Kernel Development Knowledge Documentation

Contents

Chapter 1

Kernel Syntax and Code

1.1 Video Memory

Video Memory or VM is an 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 address `0xB8000` which is where the VM is located in the RAM.

1.1.1 Important Calculation

To find the current cursor position where 24 is current row and j is current column:

$$\text{Position} = (24 \times 80 + j) \times 2 \tag{1.1}$$

1.1.2 Color

The text and background color is chosen with the color code `0x00` where `0x` just stands 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 an even position. If the position is not even then it is instead what text shall appear there. So what char shall be put there.

1.2 Bitwise Operations

1.2.1 Left Shift

Left shift means that we have lets say `bg << 4` this is a left shift operation.
This takes the background color and shifts all its bits 4 positions to the left.
If bg is 3 (binary: 0011), shifting left by 4 gives 00110000.

1.2.2 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.

1.2.3 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 code for kernel development.

1.2.4 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!

1.3 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_start` 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.

```
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
```

1.4 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

1.4.1 AL

AL is part of EAX.

- AL = lower 8 bits of EAX
- AH = next 8 bits of EAX
- AX = lower 16 bits of EAX

`outb` only uses the AL part of EAX.

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

1.5 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"` use 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 an optimization to not generate unnecessary long instructions.

1.5.1 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

1.6 Outb and Inb

These are used to write a byte to and read a byte from a hardware port.

The process is described below.

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 use 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 use immediate value
```

Key points:

- `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`

1.7 BCD to Decimal/Binary

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

Let's say BCD is `0x23` then we have in binary 0010 0011, well if we go through with the equation above:

- 0010 0011 $\rightarrow$ 0000 0010 $\rightarrow$ 2 (in decimal) $\times$ 10 $\rightarrow$ 20
- 0010 0011 & 0000 1111 ($0x0F$) $\rightarrow$ 0000 0011 $\rightarrow$ 3 (in decimal) $\rightarrow$ 3
- Now in the end stage we simply add them together and we get $20 + 3 = 23$ and then return it.

The BCD cannot take in values higher than 0-9 so a, b, c, d, e, f cannot 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 an 8 bit format which would undermine the point of the BCD system.

1.8 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 follows:

```
#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:

```
#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 makefile or in my case the `run.sh` file since I manually 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
```

Chapter 2

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 needs to know where your interrupt handler functions are in memory. Without the IDT the CPU would have no idea what to do when an interrupt occurs.

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

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 our interrupt table.

2.1 Usable Interrupts

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

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

2.2 IDT Entry

2.2.1 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.

2.2.2 Selector

Selector points to which code segment the handler runs in. For kernel code this typically is `0x08` (first entry in GDT).

We'll use `0x08` for all our handlers.

2.2.3 Flags

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

We'll use `0x8E` (Present, ring 0, 32-bit interrupt gate)

2.2.4 Magic Numbers?

The number `0x08` is standard kernel 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 (least 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.

2.3 IDT lidt

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

This code above is 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.

2.4 Important IDT Note

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

2.4.1 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.

```
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;
}
```

The `__attribute__((packed))` in the structs:

- Tells compiler "don't add padding between fields"
- Hardware expects exact byte layout, no gaps

Chapter 3

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.

3.1 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.

3.2 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.

3.3 Sequence of Interrupts

When an interrupt happens this is 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 conform it to instead tell the CPU "Hey, instead of at place 12 we want you to go to place $12 + 32 = 44$ in our IDT instead."

The reason why we have the functions `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 off 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 control then all 16 IRQs would fire constantly.

Chapter 4

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 won't collide because it will constantly be at the end of the kernel.

We can do this by entering:

```
1 _end = .;
```

Chapter 5

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 and 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 separate it making 2 blocks, then `malloc` again and we have 3 blocks and so on until the giant heap has been separated into small usable memory blocks. Which is why it is dynamic.

So `kmalloc` will look through this linked list later when built and find an 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, otherwise it returns raw memory.

5.1 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:

```
1 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 automatically multiply `sizeof(struct heap)` with the size.

An 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 $1000 + (200 \times 8) = 2600$

The reason being we move 200 struct big steps in memory in the eyes of C so we have to correct it. Luckily we can fix it with Byte Arithmetic as seen below:

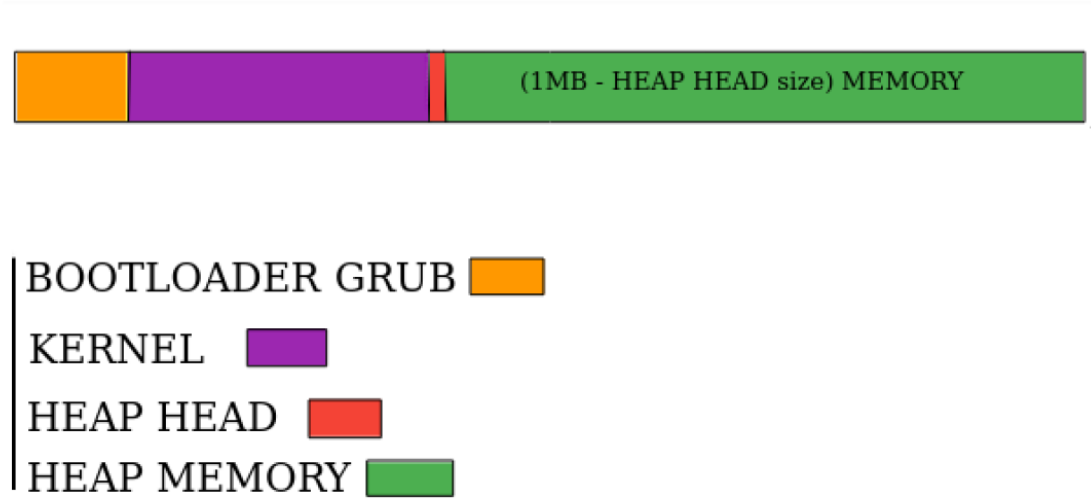

Figure 5.1: Caption

```
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 coalescence.

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.

IMPORTANT: Since the cpu only likes to speak and work with multiples of 8. We should to make it more optimized return a multiple of the users request back to them as a allocated block. So for instance if the user requests 10 Bytes then the malloc should take the input from the user and make the size request a multiple of 8, so in this scenario it would make it 16 Bytes since the user requested 10. This actual makes it better for us, Since now the size will always be a multiple of 8 the 3 first bits of size are fully available as flags and such. Since in binary 1000 is 8. This means that since we do multiples of 8 it will never be less than 8 so the last 3 0's will never be used by the size. SO we can instead use those for flag and magic.

```
uint32_t full_size_request = (user_inputer_size + sizeof(struct
    heap));
full_size_request = (full_size_request + 7) & ~7;
```

Lets show our example here:

- The user wish to malloc(10);

- 10 in binary is: 00001001
- We then add 7 (00000111) to make sure the nearest multiple to 8 is higher than our requested size.
- When we add them we get 10 + 7 = 17 which in binary is: 00010001
- Now the right side 7 means the invers of 7. So since 7 is (00000111), the inverse becomes (11111000).
- Lastly we compare them with and and get the following: 00010001 & 11111000 = 00010000
- Now we have calculated that we need 16 Bytes to fill the requirements from the user and cpu.

5.2 FREE

Free does exactly 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 their size to the leftmost 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 stopping the free command.

5.3 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 16 bytes and can be found by writing `sizeof(struct heap)`.

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

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

5.4 Ternary Syntax

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

They have this syntax:

```
condition ? value_if_true : value_if_false
```

For our debugger of heap we used:

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

5.5 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 you a sandbox where only your logic may cause problems not the input.

5.6 Debug for kmalloc

The magic numbers type is not to be underestimated. They let everything through because we converted from `define` which is an 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 entire 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.

Chapter 6

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 essentially stopped time for the process and started another and so on. We switch between them with ms difference.

6.1 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.

6.1.1 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.

6.1.2 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

6.2 List and Description of Important Values and Registers to Save

6.2.1 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. Whenever 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 entirely.

6.2.2 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 loses where the important data is located when the process is switched back.

6.2.3 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 exactly where it left off.

6.2.4 EDX - The Data Helper (The Versatile Assistant)

EDX is EAX's helper. When the process multiplies 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 an overflow registry that keeps the important data that is larger than EAX. One can think of it like the CPU's way to say "here's the extra stuff that didn't fit elsewhere."

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

ESI points to where data comes from. If a sentence is copied from one part of memory to another then the ESI points to the original sentence. This behavior 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 opposite 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 let's 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.

6.2.6 ESP - Stack Pointer (The Stack Tracker - CRITICAL)

ESP holds extremely 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.

6.2.7 EBP - Base Pointer (The Function Tracker)

EBP helps track function call boundaries. When the process calls 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.

6.2.8 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 running:

```
1 int a = 2;
2 int b = 3;
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.

6.3 Common Inline Assembly Constraints

- `"a"` = EAX
- `"b"` = EBX
- `"c"` = ECX
- `"d"` = EDX
- `"S"` = ESI
- `"D"` = EDI

6.4 Syntax for asm

One could say the baseline inline assembly syntax template is as follows:

```
asm("assembly instruction mov, etc" : 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 for EAX:

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

- `asm`: 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.

6.5 The Hard Register

The hard register is the EIP register. What is hard is that when you run an asm code to get the EIP, the EIP is at that asm code running it so it can't get itself from there. To get it there are many different ways but one way is the so called call way.

What we do is this:

```
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 behind `=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.

Chapter 7

Open the Timer Flood Gates

To open the flood gates we need to make an 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:

```
1 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 actually but the logic is alike. Reversed as in IRQ 0 becomes 32 in our IDT.

Then we need to configure the PIT.

7.1 The PIT

The PIT is used to configure things about the timer interrupt. The PIT will be used to make the interrupt appear with an 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.

7.2 Problem with EIP

A problem may occur if we don't load the EIP in the `interrupt_handler` is that we might get an interrupt call mid registry saving, causing corruption and errors potentially full system crash.

To solve this we put the entire save and load in the `timer_interrupt_handler` which will stop all other interrupts from firing since the PICs will wait for the EOI (END OF INTERRUPT). Then the CPU won't take interrupts until EFLAG is set which is when we return to our EIP with `iret`.

7.3 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 our 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
```

Register	Auto-saved by CPU on interrupt?	Must save manually otherwise?
EAX - EDI	× No	Yes
ESP / EBP	× No	Yes
EIP, CS, EFLAGS	Yes (on interrupt/trap/PL change)	Yes (if not using interrupts)
SS, DS, ES	Only SS on privilege change	Depends on use case

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

7.4 How to Control the EIP, CS, and EFLAG

The EIP, CS and EFLAG is sent to the stack by the CPU but if we want to give another process the wheels we want to replace those values with our own saved values. We can do this by first popping 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 run the CPU will pop the values in the order and start from the new EIP, CS, and EFLAG values. We essentially 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 control.

IMPORTANT: If you have an 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 its end and run the last part of the wrapper.

In the assembly wrapper we also have something named POPA. It will remove all your progress restoring the register values and saving them over the ones you just loaded destroying 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 won't run without POPA in assembly.

7.5 I Was Wrong!

To make a switch we only need to save and load the ESP nothing more, not 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.

Chapter 8

Context switching and offset

8.1 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:

```
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, therefore it is offset by just 0, but if we look at the one below 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: $x \times 4$.

8.2 Pointer Dereferencing in C

So normal pointer dereferencing in C is to essentially just access the address the pointer points to.

So for example dereferencing in C can look like this:

```
current_process->saved_esp = some_value;
```

We are dereferencing the pointer to get to our `saved_esp` value and then we assign that value to `some_value`.

8.3 Pointer Dereferencing 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 dereference works by writing `()` around the variable, like opening a present.

Now that we have the extern variable and the dereference 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
```

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 [eax + 4], esp` but we don't want to move PID into esp register so we will stick with `mov [eax + 0], esp`.

IMPORTANT TO UNDERSTAND: We do 2 dereferences in assembly, so first we do `mov eax, [current_process]` - this is like reading a paper finding the address. Then we do `mov [eax + 0], esp` - The second dereference `[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.

8.4 Load and Store ESP with Assembly

Great! Now that we know about dereferencing and offset, we can begin to load and store in assembly.

We begin by doing what we did before: create an extern pointer variable like our `current_process` and then we go to assembly.

In assembly we then begin to dereference and assign/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 register's 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_esp = 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 an 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 makes the program now start the new process with all its stack values.

IMPORTANT: I found the problem that has been haunting my process code. NEVER EVER ASSUME 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));
```

8.5 New Idea for Process Switching

Instead of keeping it open and only in the timer interrupt function we can instead make it a function we can call 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 ones.

Will use system interrupt `0x80` to call different system calls depending on eax and ebx.

We use round robin functionality to get our process switching to begin working. Our setup is as below:

```
int current_idx = (current_process - process_lists);
int next_idx = (current_idx + 1) % current_index;
next_process = &process_lists[next_idx];
```

We find our `current_idx` by taking the `current_process` which is a pointer and removing that address from the `process_list` which contains all the processes in an array list. We will then get a number with just the index we are at. A bit complicated but it works. Then we find the `next_idx` by taking the `current_idx` and adding 1 (We are adding 1 because we want the next process index not the current). Then we do modulo to make sure it stays in the bounds. For example if we have 4 processes it should not be able to go to 25 for some random reason, so modulo is a safety check.

8.6 Parameters in asm

To be able to get values into parameters from assembly we can make use of `pushl`.

So what we do is the following:

```
movl %esp, %ebx
pushl %ebx
call timer_interrupt_handler
addl $4, %esp  # realign stack
```

We move the esp into the ebx register. We then push the ebx to the stack. Since this is the latest push it will be taken as the parameter in the function. We call the function and then when it returns back to the assembly we have to add 4 to our esp to realign the stack so the CPU won't cry.

8.7 Clobber and Calling Interrupt

To run the sleep function we make use of clobber and interrupt call. So we first find out that our eax can be accessed by writing `28(%%esp)`. Don't ask why - the CPU has just clogged up the stack and made it so. Lots of testing and printing made it possible to find it. With this code we can move a magic value (code) into the eax register and then call an interrupt to use it, like a system call.

We run assembly in our C code. Here we move the magic number 555555 into our eax which lays at `28(%%esp)` in our stack. I know, weird. But it works. Then we call the timer interrupt with `int $0x20` and in the end we tell the compiler with clobber that eax is a value we will clobber aka write over and use, so it will be trash. Save the value that is currently in eax so it does not disappear.

IMPORTANT: Do not write to `28(%%esp)`. Instead write to `(%%eax)` directly. We will use something I call stack mapping to find our eax without having to manually calculate offset later.

8.8 IDLE Process

After some thinking and working, we can figure out that we need some kind of default process that runs undisturbed and will never sleep. Think like this: if all processes are sleeping what happens then? Nothing runs? No, we should have an idle process at `process_list[0]` that is the start of it all and if everything else is sleeping it will be the only thing that still runs.

Chapter 9

Fork() Spawning Baby Processes

Fork is important because we can spawn baby processes that do multiple things. Think like when we in command terminal do:

```
> ls
```

The command terminal is one process and we want to run `ls` to print out a list of the directory. We create a sub (baby process) that runs the `ls`, moves the values to the buffer or a file and then prints it to the terminal, so the process running the terminal is constantly running just waiting for the baby process to finish and return.

As we can see this is really great and important if we want a responsive and usable system.

Important: the baby will run the same process but it will have a different process id.

9.1 How Return Works in Assembly

In reality when we write `return 5;` at the end of a function the CPU gets this:

```
1 mov 5, %eax
2 ret
```

So what happens is we write our value to eax and then return it. That is it. So we can manipulate it.

Chapter 10

How We Can Map the Stack and Manipulate the Compiler

In the section about offset we talked about how in assembly we have to offset differently depending on where the variable is in the struct. So to iterate:

If we have a struct:

```
typedef struct PCB {
    uint32_t saved_esp;
    struct registers reg;
    uint16_t PID;
    int sleep_time;
};
```

Then to get to `saved_esp` we would have to do an offset of 0 but if we wanted to find the PID for example we would need an offset of 8. Now this does not seem revolutionary to most but if we take it a step further, I here have created a struct registers:

```
typedef struct registers {  // This is a map for the stack.
   Nothing more
    uint32_t edi;      // [0]
    uint32_t esi;      // [1]
    uint32_t ebp;      // [2]
    uint32_t esp;      // [3]
    uint32_t ebx;      // [4]
    uint32_t edx;      // [5]
    uint32_t ecx;      // [6]
    uint32_t eax;      // [7]
};
```

So if I were to try to find the offset of esi in this struct it would be 4 right. $1 \times 4 = 4$ or if I want to find esp it would be $3 \times 4 = 12$. So this one maps the offset manually but we can also manipulate the compiler to calculate the stack's offset for us. To do this I need to go into what casting really does.

10.1 What is Casting and What Happens When We Do It?

Casting is just telling the compiler to handle that variable in memory differently. It is like byte arithmetic we talked about before where because we wanted to move exact steps in the memory we casted our structs to chars that move only 1 byte arithmetically in memory and then cast back to structs to be able to find the data. That is how casting works.

In our registers struct and process handler we do this:

We get the location of the esp from assembly with:

```
1 pushl esp
```

This makes it an argument in the parameters. In our parameter we tell the compiler to look at it like: `uint32_t* stack` because it is an address and just easier or more normal to hold it in a `uint32_t` for me. Then we simply cast it to our registers struct. Look what happens:

The esp will begin here:

`[EDI] [ESI] [EBP] [ESP] [EBX] [EDX] [ECX] [EAX]`

The problem we want to solve is, how do we get to ECX for example? We force the compiler to calculate for us.

The esp will begin here:

`[EDI] [ESI] [EBP] [ESP] [EBX] [EDX] [ECX] [EAX]`

We cast our esp pointer to our registers struct and then we can now just write `reg->just write reg->ecx` and the offset will automatically be calculated since it is in the 6th position in the struct.

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 makes the program now start the new process with all its stack values.

Since it works the same as binary arithmetic the compiler will just jump as far as the struct says. So since we cast to our register struct it is like opening a present. The `[EDI] [ESI] [EBP] [ESP] [EBX] [EDX] [ECX] [EAX]` frame we get the actual position.

I know this is hard but it is really cool!

Figure 10.1: Caption

Chapter 11

Physical vs Virtual Memory

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

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

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

11.1 Page Tables

Page tables make it so that each process gets 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.

Right now in the kernel implementation we write straight to RAM, so if we write to 0x100000, it goes exacly to that cell om the RAM stick, this causes problems for us because we get fragmentation since we might have processes mallocing odd sizes in the heap causing it. It also affects saftey because Process A can accidentaly or maliciously write over Process B's memory. Lastly we want the process to believe it owns the computer so we essencialy lie to them that they have more memory than they have, In reality the entier system can only use 4GB of memory, but we tell each process that they have 4GB of memory each, this essencialy multiplies our memory by lying.

11.2 The great lie

With paging we lie to the program so we give them a virtual address (a fake address) and then the cpu's MMU (Memory Management Unit) automatically translates it to a Physical Address (the real RAM) using a lookup table.

In x86 the 32-bit virtual address is split into three parts, we have the Directory index (top 10 bits) it tells us which chapter of the phone book we are looking at. Then we have table index (middle 10 bits) which page we should look for in that chapter. Then lastly we have offset (bottom 12 bits) which specifies which line to look at on the page.

11.3 The begining

So how do we even start with paging? It looks so confusing and scary. We can first understand how paging is activated. To activate paging we just have to flip a bit (bit 31 in the CR0 register) When that bit has been flipped aka is 1 then we have activated the paging and the cpu will now split up all addresses it is handed in this format: $|Top|Middle|Bottom|$.
It splits it to 10 top bits, 10 middle bits and, 12 bottom bits. The top bits are the Directory index which tells the us which chapter in the phone book we are looking at. The Middle bits is the table index which tells us which page we should look for in the phone book. The bottom bits are used to specify which line to look at on the page.
Together these one translates the virtual address we put in to the real physical address.

11.4 Walk through of a paging processing

Lets say we want to read the address 0xDEADBEEF and hand it to the cpu. The following happens:
The cpu will look at the CR3 register since it holds the physical address of the Page Directory, it says: "Here is the book."
The CPU then to get the Directory index takes the top 10 bits of 0xDEADBEEF and lets say for our sake that it equals 6, the cpu then goes to entry #5 in the Page Directory and entry #5 contains an address, it says "The Page Table you need is located at Physical address 0x20000 for example."
Then the cpu takes out the Table index by taking the 10 middle bits of 0xDEADBEEF and lets say for simlicity again that they equal 50.
The CPU goes to the Physical Address 0x20000 (Page table) and looks up Entry #50
The netry #50 contains an address it says "The actual 4KB block of RAM you want starts at physical address 0x80000."
Then the CPU uses the bottom 12 bits which we say for simplicity equals 200 for example and then the CPU adds the Physical Frame Address (0x80000) + the offset (200) so it will read that the Physical RAM address we want is at 0x80200.

11.5 Before the flip

Before we flip the 31th bit in CR0 to activate paging we need to actualy make the paging tables and the infrastructure, think of it like this, Before we turn on a translation service we have to first tell it which dictionaries to use.

We begin by making a page directory which must be 4KB aligned because the hardware requires it. It needs the least 12 bits to be free to use for reading and such things that the cpu can use and flip. So We make sure it is 4KB aligned, The paging directory address also needs to be alligned and we can solved this by going into the linker and adding the command: . = ALIGN(4096); Which will force the linker to make sure _end is aligned when placed in memory. Then at the start of the kenel right after the idt and heap is setup we allocate the page directory so we are sure nothing else is ruining the heap and making misalignment. We do kmalloc(4096).

CHANGER OF PLANS: The heap and paging should be seperate. So the paging memory should be put at the end of the kernel memory and when it ends we should put

the heap to start.