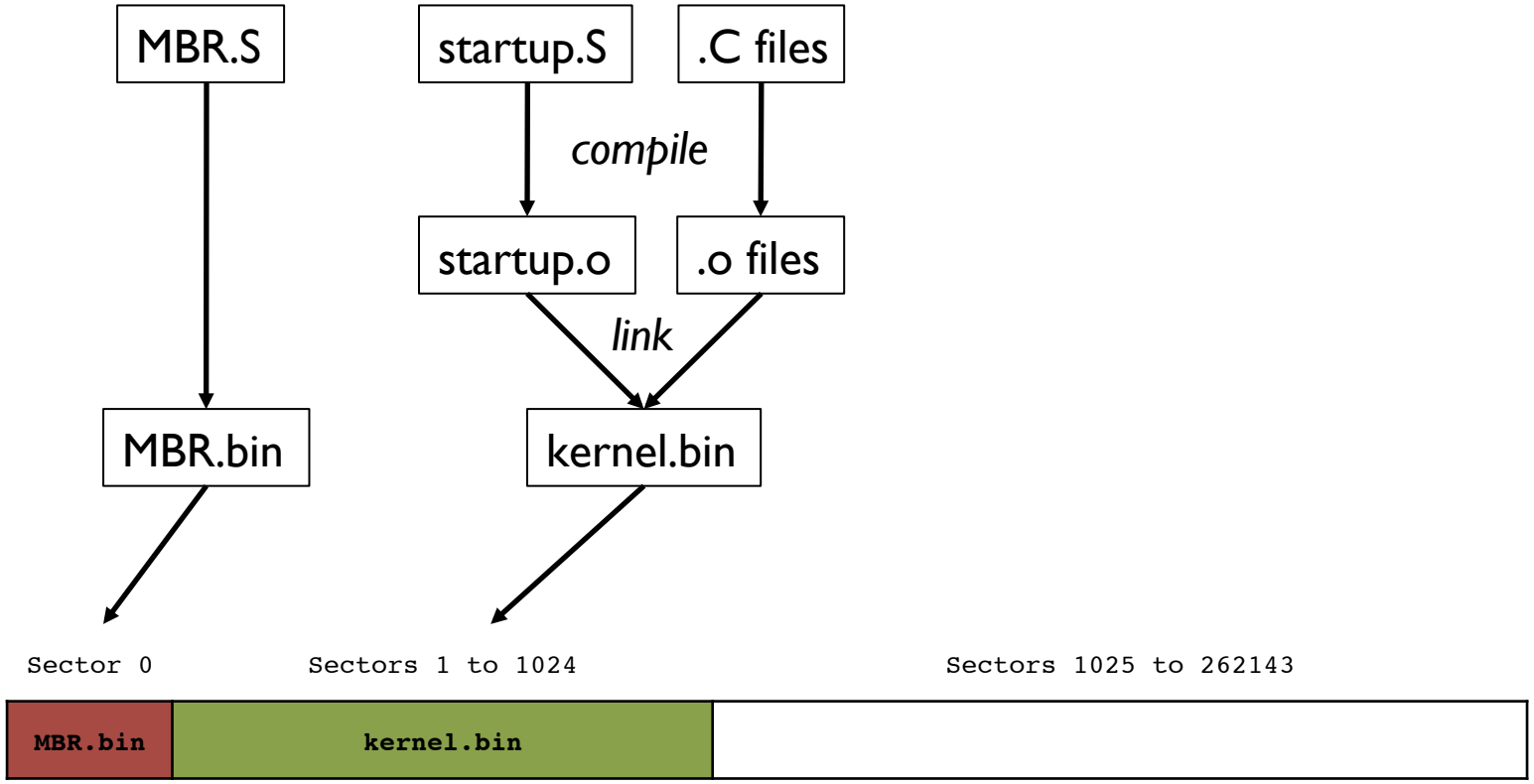
Interrupt Map



The Console


```
void start_console(void) {
     char command_buffer[513];
     uint16_t command_length;
     uint8_t status;
     int i;

     while(1) {
          for (i=0; i<513; i++) command_buffer[i]=0;
          read_command(command_buffer, &command_length);
          status = process_command(command_buffer,command_length);
          if (status == 0XFF) return;
     }
}
```

Putting It All Together



SOS.dsk (128 MB)

GDT Entries for User Programs

entry 0	`0x0000000000000000`	not used
entry 1	`0x00cf9a000000ffff`	base=0, limit=4GB, code, DPL=0
entry 2	`0x00cf92000000ffff`	base=0, limit=4GB, data, DPL=0
entry 3	`0x0000000000000000`	for user code (not initialized)
entry 4	`0x0000000000000000`	for user data (not initialized)
entry 5	`0x0000000000000000`	for TSS (coming up)

```
extern GDT_DESCRIPTOR gdt[6];
```

Default Exception Handler

exceptions.c

```
void default_exception_handler(void) {
    asm volatile ("movl $0x10, %eax\n"
                  "movl %eax, %ds\n"
                  "movl %eax, %es\n"
                  "movl %eax, %fs\n"
                  "movl %eax, %gs\n");

    puts(" OUCHH! Fatal exception. ");

    asm volatile ("movl %0,%%esp\n"::"m"(console.cpu.esp));
    asm volatile ("movl %0,%%ebp\n"::"m"(console.cpu.ebp));
    asm volatile ("pushl %0\n"::"m"(console.cpu.eflags));
    asm volatile ("popfl\n");
    asm volatile ("jmp *%0\n": :"r" (console.cpu.eip));
}
```

```
for (i=0; i<32; i++) // all of these are exceptions
    install_interrupt_handler(i,default_exception_handler,
                                0x0008,0x8E);
```

User Programs

```
void main() {
    int i;
    for (i=0; i>=0; i++);
}
```

equivalent opcodes

```
55 89 e5
83 ec 10
c7 45 fc
00 00 00
00 eb 04
83 45 fc
01 83 7d
fc 00 79
f6 c9 c3
```

```
push ebp
mov ebp,esp
sub esp,0x10
mov DWORD PTR [ebp-0x4],0x0
jmp 0x13
add DWORD PTDR [ebp-0x4],0x1
cmp DWORD PTR [ebp-0x4],0x0
jns 0xf
leave
ret
```

Executable Format

```
void main() {
    int i;
    for (i=0; i>=0; i++);
}
```

gcc

Linux ELF format

```
7F 45 4C 46 01 01 01 00 00 00 00
00 00 00 00 00 02 00 03 00 01 00
00 00 00 83 04 08 34 00 00 00 38
11 00 00 00 00 00 00 34 00 20 00
09 00 28 00 1E 00 1B 00 06 00 00
00 34 00 00 00 34 80 04 08 34 80
04 08 20 01 00 00 20 01 00 00 05
00 00 00 04 00 00 00 03 00 00 00
54 01 00 00 54 81 04 08 54 81 04
08 13 00 00 00 13 00 00 00 04 00
00 00 01 00 00 00 01 00 00 00 00
00 00 00 00 80 04 08 00 80 04 08
98 05 00 00 98 05 00 00 05 00 00
00 00 10 00 00 01 00 00 00 14 0F
00 00 14 9F 04 08 14 9F 04 08 FC
00 00 00 04 01 00 00 06 00 00 00
00 10 00 00 02 00 00 00 28 0F 00
00 28 9F 04 08 28 9F 04 08 C8 00
00 00 C8 00 00 00 06 00 00 00 04
00 00 00 04 00 00 00 68 01 00 00
68 81 04 08 68 81 04 08 44 00 00
00 44 00 00 00 04 00 00 00 04 00
00 00 50 E5 74 64 A0 04 00 00 A0
84 04 08 A0 84 04 08 34 00 00 00
34 00 00 00 04 00 00 00 04 00 00
00 51 E5 74 64 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00
00 00 00 06 00 00 00 04 00 00 00
```

Raw Binary Format

```
void main() {
    int i;
    for (i=0; i<0; i++);
}
```

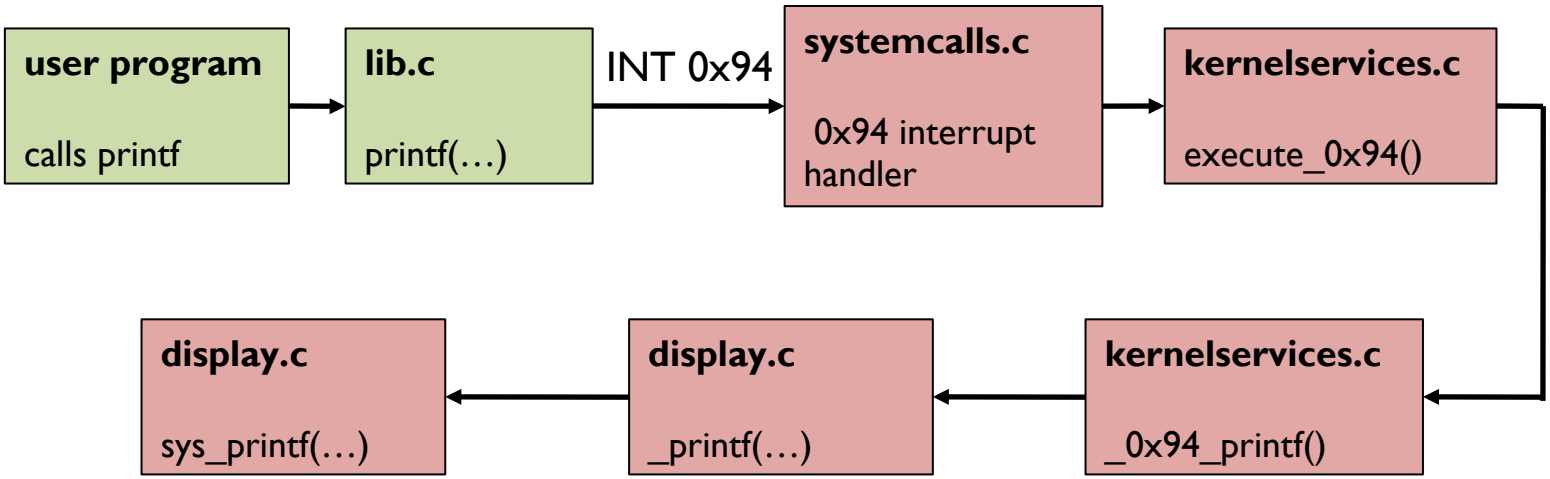
gcc2

```
55 89 e5
83 ec 10
c7 45 fc
00 00 00
00 eb 04
83 45 fc
01 83 7d
fc 00 79
f6 c9 c3
```

gcc2 is located in the `userprogs` folder in SOS1

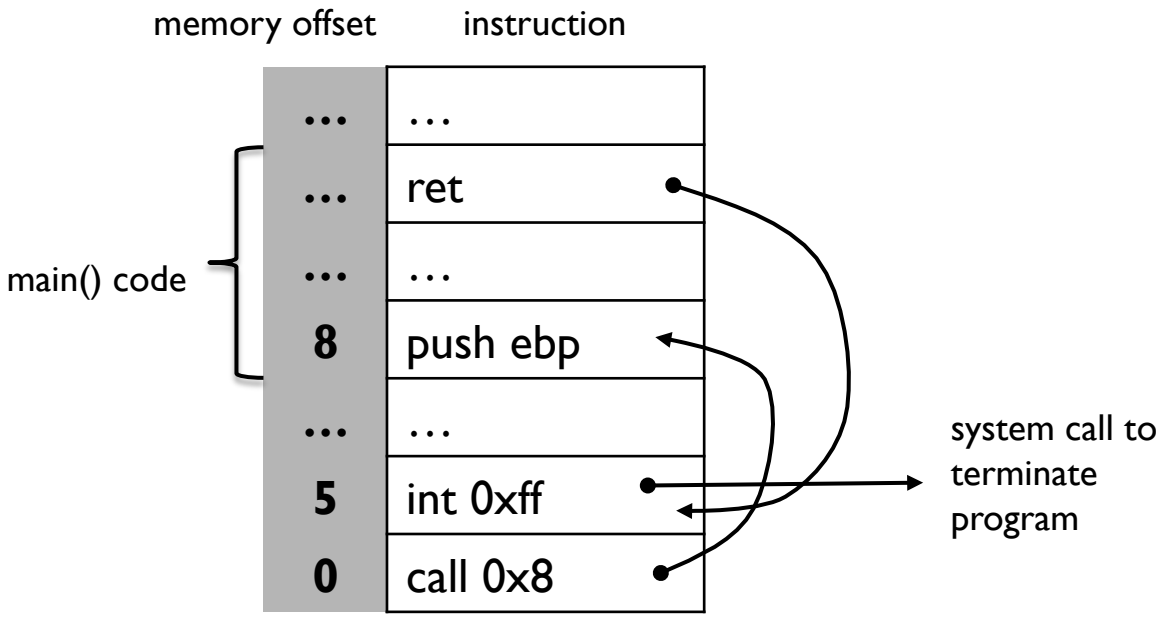
System Calls

- `printf` and `getc`



Notifying Program Termination

- How will SOS know that a program has ended?
  - we will make the program generate interrupt 0xFF when it finishes



INT OxFF Handler

systemcalls.c

```
asm("handler_syscall_0XFF_entry:\n"
    "movl %esp, %ecx\n"
    "jmp handler_syscall_0XFF\n"
);
__attribute__((fastcall)) void handler_syscall_0XFF(void) {
    asm volatile ("movl %ecx, %esp\n");
    asm volatile ("movl $0x10, %eax\n"
                  "movl %eax, %ds\n"
                  "movl %eax, %es\n"
                  "movl %eax, %fs\n"
                  "movl %eax, %gs\n");

    asm volatile ("movl %0,%%esp\n"::"m"(console.cpu.esp));
    asm volatile ("movl %0,%%ebp\n"::"m"(console.cpu.ebp));
    asm volatile ("pushl %0\n"::"m"(console.cpu.eflags));
    asm volatile ("popfl\n");
    asm volatile ("sti\n");
    asm volatile ("jmp *%0\n": :"r" (console.cpu.eip));
}
```

No state save! resumes the console

Creating User Programs

- SOS has no editor or compiler
- Type program outside of SOS
- Create program executable outside of SOS
  - use `gcc2`
    - will add "call main" and "interrupt 0xff" part to all programs
    - will also compile in `lib.c` into the executable

- `create` script copies executables from `userprogs` directory to `SOS.dsk`

```
##### Creating null disk of size 128  MB
##### Writing boot sector
##### Writing kernel image
##### Writing user programs
test.out @ 1100 (716 bytes)

Created disk image SOS.dsk.
```

The `run` Command

```
run 1100 2
```

Run the program executable that occupies 2 sectors starting at sector 1100

Implemented in `runprogram.c`

Running a Program

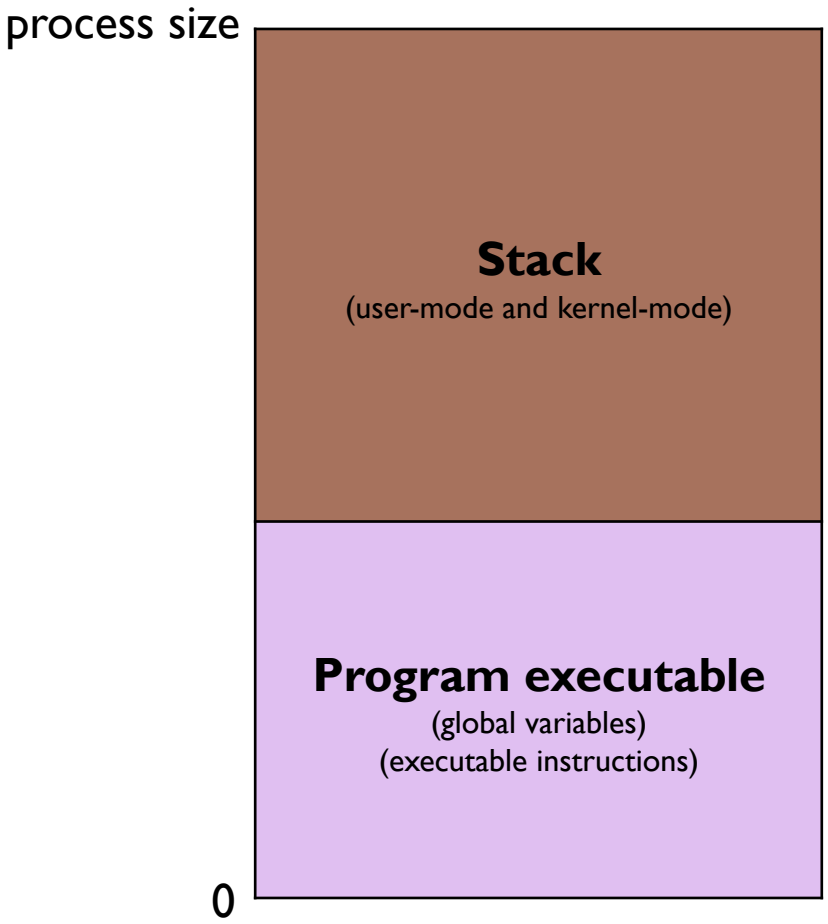
- Transfer the executable from disk to memory
  - **memory manager** tells us where in memory
- Save the console's state
  - need process control block (PCB)
- Set up a PCB for the user program
- Set up GDT entries and TSS stack
- Switch control to the user program

The DUMB Memory Manager

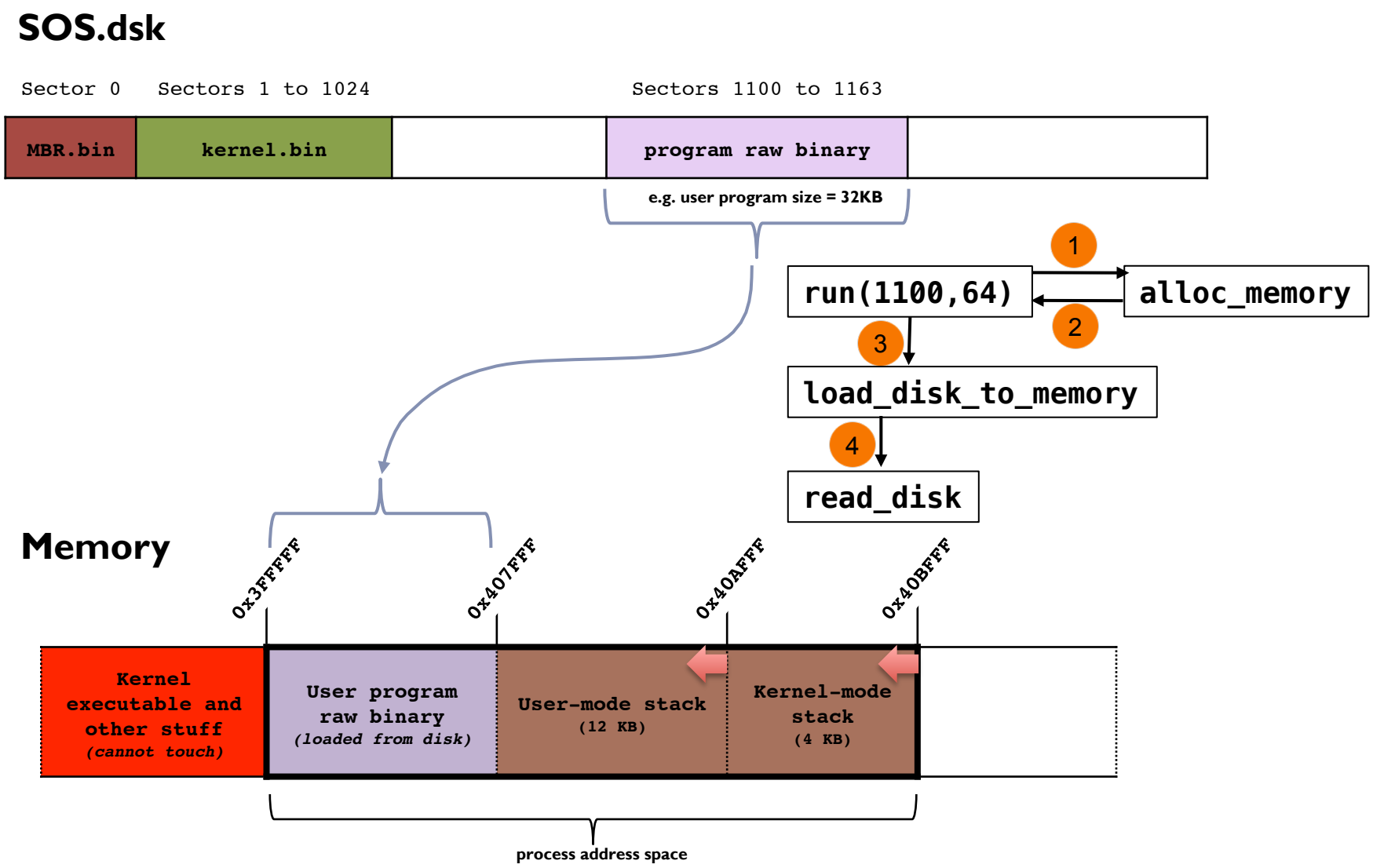
memman.c

```
extern uint64_t total_memory;

uint32_t *alloc_base; // the returned memory base address
uint32_t total_memory_bytes;

/*** Initialize memory manager ***/
void init_memory_manager(void) {
    total_memory_bytes = total_memory * 1024;

    // allocated memory always begins at 4MB mark
    alloc_base = (uint32_t *)(0x400000);
}

/*** Allocate count bytes of memory ***/
void *alloc_memory(uint32_t count) {
    int i;

    // check if we have the requested memory
    if ((uint32_t)alloc_base + count > total_memory_bytes)
        return NULL;

    return alloc_base;
}
```

Process Address Space



Layout of User Program in Memory



Process Control Block

kernel_only.h

```
typedef struct process_control_block {
    struct {
        uint32_t ss;
        uint32_t cs;
        uint32_t esp;
        uint32_t ebp;
        uint32_t eip;
        uint32_t eflags;
        uint32_t eax;
        uint32_t ebx;
        uint32_t ecx;
        uint32_t edx;
        uint32_t esi;
        uint32_t edi;
    } cpu;

    uint32_t memory_base;
    uint32_t memory_limit;

} __attribute__ ((packed)) PCB;
```

in runprogram.c

```
PCB console;
PCB user_program;

PCB *current_process
```

start address of process

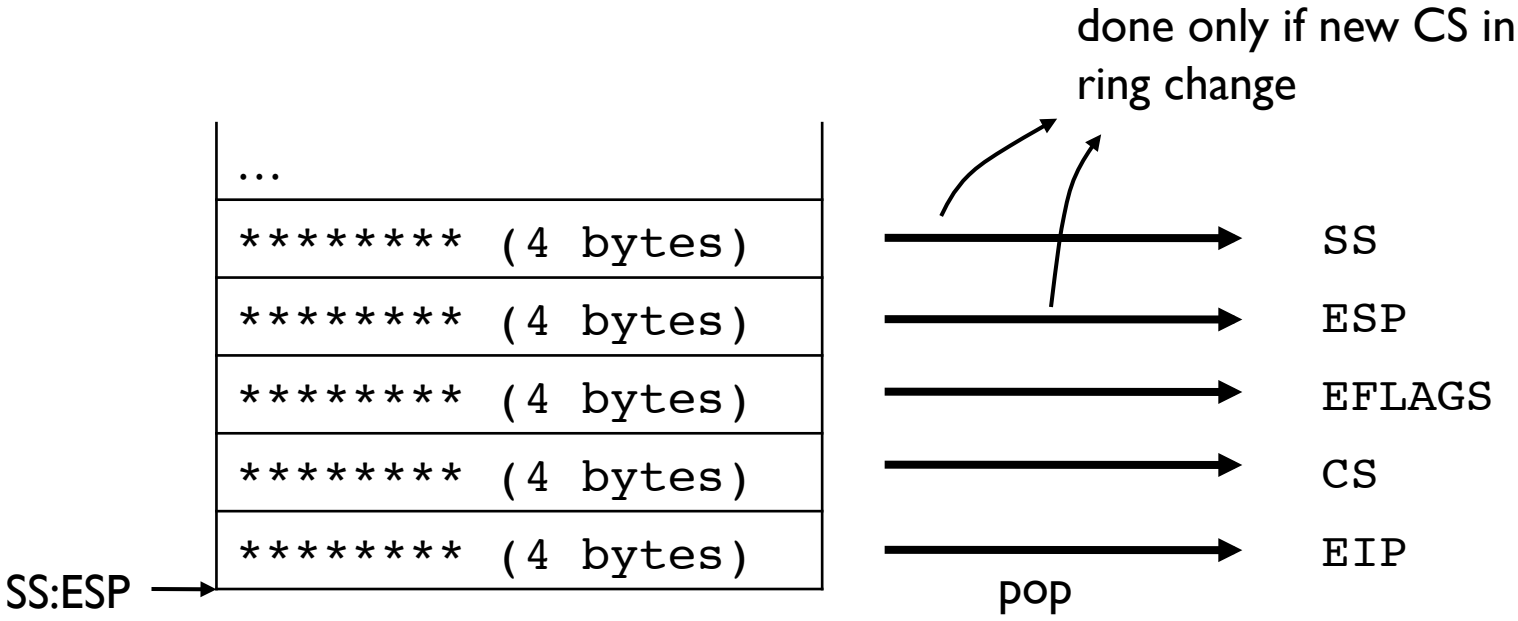
(process size – 1)

Save State to current_process

systemcalls.c

```
asm("handler_syscall_0X94_entry: \n"
    "pushal\n"
    "movl %esp, %ecx\n"
    "jmp handler_syscall_0X94\n"
);
__attribute__((fastcall)) void handler_syscall_0X94(void) {
    …
    // save CPU state in process PCB
    asm volatile ("movl %%esp, %0\n": "=r"(current_process->cpu.edi));
    asm volatile ("movl 4(%%esp), %0\n": "=r"(current_process->cpu.esi));
    asm volatile ("movl 8(%%esp), %0\n": "=r"(current_process->cpu.ebp));
    asm volatile ("movl 16(%%esp), %0\n": "=r"(current_process->cpu.ebx));
    asm volatile ("movl 20(%%esp), %0\n": "=r"(current_process->cpu.edx));
    asm volatile ("movl 24(%%esp), %0\n": "=r"(current_process->cpu.ecx));
    asm volatile ("movl 28(%%esp), %0\n": "=r"(current_process->cpu.eax));
    asm volatile ("movl 32(%%esp), %0\n": "=r"(current_process->cpu.eip));
    asm volatile ("movl 36(%%esp), %0\n": "=r"(current_process->cpu.cs));
    asm volatile ("movl 40(%%esp), %0\n": "=r"(current_process->cpu.eflags));
    asm volatile ("movl 44(%%esp), %0\n": "=r"(current_process->cpu.esp));
    asm volatile ("movl 48(%%esp), %0\n": "=r"(current_process->cpu.ss));
    …
}
```

Switching to Ring 3 Using IRETL



IRETL operation

- Push the right values in stack and use the IRETL instruction

Switching to User Process

- Save state to `console` PCB
- Create `user_program` PCB
- Call `switch_to_user_process(&user_program)`
- `switch_to_user_process(PCB *p)`
  - set up kernel-mode stack
  - set up user GDT entries
  - load CPU state from `p`
  - push values for SS, ESP, EFLAGS, CS, EIP from `p` to stack
    - SS: should point to GDT entry 4 with RPL=3
    - ESP: see process layout
    - EFLAGS: can simply keep current one
    - CS: should point to GDT entry 3 with RPL=3
    - EIP: first instruction of user program
  - IRETL

User GDT Entries

- User program will use `gdt[3]` and `gdt[4]`
  - set those up before switching to ring 3

kernel_only.h

```
typedef struct {
        uint16_t limit_0_15;
        uint16_t base_0_15;
        uint8_t  base_16_23;
        uint8_t  access_byte;
        uint8_t  limit_and_flag
        uint8_t  base_24_31;
} __attribute__ ((packed)) GDT_DESCRIPTOR;
```

Kernel-Mode Stack

- CPU switches to a different stack, a.k.a. **kernel-mode stack**, when moving from ring 3 to ring 0 (during a system call)
- How to specify this stack?
  - declare a variable of type `TSS_STRUCTURE`
    - already done `TSS_STRUCTURE TSS` in `systemcalls.c`
    - `TSS_STRUCTURE` is defined in `kernel_only.h`
  - create GDT entry that describes the memory area occupied by this variable
    - already done in `setup_TSS()` in `systemcalls.c`; this is why we have that sixth entry in the GDT

Kernel-Mode Stack (contd.)

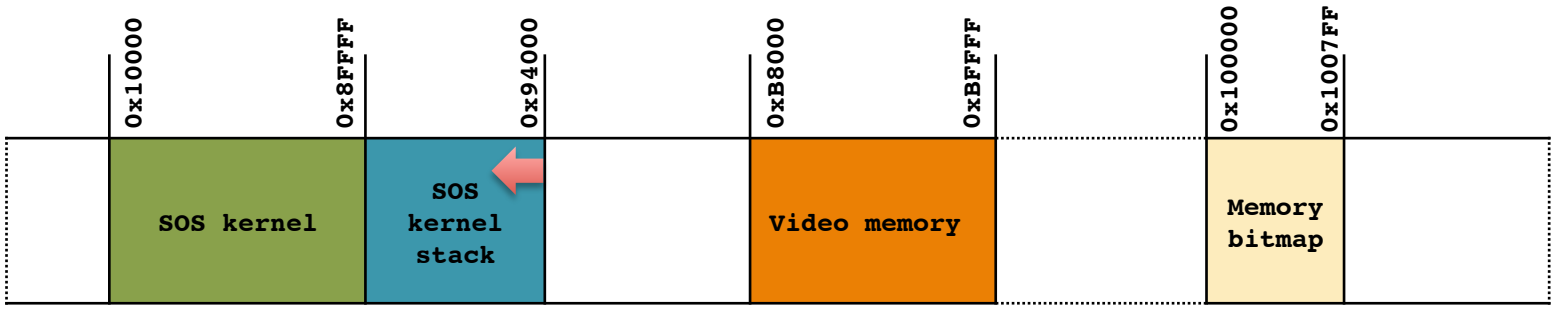
- tell CPU about location of this variable
  - already done in `setup_TSS()` in `systemcalls.c`; we use the LTR instruction for this
- put the location of the kernel-mode stack in **`TSS.esp0`** and **`TSS.ss0`**
  - must do this before switching to user program

Running Multiple Programs

- Better memory manager
- Processes with states
- A scheduler

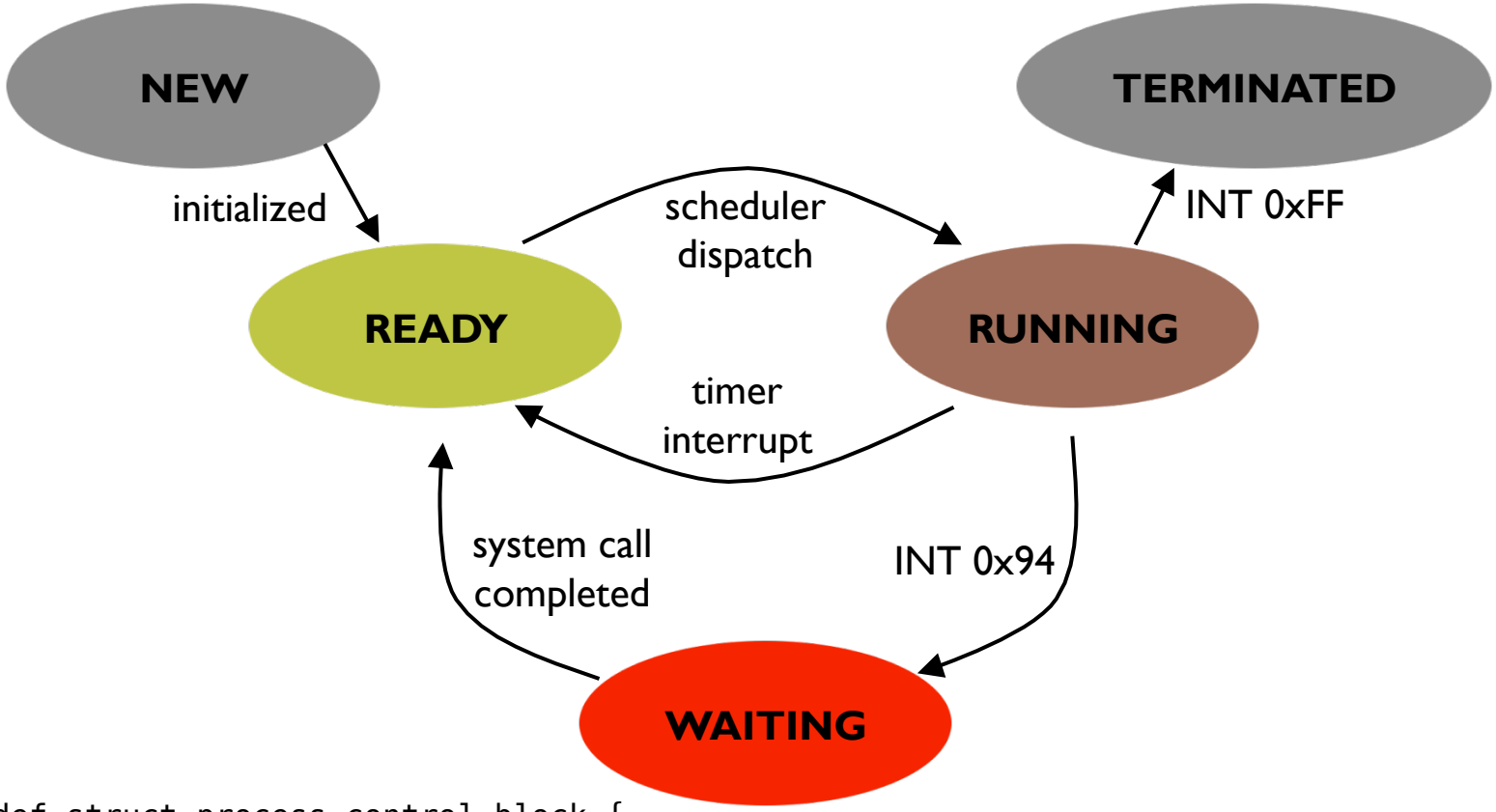
The NAÏVE Memory Manager

- SOS will support a maximum of 64MB RAM
  - allocation unit = 4KB
- Memory tracking using a bitmap
  - placed at 0x100000 and 2 KB maximum size



- First-fit allocation
  - `void *alloc_memory(uint32_t count_bytes);`
  - `void dealloc_memory(void *start_address);`

Process States

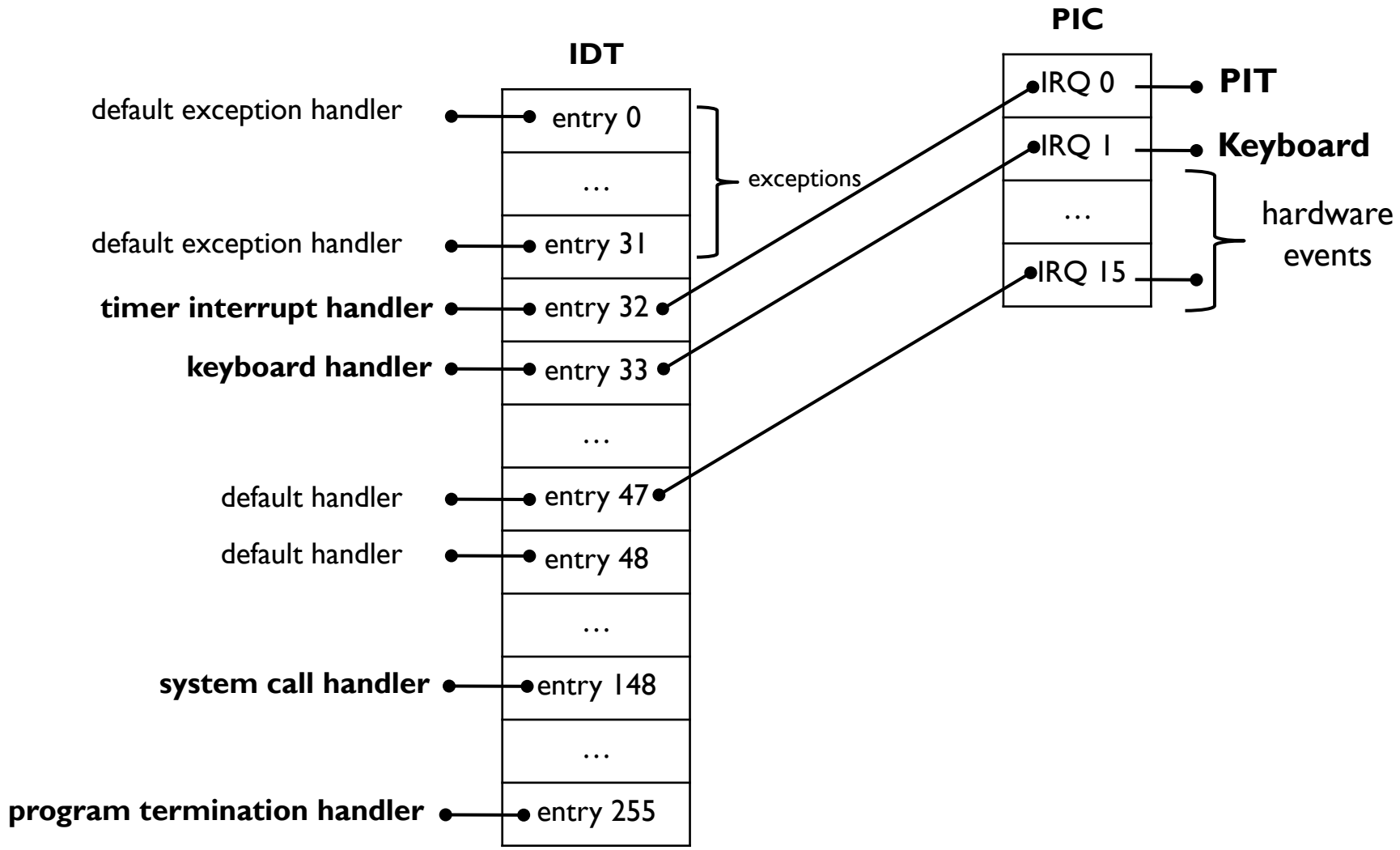



```
typedef struct process_control_block {
     struct { ... } cpu;
     uint32_t pid;
     ...
     enum {NEW, READY, RUNNING, WAITING, TERMINATED} state;
     uint32_t sleep_end;
     struct process_control_block *prev_PCB, *next_PCB;
} __attribute__ ((packed)) PCB;
```

Programmable Interval Timer (PIT)

- Intel 8253: timer hardware that generates 1193182 output signals every second
  - reconfigured to generate 100 signals per second, or a signal every 10 millisecond
- Connected to IRQ0 of PIC
- We connected IRQ0 to interrupt 32
- PIT signal → Interrupt 32 fired
  - once every 10 milliseconds

Interrupt Mapping



Timer Interrupt Handler

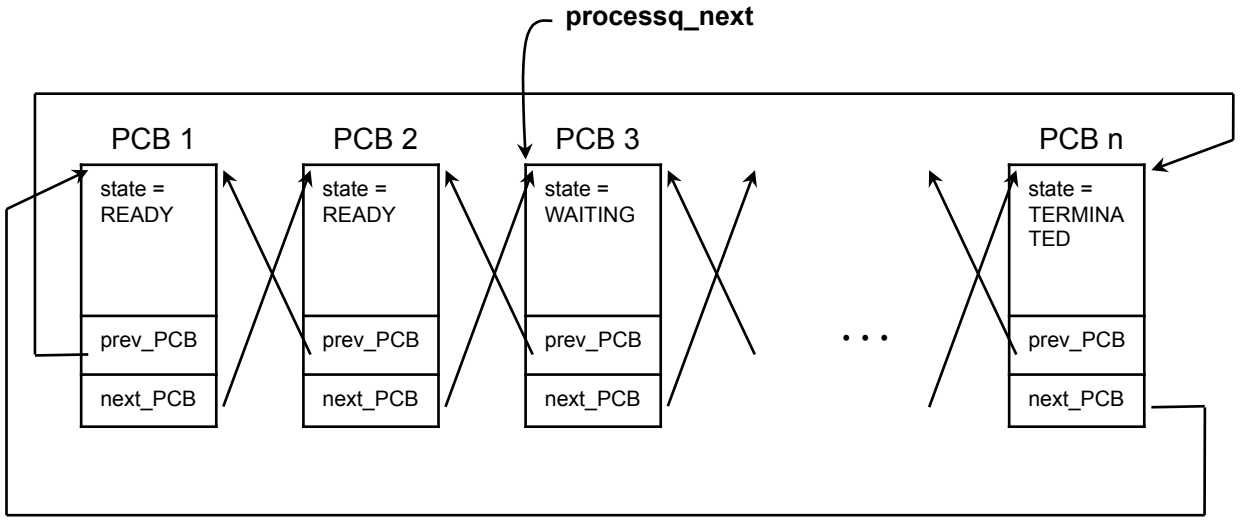
```
/*** The timer (IRQ0) handler ***/
asm("handler_timer_entry: \n"
    // CPU would have already pushed these in order:
    // [SS, ESP](only if not in Ring 0), EFLAGS, CS and EIP
    "pushal\n" // push all general purpose registers
    // save stack pointer
    "movl %esp, %ecx\n"
    "jmp timer_interrupt_handler\n"
);
__attribute__((fastcall)) void timer_interrupt_handler() {
    // reload stack pointer (discards C function prologue)
    asm volatile ("movl %ecx, %esp\n");

    // save CPU state to current_process PCB
    ...


    elapsed_epoch++; // each epoch is 10ms long

    update_display_time();

    ...

    // invoke scheduler
    schedule_something();
}
```

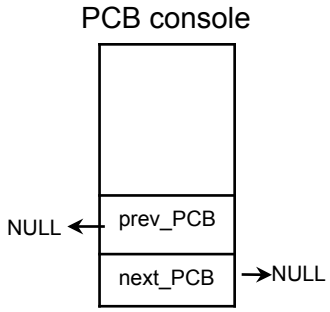
Running a Program

- Transfer the executable from disk to memory
  - **NAÏVE memory manager** tells us where in memory
- ~~Save the console's state~~
- Set up a PCB for the user program
  - allocated dynamically using `alloc_memory`
- Add the process PCB to a *process queue*
  - `PCB *add_to_processq(PCB *);`
  - process execution is not started immediately

- SOS2: only runs programs as background jobs
  - no interaction with program possible ☹
  - trying to use getc will block the process forever

Process Queue





`processq_next`: beginning of queue

`add_to_processq` inserts PCB before `processq_next`

Round-Robin Scheduler

- Picks READY process from process queue, but every alternate timer interrupt it runs the console process
- E.g. P1, P2 and P5 are READY processes
  - schedule: P1→console→P2→console→P5→console→P1→...
- Picks console if no process is READY

sleep System Call

```
/*** Make process sleep ***/

void _0x94_sleep(void) {
    uint32_t tts = current_process->cpu.ebx;

    current_process->sleep_end = get_epochs() +
                                 tts/get_epoch_length();

    current_process->state = WAITING;
}
```

Wake up process before scheduling once current time > sleep_end

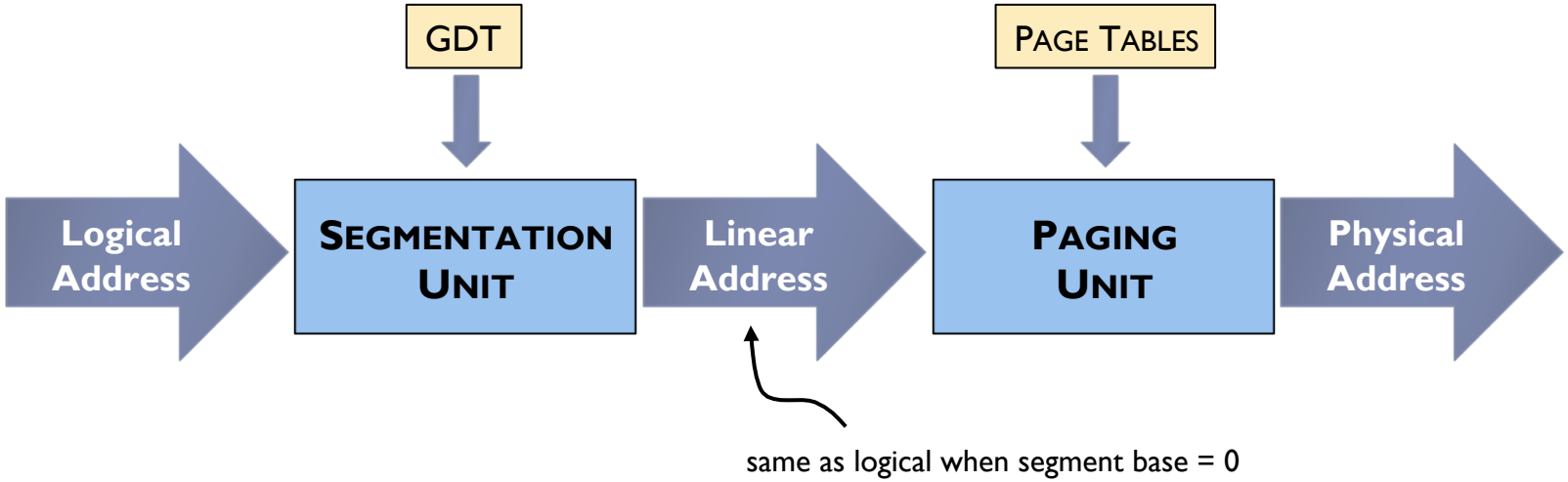
schedule_something()

- Clean up TERMINATED process
  - remove from process queue and deallocate memory
  - `PCB *remove_from_processq(PCB *);`
- Wake up WAITING process that blocked by calling `sleep`
  - change state to READY
- Run scheduler: pick READY process from queue, or the console
- Switch to picked process

Paging

- Logical address space layout
  - kernel and user programs
- Page directory/tables set up
  - kernel and user programs

Address Translation



Flat Segmentation

entry 0	0x0000000000000000	not used
entry 1	0x00cf9a000000ffff	base=0, limit=4GB, code, DPL=0
entry 2	0x00cf92000000ffff	base=0, limit=4GB, data, DPL=0
entry 3	0x00cffa000000ffff	base=0, limit=4GB, code, DPL=3
entry 4	0x00cff2000000ffff	base=0, limit=4GB, data, DPL=3
entry 5	0x0000000000000000	initialized in setup_TSS()

Logical Addresses

- 32-bit logical address space
  - address range 0 to 2^{32}-1 = 0xFFFFFFFF

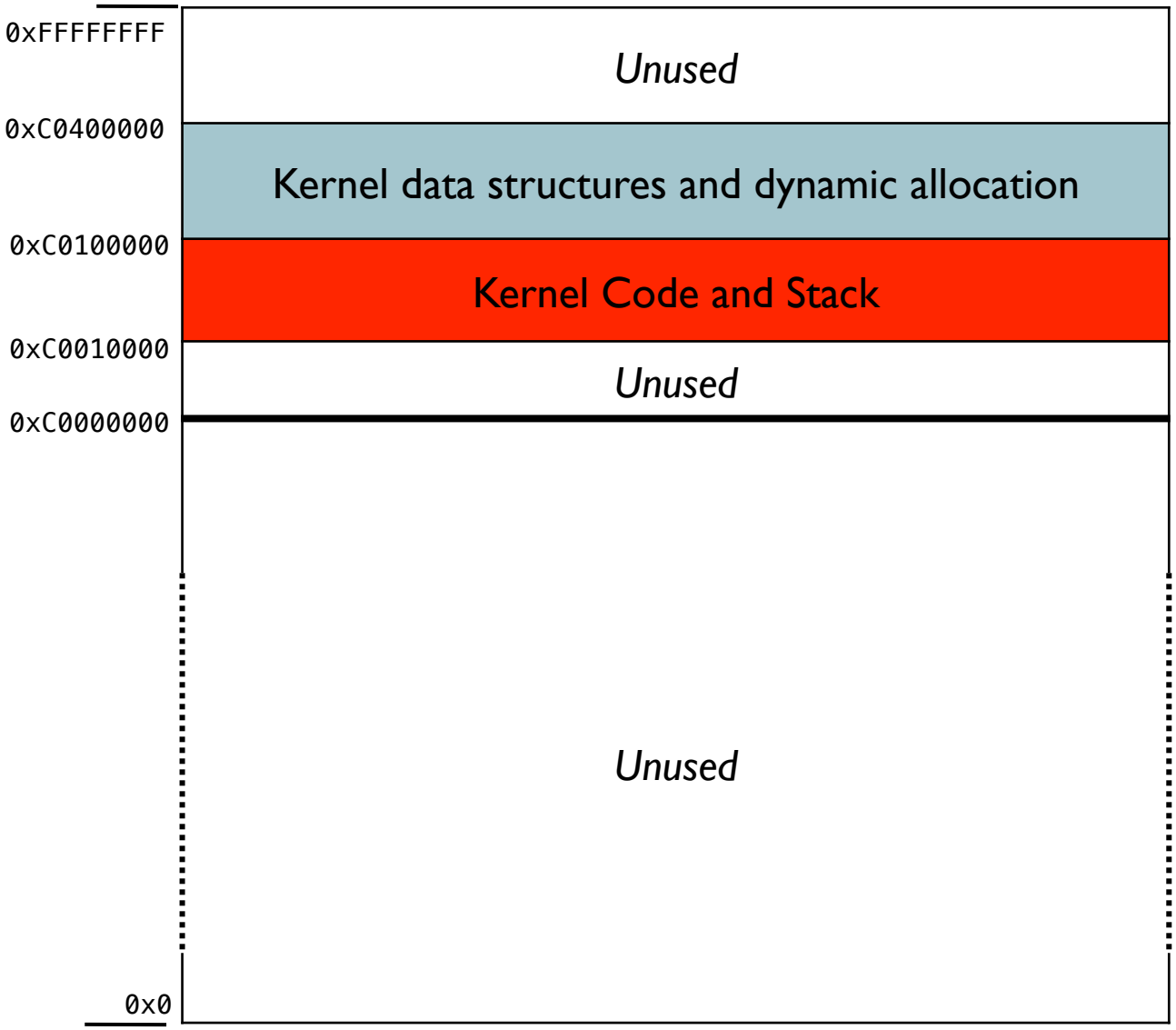
- Design how the kernel and user programs will use their logical address space

NAÏVE Physical Memory Manager

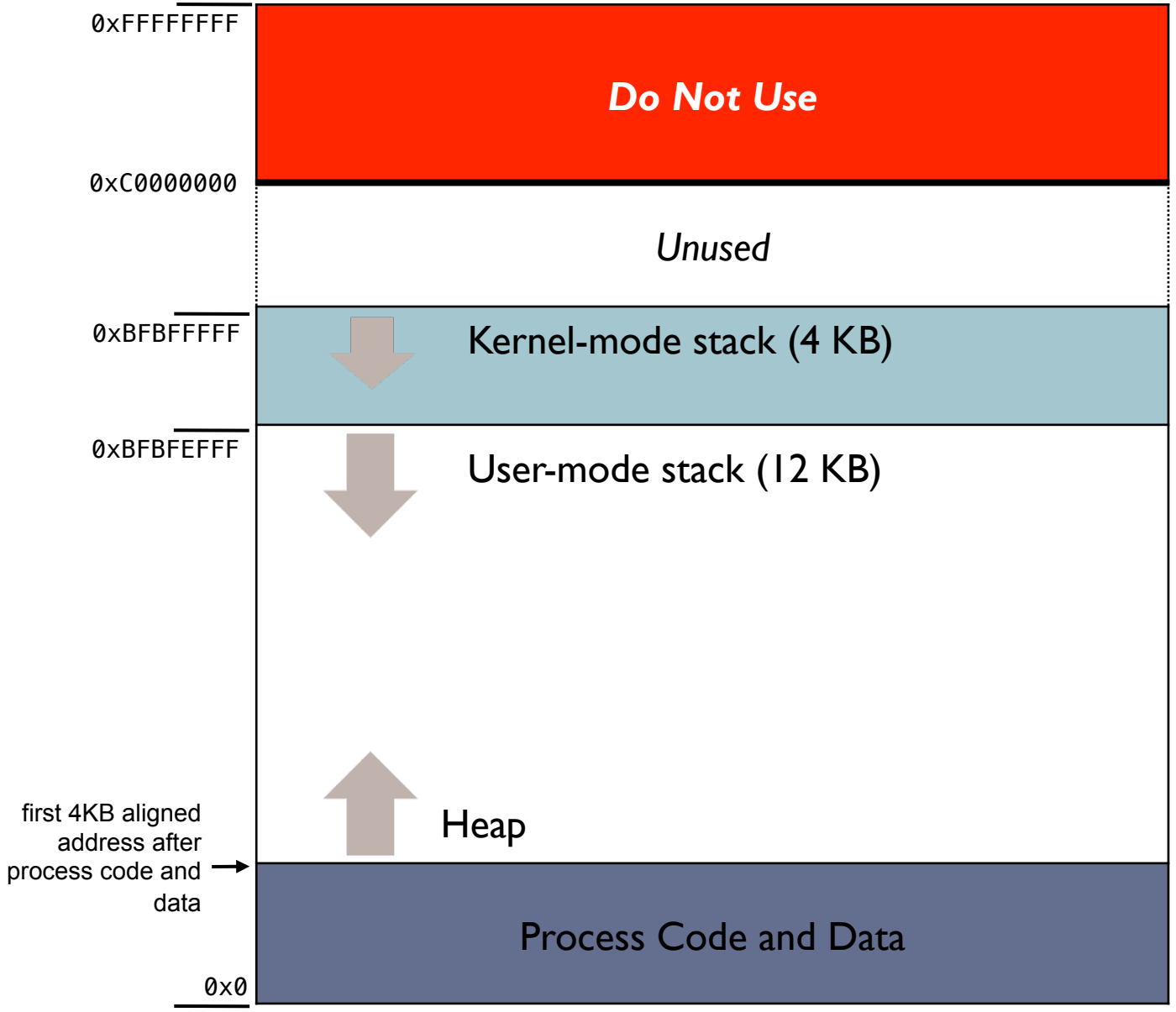
- Same as SOS2, but
  - first 4 MB used by kernel only
  - rest used for user programs

- Functions
  - `void *alloc_frames(uint32_t n_frames, bool mode);`
    - `mode = KERNEL_ALLOC` : allocation from kernel memory
    - `mode = USER_ALLOC` : allocation from user memory
  - `void dealloc_frames(void *start, uint32_t n_frames);`

Kernel Logical Address Space



User Process Logical Address Space



New Entries in PCB

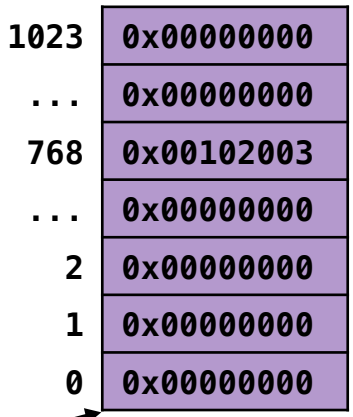
```
typedef struct process_control_block {
    struct {
        . . .
    } cpu;
    uint32_t pid;
    uint32_t memory_base;
    uint32_t memory_limit;
    . . .

    struct {
        uint32_t start_code;
        uint32_t end_code;
        uint32_t start_brk;
        uint32_t brk;
        uint32_t start_stack;
        PDE *page_directory;
    } mem;

    struct {
        uint32_t LBA;
        uint32_t n_sectors;
    } disk;
} __attribute__ ((packed)) PCB;
```

not needed since base and limit are fixed

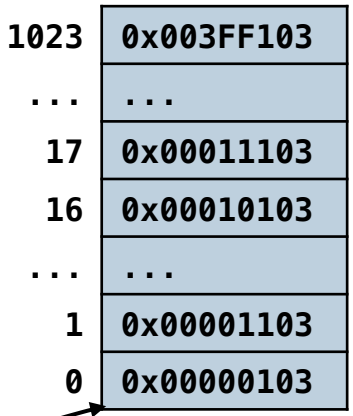
Kernel Page Tables

- First 4 MB of physical memory is for kernel use
  - physical address range: 0x0 to 0x3FFFFF
- Kernel uses logical addresses 0xC0000000 to 0xC03FFFFF
- Need to set up page directory and page tables to map logical 0xC0000000–0xC03FFFFF to physical 0x0–0x3FFFFF

Kernel Paging Structures



PDE *k_page_directory → 0x101000
[present and read/write flags set]



PTE *pages_768 → 0x102000
[present, global and read/write flags set]

Example: **logical address = 0xC0012345**

Which page directory entry: `0xC0012345 >> 22 = 768`
Where is page table of entry 768: `0x00102003 & 0xFFFFF000 = 0x102000` (pages_768)
Which page table entry: `(0xC0012345 >> 12) & 0x000003FF = 18`
Where is physical frame start: `pages_768[18] & 0xFFFFF000 =`
`0x00012103 & 0xFFFFF000 = 0x12000`
What offset: `0xC0012345 & 0x00000FFF = 0x345`

physical address = 0x12000 + 0x345 = 0x12345

Loading Page Tables

- How to tell CPU to use a particular page directory?
- Load CR3 register with **physical address** of page directory
  - kernel's page directory is located at 0x101000
  - to load kernel's page directory, load CR3 with 0x101000
- Address translations will then be done automatically by MMU using kernel's page directory and page table

Process Paging Structures

- Set up in `init_logical_memory`
- One page directory
- Multiple page tables: corresponding to address ranges belonging to
  - process code and data
  - process user-mode stack
  - process kernel-mode stack
- Non-zero entry only if corresponding logical address range is used by process

init_logical_memory

- How many frames for process code/data/stacks?
- Allocate those frames
  - we now know physical address of the different process sections
- How many page tables will be needed?
- Allocate frames for page directory and page tables
- Fill in the entries
  - most of them will be zero

Higher Half Kernel

- When switching to user process, CR3 is loaded with page directory address of process
  - `mem.page_directory`
- During system calls, kernel code runs
  - problem: user process page directory has no mapping for logical address ranges used by kernel code!
- Solution: fill entry 768 of user process page directory with value from kernel's page directory
  - `user process page directory entry 768 = k_page_directory[768]`
  - no conflicts since user process does not use adress 0xC0000000 onwards

How is Paging Enabled?

- Set up page directory/tables
- Load physical address of page directory in CR3
- Enable bit 31 in the CR0 register

Page Fault Exception Handler

- Exception 14 is thrown if page directory or page table entry is not present (bit 0 is zero) when translating a logical address
  - also called **page fault exception**
- Display error and terminate process on page fault (SOS3)
- Otherwise, handle page fault
  - bring page from disk swap space
  - allocate space for page and update page tables
  - terminate process if illegal access