Kernel development knowledge documentation

Kernel syntax and code

Video Memory:

Video Memory or VM is a array type of screen where the entire screen is like an array on Video_memory[] and can be called a magical window. It is used when creating basic kernels. To use this we create a volatile (Un-optimised) variable called video_memory that we equal to the adress 0xB8000 which is where the VM is located in the ram.

Important calculation to find the current cursor position. 24 is current row and j is current column:

```
Position =(24 * 80 + j) * 2;
```

Color:

The text and background color is chosen with the color code 0x00 Where 0x just stand for hexadecimal, the first 0 after the x controls what background color the text will get and the last 0 controls what color the text shall have. For example white text on a black background would be 0x0F 0 for black and F for white. Important note is that in video_memory text and background color attributes can only be changed at a even position. If the position is not even then it is instead what text shall appear there. So what char shall be put there.

BitWise Operations

Left shift:

Left shift means that we have lets say bg << 4 this is a left shift operation

This takes the background color and shift all its bits 4 positions to the left

If bg is 3 (binary: 0011), shiftiing left by 4 gives 00110000

OR Operation:

fg | (bg << 4) This is a bitwise OR operation

It combines the fg and shifted background using OR

OR means if either bit is 1 the result is 1.

OR operation and shift operation:

```
fg = 5 (binary: 0101)     // foreground color
bg = 3 (binary: 0011)     // background color

bg << 4 = 00110000        // background shifted to upper bits
fg      = 00000101        // foreground in lower bits

Result  = 00110101        // combined: bg in upper 4, fg in lower 4
        = 53 in decimal
```

Bitwise operations are important for functional and code for kernel development.

C handling of hex:

Fantastical c handles hex and many more forms automatically. If you write:

Int a = 15

int b = 0x0F

int c = 017

int d = 0b1111

The c compiler will interpret them all as the same thing…. the decimal number 15!

va_arg, va_list, va_start, va_end

Magic. They are used to give a parameter unlimited amount of arguments to be taken in.

va_list is the so called list created for all these unlimited values.
va_starts says that we begin after the chosen start parameter, example va_start(start_arg) in void function (start_arg, …) it will begin after start_arg. Like a pointer.
va_arg is the way to grab the current parameter argument pointed at and then move forward in the list to the next parameter/argument to catch. Like a pointer moving forward in a list.
va_end just puts the end of the list to clean up.

```c
va_list args;         // Create a tape reader (but no tape loaded)
va_start(args, count);  // Load the tape and position at first argument
va_arg(args, int);    // Read current value, advance tape one position
va_arg(args, int);    // Read current value, advance tape one position
va_end(args);         // Eject the tape and clean up
```

EAX and EDX

EAX and EDX are 32-bit general purpose registers in x86 processors.

EAX is the Accumulator register

EDX is the Data register

AL

AL is part of EAX.

AL = lower 8 bits of EAX

AH = next 8 bits of EAX

AX lowers 16 bits of EAX

```
EAX: [31-16 bits][15-8 bits AH][7-0 bits AL]
```

outb only uses the AL part of EAX.

Constraint Letters for outb and inb

In inline assembly, constraint letters tell the compiler which register or addressing mode to use:

“a” use EAX register or AX/AL for small operations

“d” user EDX register

“N” use immediate integer constant 0-255 for x86 port numbers

Nd means use “N” (immediate constant) OR “d” (EDX register)

If port number is 0-255: compiler can use it as immediate value

If port number is >255: compiler must put it in EDX register

This Nd approach is a optimization to not generate UN-necessary long instructions.

Valid single-letter constraints for x86:

- `a, b, c, d` = specific registers (EAX, EBX, ECX, EDX)
- `r` = any general register
- `m` = memory location
- `i` = immediate constant
- `N` = immediate constant 0-255

Outb and Inb

These are used to write a byte to and read a bite from a hardware port.
The process is described below.

- `static inline` = function gets compiled directly into calling code (faster)
- `__asm__ volatile` = inline assembly that won't be optimized away
- `"outb %0, %1"` = assembly instruction with placeholders %0, %1
- `: "a"(value)` = put `value` in register AL (the "a" constraint)
- `"Nd"(port)` = put `port` in register DX ("N" = 0-255 immediate, "d" = DX register)
- `"=a"(result)` = output goes to `result` variable via AL register

```
outb(0x70, reg);
return inb(0x71);
```

We select register on port 0x70 and then read data from 0x71

BCD to decimal/binary

```
return ((bcd >> 4) * 10) + (bcd & 0x0F);
```

Lets say BCD is 0x23 then we have in binary 0010 0011 well if we go through with the equation above.

- 0010 0011 → 0000 0010 → 2(in decimal) *10 → 20
- 0010 0011 & 0000 1111 (0x0F) → 0000 0011 → 3 (in decimal) → 3
- Now in the end stage we simply add them together and we get 20 + 3 equals to 23 and then return it.
  The BCD can not take in values higher than 0-9 so a, b, c, d, e, f can not be used. Only 0-9. The reason being you can't write 0x2A since A = 10. You would not get a human readable number in a 8 bit format which would undermine the point of the BCD system.

Separation of code into .c and .h files

Separated code into different lib to clean up code. The format for the .h file is as follow:

```h
#ifndef IO_H
#define IO_H

#include <stdint.h>

void outb(uint16_t port, uint8_t value);
uint8_t inb(uint16_t port);

#endif
```

Then after the .h file has been made we have to create the .c file. We will use the same function files for this example:

```c
#include "../include/io.h"

void outb(uint16_t port, uint8_t value) {
    __asm__ volatile ("outb %0, %1"
                      :
                      : "a"(value), "Nd"(port));
}

uint8_t inb(uint16_t port) {
    uint8_t result;
    __asm__ volatile ("inb %1, %0"
                      : "=a"(result)
                      : "Nd"(port));
    return result;
}
```

We have included the io.h file for this and also then created the functionality of the functions defined in io.h

Then lastly we have to update the metafile or in my case the run.sh file since I manualy update the files. Update as below:

```
i686-elf-gcc -c lib/io.c -o lib/io.o -std=gnu99 -ffreestanding -O2 -Wall -Wextra
```

With this line added we have now made the io.c file compile into a .o (object file)
We then just need to link it:

```
i686-elf-gcc -T linker.ld -o myos.bin -ffreestanding -O2 -nostdlib boot.o kernel.o lib/io.o lib/rtc.o lib/vga.o -lgcc
```

Then just lastly we add it to the isodir/boot/ folder and run the following command to create the ISO file:

```
grub-mkrescue -o myos.iso isodir
```

Now we run the ISO file with this command:

```
qemu-system-i386 -cdrom myos.iso
```

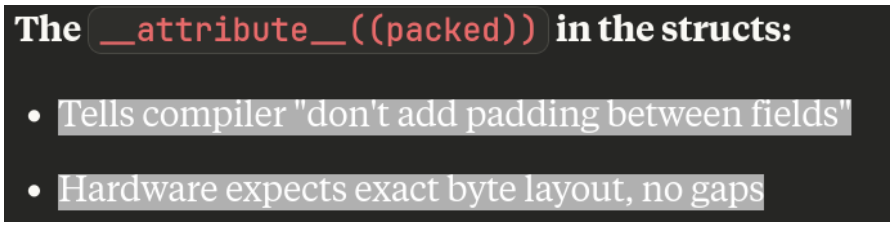
IDT

IDT is like the phone book for interrupts, when interrupt #0 happens the CPU looks up entry #0 in the IDT to find which function to call.
We need it because the CPU need to know where your interrupt handler function are in memory.
Without the IDT the CPU would have no idea what to do when and interrupt occurs

We use a `static struct idt_entry idt[256];` which will contain all the possible interrupts locations.
Intel x86 supports interrupt numbers 0-255 which becomes 256 entires.

We also use a `static struct idt_ptr idtp;` to tell our CPU where our IDT table is located in memory.
It is a pointer to our table.

We use static to make the table internal to this file only. No other file shall mess with out interrupt table.



Usable Interrupts

0-31 are the CPU exceptions that fire when something wrong happens like dividing by zero, page fault, ect.

This means that 32-255 are available for hardware like keyboard, timer, etc.

IDT_entry

```
void set_idt_entry(uint8_t num, uint32_t handler_address, uint16_t selector, uint8_t flags) {
    idt[num].base_low = handler_address & 0xFFFF;
    idt[num].base_high = (handler_address >> 16) & 0xFFFF;
    idt[num].selector = selector;
    idt[num].always0 = always0;
    idt[num].flags = flags;
}
```

base_low and base_high

Handler address is 32 bits and the IDT entry format requires it split. This is just how Intel designed the hardware format.

Selector

Selector is the points to which code segment the handler runs in, for kernel code this typically is 0x08 (first entry in GDT)
Well use 0x08 for all our handlers.

Flags

Tells CPU “This is an interrupt gate, kernel-level access only”

Well use 0x8E (Present, ring 0, 32-bit interrupt gate)

Magic numbers?

The number 0x08 is standard kenel code segment selector.
The number 0x8E is Binary 10001110 = Present(1) + Ring0(00) + InterruptGate(01110)

Present (P) – Bit 7 = 1

This IDT entry is valid and ready to use

CPU checks this first, if 0, CPU generates a fault instead of calling the handler

Can be thought of like a light switch 1=on/ready, 0=off/broken

DPL (Descriptor Privilege Level) – Bits 6-5 = 00

Only Ring 0 (Kernel) code can trigger this interrupt

Prevents user programs from triggering kernel interrupts directly

0=kernel (most privileged, 3=user programs (lease privileged)

Bit 4 = 0

This is a system descriptor (always 0 for interrupts)

Just how Intel designed it

Bits 3-0 = 1110

This is a 32-bit interrupt gate

It is called gate because it’s a controlled entrance point into the kernel code.

IDT lidt

```
__asm__ volatile ("lidt %0" : : "m" (idtp));
```

This code above is a inline assembly code that tells the CPU to now use our IDT instead of the BIOS’s own IDT.

Lidt stands for Load IDT which tells CPU hardware to use this IDT table

Now when this is done the CPU will now look at our IDT when an interrupt activates.

Important IDT note

Always check current segment registers, never assume. I have 16 in my system not 8.

What I did wrong

Used 0x08 (which is 8 in decimal) but the actual kernel code segment is 16

The CPU could not find the handler because we pointed to the wrong memory segment.

PIC

Pic handles all the interrupts coming from hardware. So we have a Master PIC and a Slave PIC. Master PIC is the only one who can communicate with the CPU. The Master PIC handles IRQ 0-7 while the Slave PIC handles IRQ 8-15. A problem is that these do not work with our IDT since we saw before that 32 is where usable interrupts can be stored. So what we do is that we convert it.

Master

The master is directly connected to the CPU and handles IRQ 0-7

It receives signals from Slave PIC and Makes decisions about which interrupt to send to CPU

The output from the Master PIC is then conformed to the IDT

Slave

The slave PIC handles IRQ 8-15

it sends its interrupts to the master PIC
The master and slave PIC are connected through the IRQ 2 line.

Sequence of interrupts:

When a interrupt happens this the flow of the sequence:

[Mouse movement] → IRQ 12 → Slave PIC → IRQ 2 → Master PIC → IRQ 12 → CPU

When it sends it to the CPU we comform it to instead tell the CPU “Hey, instead of at place 12 we want you to go to place 12 + 32 so 44 in our IDT instead.”

The reason why we have the function pic_disable_irq and pic_enable_irq is because of interrupts such as timer which fires thousands of times per second.

To not be overwhelmed we turn turn of the IRQ for them to only handle our chosen ones at that moment. Also they are not secure so we introduce more secure and controlled IRQ control. Which is why it is called PIC Programmable Interrupt Controller

If we don’t have this controll then all 16 IRQ’s would fire constantly.

Physical vs Virtual memory

The physical memory is the Ram chips in your computer with addresses as 0x0, 0x1 …. up to the your max Ram amount.

The virtual memory works that each process has it’s own fake virtual address space, so this kernel thinks it owns addresses like 0x100000, ... etc which are not real but the CPU take these false and virtual addresses and translate them into real physical addresses on the RAM using something called a page table.

When the kernel boots, the grub boots it into a memory at a specific location.

Page tables

Page tables makes it so that each process get its virtual memory translated into actual physical memory, this is something the CPU does.

It’s like a phone book that translates fake addresses to real addresses.

Finding end of kernel

We can in the linker put start and end signs that mark the start address of the kernel and the end address of the kernel. After the end address we can allocate the heap. Now the kernel can grow and the heap wont collide because it will constantly be at the end of the kernel.
We can do this by entering:

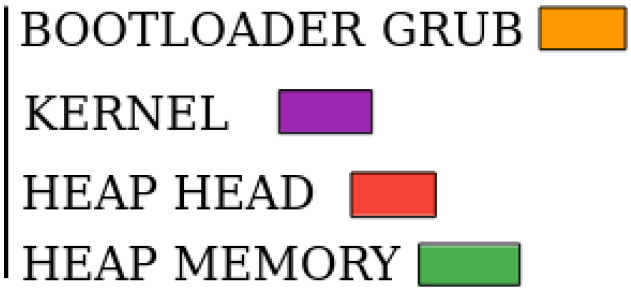
_end = .;

The construction of heap

The heap are blocks of memory where we and the processes can allocate memory. A heap can be made in different ways but for this kernel it will be a heap block containing, size of the heap memory, flag if it is in use, and what the address to the next heap block in memory is. It will be together as a linked list. The heap block do not need to have the address to the next block since they will be next to each other in memory. (Update): We can just use size to get the next memory block available.
The heap begins as a giant block of memory spanning in our case 1MB from the end of kernel. Then when malloc is called and allocation happens we take a chunk of the heap and seperate it making 2 blocks then malloc again and we have 3 blocks and so on until the giant heap has been seperated into small usable memory blocks. Which is why it is dynamic.

So kmalloc will look through this linked list later when built and find a unused heap where the used memory flag is turned 1 and has enough memory to support the size.

The *void type is a universal pointer type which is the reason why when one writes malloc one has to cast what malloc returns, otherswise it returns raw memory.



Byte arithmetic

When wanting to move around in memory with precision use char* since they are always 1 byte so in any calculation with them you will always move or do the thing once.
Example in this one:

```
struct heap* next_block = current_block + current_block→size;
```

This looks perfectly fine at first but there is one problem. We are adding the size to the current_block but doing that causes c to automaticaly multiply sizeof(struct heap) with the size.

A example of this is as follows:
current_block is at address 1000

current_block → size is 200

sizeof(struct heap) is 8

Then current_block + current_block → size gives us 1000 + (200 * 8) = 2600
The reason being we move 200 struct big steps in memory in the eyes of c so we have to correct it.
Luckly we can fix it with Byte arithmetics as seen below:

```
struct heap* next_block = (struct heap*)((char*)current_block + current_block→size);
```

The heap is now created with a malloc function, congratulations now we only need to fix the free with coalition.

FREE

Free does exacly what it says it frees the memory from the heap and sets it ready to be used. But to optimize it we can add free memory next to each other together to make a larger free memory to use and split up.

When we free a memory we go left until we find a memory block that is allocated with the flag = 0 and we stop there. Then we create a variable that will hold the total size and we go right merging the free blocks giving the old heap headers a magic number or type.