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

```
// Takes a 16-bit port number EDX and 8-bit value EAX to send to that port.
void outb(uint16_t port, uint8_t value) {
    __asm__ volatile ("outb %0, %1"
    :
    : "a"(value), "Nd"(port));
    }
// "outb" %0, %1: Sends the value from EAX into the port located in EDX
// "a" (value): put the value parameter into EAX register
// "Nd" (port): Put port into EDX register or user immediate value


// Takes a port number, returns the byte from that port.
uint8_t inb(uint16_t port) {
        uint8_t result;
    __asm__ volatile ("inb %1, %0"
        : "=a"(result)
        : "Nd"(port));
        return result;
        }
// "inb %1, %0": Reads the value from the port specified in EDX, stores result in EAX
// "=a": Stores the result in EAX register
```

// (result): Copies the EAX value into our result variable.
// "Nd" (port): Put port into EDX register or user immediate value

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:

```
#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
```

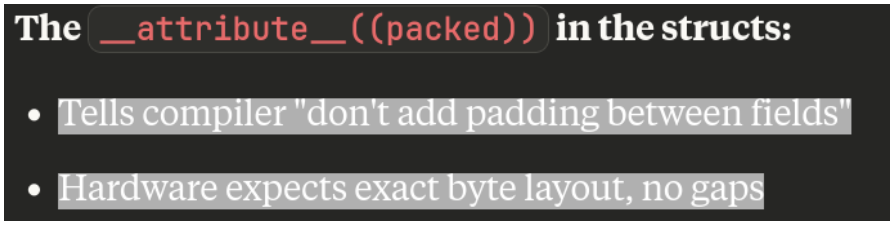
IDT

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

```c
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:

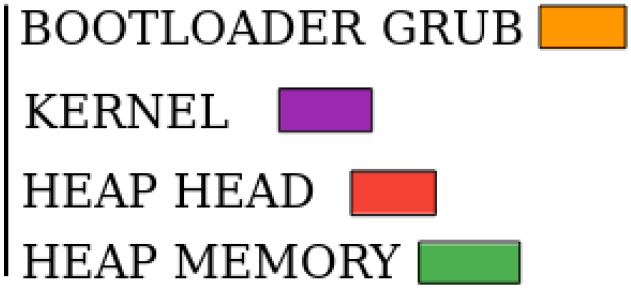
_end = .;

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. 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);
```

In the malloc if we make the size hold the entire block including the header then we can simplify the Byte Arithmetic making it less error prone.

The heap is now created with a malloc function, congratulations now we only need to fix the free with coalition.
Problem that was found with the implementation was that we used 1024*1024 for the heap which is fine, But when the program started and the debug was printed it said size 65,536, then when I made

the size just 1024*1024 directly it showed 0. The problem was the program used uint16_t which just looped back and 1,048,576 % 65,536 = 0. When changed size from uint16_t to uint32_t the problem was solved and it now works.

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 = 1 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.

The magic number 555555 will be ignored. So in a simple way we just traverse the memories that are available and merge them together adding there size to the left most block. A heap block does not stop existing we just ignore them when we give them the magic number 555555. Then we just check for the magic number in kmalloc so it ignores them when looking for memory and kfree tells the user it is not okay to free a dangling pointer and then returns stoping the free command.

The head size law

Never and I repeat never create a memory block that is less than the size of the head. For example in our kernel the heap head is 16bytes and can be found by writing sizeof(struct heap).

WARNING: important never assign a variable pointer to a address or calculated address without knowing 100% that it is inside of the heap and not outside.

Normaly a struct heap has to be aligned manualy but in this situation the header of our heap is padded by the compiler to get 16-byte length which gives us alignment.

Ternary Syntax

Now that we have a more complete kernel we would want to debug and an good and quite simple way to do it with as little codes as possible is using Ternary sytax.

They have this sytax:
condition ? value_if_true : value_if_false
for our debuger of heap we used:

current_block->magic == -2 ? 'G' : current_block->flag == 1 ? 'F' : 'A'

Safe space

When creating things such as heaps or other functionality make sure you have a robust and strong gate before fixing the initial logic. The better gates even if inefficient at first will stop your entire kernel from crashing or getting stuck. This will make it much easier to debug and fix your logic errors without having to think about if it came from the outside. If you have strong gates then it is just logic flaws and not from the outside. In short it gives your a sandbox where only your logic may cause problems not the input.

Debug for kmalloc

The magic numbers type is not to be underestimated. They let everything through because we converted from define which is a int to uint8_t which causes the value entered if higher than 255 to loop around and become in this situation 53. This destroyed my entier loop and caused 0, and even negative numbers to appear in a long unlogical loop. After the fix to change it to uint32_t in the h file, the bug dispersed.

Process

A process is a screenshot of what the CPU is doing at any moment. When the kernel runs we have multiple registers, it has stacks and such. A process is all that at the moment. So We can save all that information on the heap and then load another from the heap and run that process like nothing has changed. We essencialy stoped time for the process and started another and so on. We switch between them with ms difference.

Important notes:

Think about what the CPU needs to know to resume execution. We will have to save register states, flags, pointers, stack points and such.

Your first design decision:

How do you want to represent a process? You'll need a data structure that holds all the important CPU state. Think about what the CPU needs to know to resume execution:

- All the general-purpose registers (EAX, EBX, ECX, etc.)
- Stack pointer (ESP)
- Instruction pointer (EIP) - where to continue executing
- Maybe some flags

The most tricky part:

The tricky part will be how and when I will save the information of a process to the heap. The processes will be running while I save them.

List and description of important values and registers to save

EAX-The Accumulator

The EAX registry was covered in the start of this document but here we will deep dive more into what they do and why they are important to save, for switching.

One can call the EAX the CPU's favorite register, when ever a return happens or a calculation happens then it is most likely stored in the EAX or accumulator. In a calculation (a / b) + f when we switch we might be halfway through the calculation only having done a/b then that would be found in EAX, so if we don't save EAX it would then start at that calculation where we left but not keep the a/b ruining the calculation entierly.

EBX-The Base Register (The Address Keeper)

EBX often is used to hold memory addresses or important data that needs to stick around. It can be thought of as a base reference point. If the process is working with an array then EBX might hold the address of the first element, Or if we are processing a string then the EBX might hold the address of the first position of the string. It helps the CPU remember where important data lives in memory. If the data in EBX is lost then the CPU lose where the important data is located when the process is switch back.

ECX-The Counter (The Loop Master)

The ECX is the CPU's built in counter. When the program runs a for loop that runs 150 times, ECX often keeps track of how many iterations have been completed. It's the register that answers "How many more times do I need to do this?" that the CPU asks. The importance is that if we switch in the middle of a loop then the ECX data will tell the process exacly where it left off.

EDX-The Data Helper (The Versatile Assistant)

EDX is EAX's helper, When the process multiply two large values the result might be too big for just the EAX so it spills over into EDX. It is used for I/O operations when talking to hardware devices. EDX often holds port numbers or data, In summary it is a overflow registry that keeps the important data that is larger than EAX. One can think it like the CPU's way to say "here's the extra stuff that didn't fit elsewhere."

ESI-Source Index (The Data Finder) and EDI-Destination Index (The Data Placer)

ESI points to where data comes from. If a scentance is copied from one part of memory to another then the ESI points to the original scentence. This behavious is called "source index" because it points to the source of data. If we are copying from 1000 to 2000 in memory the word "HELLO"

The ESI starts at 1000 and then reads H.

EDI is the oposite of ESI – it points to where data goes to. EDI starts at 2000 in this example and will write H at 2000.

1000 ESI → “H” read

2000 EDI → “H” write

1001 ESI→ ”E” read

2001 EDI → “E“ write

...

This is important since lets say we want to copy “HELLO” and we only get to “HE” as in our example, well then we would know from the EDI and ESI that we just copied E which is 1001 ESI and 2001 EDI.

ESP – Stack Pointer (The Stack Tracker – CRITICAL)

ESP holds extreamly important data and we need to save It. It points to the top of the process’s stack. The stack is where all the variables, arrays and such are stored. If one loses ESP then the process loses the track of all local variables, and can’t even go back from a return.

EBP-Base Pointer (The Function Tracker)

EBP helps track function call boundaries. When the process call a function EBP points to the start of that function’s area on the stack. It saves where the functions begin. Very important if your process uses functions.

EIP-Instruction Pointer (The Program Counter – CRITICAL)

EIP is the most important register for process switching. It points directly to the next instruction the CPU should execute. Really important.
If we have the process runing:

int a = 2;

int b = 3;

kprintf(“%d + %d = %d”, a, b, a+b);

If we stop right after int a = 2 and stop then the EIP will know that when we switch back it should now start to run int b = 3. Not having to redo the int a = 2 instruction.

Common inline assembly constraints

“a” = EAX

“b” = EBX

“c” = ECX

“d” = EDX

“S” = ESI

“D” = EDI

Syntax for asm

One could say the baseline inline assembly syntax template is as follows:
asm(“assembly instruction mov, ect” : output constraints : input constraints)

“assembly instructions” actual assembly code like mov, add, etc

output constraints where the results go (What gets written to)

input constraints where inputs come from (What gets read FROM)

To write to the array we have created to save registers we use this code:
example eax:

```
asm ("mov %0, %%eax " : "=m" (registers.registers[EAX_INDEX]));
```

ams: we are using assembly.
Mov: we are moving something.

%0: we have one constraint either output or input.

%%eax: The eax registry.

Since we only have one : colon then we want to output.
=m means TO a memory.

Then we choose our memory.

To save to the register from our memory:

```
asm ("mov %0, %%eax " : : "m" (registers.registers[EAX_INDEX]));
```

now we switch the %%eax and %0.
Also we have : : showing we want to input from the memory we chosen.

The hard register

The hard register is the EIP register. What is hard is that when you run a asm code to get the EIP the EIP is at that asm code runing it so it can't get itself from there. No to get it there are many different ways but one way is the so called call way.
What we do is this:

```
c

asm("call 1f\n"
    "1: pop %0"
    : "=r" (registers.registers[EIP_INDEX]));
```

This asm does this:
Call 1f – calls the label "1" which is the next line after the asm.

This will make a return address get pushed onto the stack.

1: the label (landing spot)

pop %0 pops that return address into your variable

What happens is this. We call the label with the name 1: we do f because the label is below the call. Since we call it acts like a function and the address is sent to the stack. We immediately then pop the top of the stack (The address) into our registers.

We then capture it with =r and throw it into our registers. The reason behined =r is because it is for general cases and since it goes into the stack and then we get it we can't use =m here.

Open the Timer flood gates

To open the flood gates we need to make a interrupt handler and connect it to IRQ 0. Change the interrupt.s file to create a wrapper for 32. Then in the .h file for interrupt we extern the wrapper and lastly we then with this code: `set_idt_entry(32, (uint32_t)isr_wrapper_32, cs, 0x8E);`

set the wrapper 32 to be position 32 in our IDT which maps to IRQ 0. Reversed actualy but the logic is alike. Reversed as in IRQ 0 becomes 32 in our IDT.
Then we need to configure the PIT.

The PIT

The PIT is used to configure things about the timer interrupt.
The PIT will be used to make the interrupt appear with a interval of 10ms.

To do this we have to send 0x36 into port 0x43, with this we Configure the PIT to send square wave pulses and ready for change.
We then have to pass a 16-bit value into port 0x40. For reasons only the architect of the CPU knows we have to send the divisor as first the bottom 8-bits and then the top 8-bits which can be found this way.
We can find the bottom 8-bits by doing (divisor & 0xFF)
We can find the top 8-bits by doing (divisor >> 8) & 0xFF)
0xFF aka 0x00FF = 0000000011111111

The divisor we will be using is 11932 this will give us 100 interrupts per second or 1 interrupt per 10ms.

Problem with EIP

A problem may occur if we don’t load the EIP in the interrupt_handler is that we might get a interrupt call mid registry saving, causing corruption and errors potensialy full system crash.
To solve this we put the entier save and load in the timer_interrupt_handler which will stop all other interrupts from fiering since the PIC’s will wait for the EOI (END OF INTERRUPT). Then the CPU wont take interrupts until EFLAG is set which is when we return to our EIP with iret.

How the asm instructions for EIP, CS, and FLAGS work

We save flags by doing this:

```
asm volatile("pop %0" : "=m" (registers.reg[EIP_INDEX]));

asm volatile("pop %0" : "=m" (registers.reg[CS_INDEX]));
asm volatile("pop %0" : "=m" (registers.reg[EFLAGS_INDEX]));
```

Since the CPU is pushing up the EIP, CS and EFLAGS in this order:
EFLAGS

CS

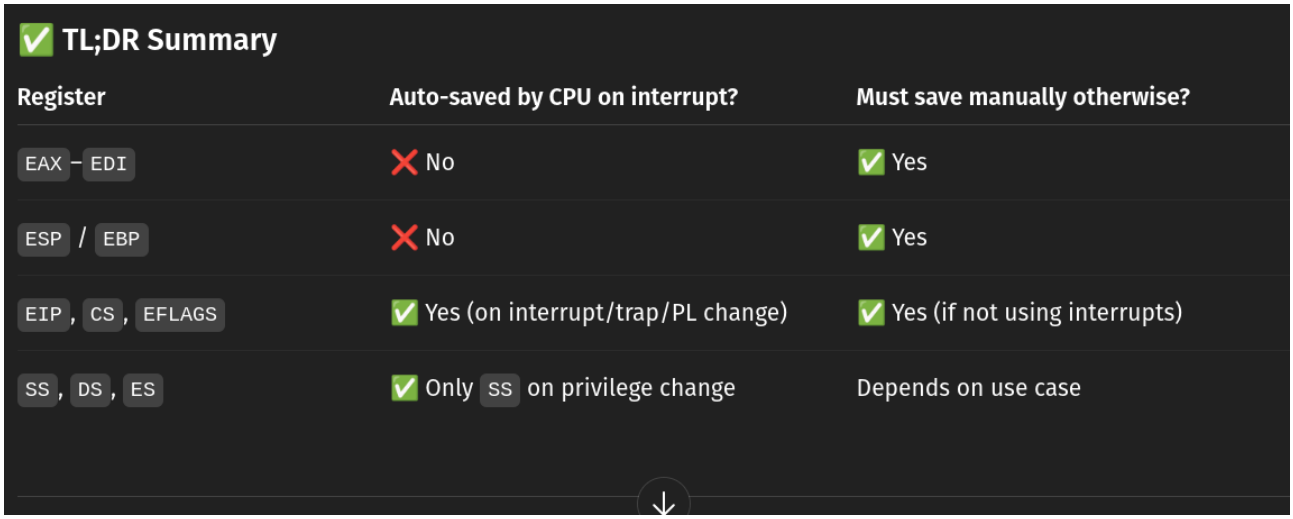
EIP

We have to pop them into out register which we can do when saving to save on instruction overhead. This also gives us a safe space beneath where we can work on load without the fear of colliding with the CPU pushed values. We instead replace them with our values.
When loading aka replacing with our values we will simply push in the opposite order since the stack is LIFO.

We will do:
push EIP

push CS

push EFLAGS



Important to know that the CPU pushes the EIP, CS and EFLAG onto the stack when a interrupt is called. The EIP is the return address to start where we called/interrupted to continue from there.

How to controll the EIP, CS, and EFLAG

The EIP, CS and EFLAG is send to the stack by the CPU but if we want to give another process the wheels we want to replace those values with out own saved values. We can do this by first poping 3 times when saving, removing the 3 top stack items which will be EIP->CS->EFLAGS
But how push works with LIFO we have to push our values in reverse so first we push EIP then CS and then EFLAG. Then when the iret is ran the CPU will pop the values in the order and start from the new EIP, CS, and EFLAG values. We essencialy just switched out the values without the CPU knowing since we made the stack look the same when we returned it to the CPU's controll.

IMPORTANT:
IF you have a assembly wrapper then do not run "iret" or "ret" in your c interrupt handler. The problem is it will try to run iret but the wrapper itself will also run iret causing a problem. It will never meet it's end and run the last part of the wrapper.

In the assembly wrapper we also has something named POPA it will remove all your progress restoring the register values and saving them over the once you just loaded destrying everything sadly. But if we remove it then we can save it and keep the changes after.
NOPE POPA is Important I was wrong. POPA saves from the stack which we replace before so it will replace but with the same values. The function wont run without POPA in assembly.

I was wrong!

To make a switch we only need to save and load the esp nothing more, the EAX, EIP, and so on. The reason is that all of those registers are already included in the stack which the ESP points to. But also in the PCB we should include the PID and the state what id it has and if it is running or not

Framebuffer

Offset

Offset when we talk about struct is more or less how far from the first memory of that struct.
So to explain more in detail here is a picture:

```c
struct PCB {
    uint32_t saved_esp;  // This is at offset 0 (first field)
    uint32_t pid;        // This would be at offset 4 (second field)
    uint32_t state;      // This would be at offset 8 (third field)
};
```

As we can see above we start with the saved_esp which is the first field / memory of the struct, there fore it is offset:ed by just 0, but if we look at the one bellow we see that PID is at offset 4 and then state is offset at 8 and so on.

We can mathematically see that offsets can be calculated by this equation: x*4.

Pointer de referencing in C

So normal pointer de referencing in c is to essentially just access the address the pointer points to.
So for example de referencing in c can look like this:
current_process→saved_esp = some_value;
We are de referencing the pointer to get to our saved_esp value and then we assign that value to some_value

Pointer de referencing in Assembly

Now depending on who it is assembly can be easy to understand or confusing, but we will take it slow on the explanation

It essentially just means that we do what we did in C but we do it in the assembly wrapper instead, so say we attempt to access the value saved_esp in assembly. How do we do it? Well we use two methods I would call it. What we do is we first make sure the pointer variables we create are accessible to our assembly script, we do this by writing extern. So as an example:
extern struct PCB* current_process;
important to note that static shall not be used here!

In assembly the de reference works by writing () around the variable, like opening a present. Now that we have the extern variable and the de reference characters we can begin to work in assembly,
mov eax, [current_process] # this means move the address of current_process to eax.
mov [eax + 0], esp # Here we then keep eax open and add on our offset of where saved_esp is. If we wanted to instead use PID then we would do mov 4(%eax), esp but we don't want move PID into esp register so we will stick with mov (%eax), esp.

IMPORTANT TO UNDERSTAND: We do 2 de references in assembly, so first we do mov eax, [current_process] this is like reading a paper finding the address, then we do mov (%eax), esp The second de reference (%eax) here is that we have gone to the address the paper pointed to and can now enter the house and begin to pack the furniture into a truck/save to the PCB

Load and store ESP with assembly

Great now that we know about de referencing and offset, we can begin to load and store in assembly.

We begin by doing what we did before, create a extern pointer variable like our current_process and then we go to assembly.
In assembly we then begin to de reference and assing/mov:

movl current_process, %eax

Here we have saved the current_process address into the eax register. We now want to store the current esp registers value into the current_process, so that we can load next_process with another esp.

movl esp, %eax

This code above is the same as doing:
current_process → saved equals esp register.
We don't do it in c since it can corrupt other registers and such, assembly is the safest and most recommended method.

Now to load the next_process which is also a extern into the esp to use after having saved the old one.
We begin by moving the process we want to save into the eax as we did before:

movl next_process, %eax
movl esp, (%eax)

We move the address once again into the eax but this time we move the (%eax) aka the saved_esp into the esp register instead, this make program now start then new process with all it's stack values.

IMPORTANT: I found the problem that has been haunting my process code. NEVER EVER ASUME THE CS. IT WILL CAUSE PROBLEMS. ALWAYS GET THE CURRENT CS FROM THE CPU.
THE CODE FOR THIS IS:

```
uint16_t actual_cs;
__asm__ volatile("mov %%cs, %0" : "=r"(actual_cs));
```

New Idea for the process switching, instead of keeping It open and only in the timer interrupt function we can instead make it a function we can call on at multiple places to run, so for example when we run our to be made sleep() function we will call a switch in it.

We will use clobber list to tell the compiler that the new eax and ebx will be trash values so they save the old once.

Will use system interryot 0x80 to call differnet system calls depending on eax and ebx.