\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Bootloader development for ARM cortex STM32Fx Microcontroller

* Bootloader:-
* It is nothing buyt a small piece of code stored in the MCU flash or ROM to act as an application loader as well as a mechanism to update the applications whenever required.
* Some micro controller comes with in circuit debugger like STMF series, TI and many more.
* SO the mcu which does not have in circuit debugger, it will always use bootloader to update application or binary when we flashed it with new program. Eg:- arduino. It will always use bootloader to update binaries.
* If there is in circuit debugger then it will use that circuit instead of bootloader. Eg;- STM has cuitcit so it will use that instead of bootloader to update binaries.

Memory organization: -

* Main Memory or Internal flash mem also called as Embedded flash (512kb): -
  1. Begins @ 0x0800\_0000.
  2. Ends @ 0x0807\_FFFF.
  3. Used to store ur app code and vector table ("vector table" in a microcontroller is a data structure that stores the memory addresses of specific interrupt service routines (ISRs)) (“switch table" refers to a data structure within the kernel that acts as a lookup table for device drivers) and ROM of the program.
  4. Non-volatile.
  5. Ther are 8 sub section in this memory from s0 to s3 (16kb), s4 (64kb) and s5 to s7 (128kb). So when we flash the code, it will start flashing or occupying the s0 section and will move towards s7.
* Internal SRAM1 (112kb): -
  1. Begins @ 0x2000\_0000.
  2. Ends @ 0x2001\_BFFF.
  3. Used to store ur app global data and static variables.
  4. Also used for stack and heap purpose.
  5. Volatile.
  6. Also execute code from this mem.
* Internal SRAM2 (16kb): -
  1. Begins @ 0x2001\_C000.
  2. Ends @ 0x2001\_FFFF.
  3. Used to store ur app global data, static variables.
  4. Also can be used for stack and heap purpose.
  5. Volatile.
  6. You can execute code from this memory.
* System mem (ROM) (30kb): -
  1. Begins @ 0x1FFF\_0000.
  2. Ends @ 0x1FFF\_77FF.
  3. Store bootloader in this mem.
  4. By default mcu will not execute any code from this mem but you can config mcu to boot or execute bootloader from this mem. This means that even if you power up the controller it will not execute any code as it will not use bootloader always when power up. So to execute from this section we need to config some boot pins to execute this code.
* OPT (One Time Programmable) mem (528b): -
  1. To store controller name and number or signature.
* Backup RAM (4kb): -

Reset sequences for a mcu

* When you reset the mcu the PC (program counter) of the processor is loaded with the value 0x0000\_0000.
* Then processor reads the value @ mem location 0x0000\_0000 into MSP (Main Stack Pointer).

MSP = value@0x0000\_0000

That means, processor first initializes the Stack pointer regs.

* After that, processor reads the values @ mem location 0x0000\_0004 into PC.
* PC jumps to reset handler.
* A reset handler is just a C or assembly function written by you to carry out any init required.
* From reset handler you call your main() func of the app.
* But as we see the main mem starts from location 0x0800\_0000 and in there the user flashed code is stored so mcu should instead of 0x0000\_0000 should go look for or start from 0x0800\_0000.
* In mcu, whether you look in 0x0000\_0000 or 0x0800\_0000 they both holds the same info. This is because of “memory aliasing”.

Memory aliasing is a way by which the code store on location 0x0800\_0000 is mapped to location 0x0000\_0000 and some of the other location are also aliased.

* It is used to map memory of some location to another location.

Boot loader overview: -

* When the core is reset, the user has the opportunity to direct the core to execute the ROM boot loader or the application in flash mem by using any GPIO signal in ports A-H as configured in the Boot config (BOOTCFG) reg.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Build a multitasking OS and kernel with interactive Shell by [**Daniel McCarthy**](https://uci.udemy.com/user/daniel-mccarthy-13/)

The data can be found at the “osdev.org” page.

Installing need for Real Mode Development

* Command on linux shell – “Sudo apt install nasm”.
  1. Then check the version of nasm using “nasm -v”.
* Cmd – “sudo apt install qemu-system-x86”.
  1. Then check using cmd – “qemu-system-x86\_64”.

Hello World displaying on booting screen:-

* /\*
* First we need to specify the origin so that Os knows the offset needed for the data
* Ideally the origin should be 0x0000 and then the system should jump to the
* designated location in the case of this system is 0x7c00.
* Then we need to specify the how much is our bit system. Here it is 16 bit system.
* Here int is interrupt and 0x10 is the BIOS interrupt location. So we are calling BIOS routine.
* To find which interrupt is at which location we can use Ralf Brown's Interrupt list.
* We can see that
* int 10/ah=0eh - is a video teletypr output
* means display a char on the screen, advancing the ccursor and scrolling the screen as necessary.
* Here we are using jump so that it does not go to boot signature as I am continously displaying 'A' on screen
* Here we are using times cmd as we atleast need to fill 510 bytes of data. Here if the given instructions
* are not enough to fill 510 bytes then it will pad it with 0's.
* Now then providing the boot signature which is 0x55AA, but we are writing here 0xAA55 due to endianess.
* \*/
* /\*
* Now to run this in the system using qemu we will first create a binary file '.bin'.
* For this we will use this cmd - 'nasm -f bin ./boot.asm -o ./boot.bin'.
* (giving ./boot.bin as the bin file name will be boot.bin)
* We can see a binary file having 512 bytes of data.
* To see the asm file data in regs use this cmd in linux terminal
* cmd - 'ndisasm ./boot.bin'
* Now to run on qemu use this cmd
* cmd - 'qemu-system-x86\_64 -hda ./boot.bin'
* \*/
* ORG 0x7c00
* BITS 16
* start:
* mov ah, 0eh
* mov al, 'A'
* mov bx, 0
* int 0x10
* jmp $
* times 510-($ - $$) db 0
* dw 0xAA55
* /\*
* To print custom message on the booting screen we will use the below code.
* \*/
* ORG 0x7c00
* BITS 16
* start:
* mov si, message
* call print
* jmp $
* print:
* mov bx, 0
* .loop:
* lodsb
* cmp al, 0
* je .done
* call print\_char
* jump .loop
* .done:
* ret
* print\_char:
* mov ah, 0eh
* int 0x10
* ret
* message: db 'Hello World!', 0
* times 510-($ - $$) db 0
* dw 0xAA55ff
* Ideally we place our start point 0x0000 and we jump from that to the desired location.
* As sometimes different system understand the code differently as placing org as 0x7c00 and then out desire location code is at the same location as we need the processor to go at that location so instead of this the system will understand to add these both location like 0x7c00 + 0x7c00 and the result will not be proper and will make the system to jump or look at undesired location.
* So, always we make ORG as 0x0000 and then we jump it to the desired location.

How to boot from thee USB stick: -

* You will need the hard drive or pendrive for this and then you need to copy this bin file directly to the usb stick and all the data and fs will be removed from the card and the bin file will be present in the card.
* Used this cmd to move the bin file to the card cmd – ‘sudo dd if=./boot.bin of=/dev/sdb’
  1. Here ‘if’ in the cmd means input file name and ‘of’ in the cmd means output file name.

How to call interrupts: -

* They are like subroutines, but you font need to know the mem address to incoke them.
* They are called through the use of interrupt nimbers rather than mem addresses.
* They can be setup by the programmer. For eg you could set up interrupt ‘0x32’ and have it point to your code. Then when someone does ‘int 0x32’ it will invode your code.
* There are total of 256 Interrupt handlers and each contains 4 bytes (offset:segment) and they are in numnerical order in the table.
* They starts at the absolute address in the RAM i.e 0x00.
* So offset – 0x00 and segment 0x7c0 – interrupt 1 address 0x00;
* Offset – 0x8d00 and segment 0x00 – interrupt 2 address 0x04; and so on
* So the address of interrupt 13 is - 0x13\*0x04 = 0x46 or 76 decimal so 76-77 = offset and 78-79 = segment.

Makefile and text file reading for boot: -

* First we need to install make using this cmd – ‘sudo apt install make’.
  1. In the Makefile we write two dd statements; 1st is for the informing the system to look up for message.txt file to display and the 2nd is for giving 512byte space for padding the message.txt file.
* We also need to install bless to visualize the code in a block file manner.
  1. To install use this cmd – ‘sudo apt install bless’.
* Now open the terminal and give this cmd – ‘make’ and then this cmd – ‘bless ./boot.bin’.

Kernel Memory management functions in C

* 1. [kmalloc](https://www.chiark.greenend.org.uk/doc/linux-doc-3.16/html/kernel-api/API-kmalloc.html) — allocate memory
  2. [kmalloc\_array](https://www.chiark.greenend.org.uk/doc/linux-doc-3.16/html/kernel-api/API-kmalloc-array.html) — allocate memory for an array.
  3. [kcalloc](https://www.chiark.greenend.org.uk/doc/linux-doc-3.16/html/kernel-api/API-kcalloc.html) — allocate memory for an array. The memory is set to zero.
  4. [kzalloc](https://www.chiark.greenend.org.uk/doc/linux-doc-3.16/html/kernel-api/API-kzalloc.html) — allocate memory. The memory is set to zero.
  5. [kzalloc\_node](https://www.chiark.greenend.org.uk/doc/linux-doc-3.16/html/kernel-api/API-kzalloc-node.html) — allocate zeroed memory from a particular memory node.
  6. [kmem\_cache\_alloc](https://www.chiark.greenend.org.uk/doc/linux-doc-3.16/html/kernel-api/API-kmem-cache-alloc.html) — Allocate an object
  7. [kmem\_cache\_alloc\_node](https://www.chiark.greenend.org.uk/doc/linux-doc-3.16/html/kernel-api/API-kmem-cache-alloc-node.html) — Allocate an object on the specified node
  8. [kmem\_cache\_free](https://www.chiark.greenend.org.uk/doc/linux-doc-3.16/html/kernel-api/API-kmem-cache-free.html) — Deallocate an object
  9. [kfree](https://www.chiark.greenend.org.uk/doc/linux-doc-3.16/html/kernel-api/API-kfree.html) — free previously allocated memory
  10. [ksize](https://www.chiark.greenend.org.uk/doc/linux-doc-3.16/html/kernel-api/API-ksize.html) — get the actual amount of memory allocated for a given object
  11. [kstrdup](https://www.chiark.greenend.org.uk/doc/linux-doc-3.16/html/kernel-api/API-kstrdup.html) — allocate space for and copy an existing string
  12. [kstrndup](https://www.chiark.greenend.org.uk/doc/linux-doc-3.16/html/kernel-api/API-kstrndup.html) — allocate space for and copy an existing string
  13. [kmemdup](https://www.chiark.greenend.org.uk/doc/linux-doc-3.16/html/kernel-api/API-kmemdup.html) — duplicate region of memory
  14. [memdup\_user](https://www.chiark.greenend.org.uk/doc/linux-doc-3.16/html/kernel-api/API-memdup-user.html) — duplicate memory region from user space
  15. [\_\_krealloc](https://www.chiark.greenend.org.uk/doc/linux-doc-3.16/html/kernel-api/API---krealloc.html) — like krealloc but don't free *p*.
  16. [krealloc](https://www.chiark.greenend.org.uk/doc/linux-doc-3.16/html/kernel-api/API-krealloc.html) — reallocate memory. The contents will remain unchanged.
  17. [kzfree](https://www.chiark.greenend.org.uk/doc/linux-doc-3.16/html/kernel-api/API-kzfree.html) — like kfree but zero memory
  18. [get\_user\_pages\_fast](https://www.chiark.greenend.org.uk/doc/linux-doc-3.16/html/kernel-api/API-get-user-pages-fast.html) — pin user pages in memory

Steps to a booted system:-

* 1. Bios is executed from ROM.
  2. The bios loads the bootloader into address 0x7C00.
  3. The bootloader loads the kernel.
* The bootloader is a small prog responsible for loading the kernel of an OS and they are generally small. And put the system into 32 bit protected mode and then executes.
* The CPU rexecutes instruction directly from the bios rom. Then bios loads its self into ram then continues from ram. It also initialize essential hardware.
* The bios looks for a bootloader to boot buy searing all storage mediums for the boot signature “0x55AA”. When it finds this signature it loads the section of this memory to 0x7c00. And process jumps to this address and begin executing OS bootloader.

REAL MODE:-

* Real mode is a compatibility mode that all modern interl precessors start in when they are switch on.
* In this mode we can only access 1 MB of RAM, it dies not matter if you have GBs of Ram.
* In this mode you have no security, so in todays tech, when we run a code if invalid mem is accessed it showed error and saves the system, but in real mode there is no safety and my curroput the system.
* So, now a days still all x86 CPUs still start in real mode when powered up, the transition to protected mode happens very quickly during boot process, making it negligible. This is still there because to support backwards compatibility so that it can support older software designed for booting.
* X86S will not use real mode. All systems having UEFI (Unified Extensible Firmware Interface) does not have real mode.

Protected mode:-

* Here it is a 4GB addressable memory of instead of 1MB ram of real mode.
* We gain access to 32 bit inst and can easily work with 32 bit regs.
* For interrupt, in real mode it was ‘Interrupt vector table’ and for protected mode it is ‘Interrupt descriptive table’.
* Now to write the boot in protected mode we will do the following
* To start the processor in the protected mode after the real mode.
* In the protected mode now we cannot read from the message.txt as we did in the real mode.
* To read the message.txt we will need disk driver to read and if we try to read using biffer like earlieer time bad things will happen.
* To use gdb, first install using this cmd - 'sudo apt install gdb'.
* Then use this cmd to run the prog with gdb debugging diretly write 'gdb' then enter.
* As you enter, you wiil enter in gdb mode and there you will write in gdb terminal is 'target remote | qemu-system-x86\_64 -hda ./boot.bin -S -gdb stdio' then type 'c' to continue and you can see the prog successful.
* In the terminal the hold(ctrl +n+n+c) then type layout asm to visualize the code of assmebly. type 'info registers' to get info of regs.
* Remember the BIOS only loads the first sector of the memory, so to load other mem sector we need to implement our own disk driver.

Enabling A20 line: -

* The A20 address line is the physical rep of the 21st bit of any mem accesss.
* So, to access and read mem beyond 20th bit we need to enable it.

Creating a cross compiler so we can code in C

* We cannot use GCC compiler as it is targeted for Linux, so we will create a cross compiler.
* So write ‘osdev cross compiler’ in browser and open the link and find the required dependencies to install.
* These are the dependencies, so to download it use this:
  1. Sudo apt install build-essential
  2. Sudo apt install bison
  3. Sudo apt install flex
  4. Sudo apt install libgmp3-dev
  5. Sudo apt install libmpc-dev
  6. Sudo apt install libmpfr-dev
  7. Sudo apt install texinfo
  8. Sudo apt install libcloog-isl-dev
  9. Sudo apt install libisl-dev
* Now download the source code from ‘Binutils website’. In this site go to ftp link and download ‘binutils-2.35.tar.xz’ released in July.
* Now download the source code from ‘GCC website’. There go to ‘mirrors’ and select ‘France – [ftp.lip.fr](ftp://ftp.lip.fr)’. This will tell you to download ftp protocol, so instead copy the link and paste it in the url section, remove the ftp and add https:// instead. Now go to releases and find ‘gcc-10.2.0’ and there select ‘gcc-10.2.0.targz’.
* Extract both the folder in the home. There create a folder name ‘src’. Move extracted ‘binutils’ file/folder to src folder and move to home folder and write this cmds.
  1. Cmd – ‘export PREFIX=”$HOME/opt/cross”
  2. Cmd – ‘export TARGET=i686-elf’
  3. Cmd – ‘export PATH=”$PREFIX/bin:$PATH”’
  4. Cmd – ‘cd $HOME/src’
  5. Cmd – ‘mkdir build-binutils’
  6. Cmd – ‘cd build-binutils’
  7. Cmd – ‘../binutils-2.35/config –target=$TARGET –prefix=”$PREFIX” –with-sysroot –disable-nls –disable-werror’
  8. Make
  9. Make install
* Now we need to do the same things for the GCC so these are the cmds for that, but first make sure the environmental variables are set correctly.
  1. Cmd – ‘export PREFIX=”$HOME/opt/cross”
  2. Cmd – ‘export TARGET=i686-elf’
  3. Cmd – ‘export PATH=”$PREFIX/bin:$PATH”’
  4. Cmd – ‘$HOME/src’
  5. Cmd – ‘which -- $TARGET-as || echo $TARGET-as is not in the PATH’
  6. Cmd – ‘mkdir build-gcc’
  7. Cmd – ‘cd build-gcc’
  8. Cmd – ‘../gcc-10.2.0/configure –target=$TARGET –prefix=”$PREFIX” –disable-nls –enable-languages=c,c++ --without-headers’
  9. Cmd – ‘make all-gcc’
  10. Cmd – ‘make all-target-libgcc’
  11. Cmd – ‘make install-gcc’
  12. Cmd – ‘make install-target-libgcc’
* You can now run your new compiler by invoking this code
  1. cmd – “$HOME/opt/cross/bin/$TARGET-gcc --version”

Creating a build file to compile C code

* To compile our C code instead of asm we need to create this build.sh file so that from now own while using a make cmd we will be using build.sh file and kernel file to create an OS.
* So the asm will just read the first sector of the mem and then all the other reading will be done by other file like kernel.c and so on.
* So we are going to remove the 32 read section from the boot.asm and will place it into own assembly file and then we are going to assemble that to an obj file and then link it into kernel.bin.
* What we will do is we will assemble the bootloader as normal and then compile our kernel.asm file. So, we will move the 32read section from the boot.asm and place it into kernel.asm.
* So Makefile will then assemble that into an obj file, now and ELF file or something and then we will use the linker to create an output binary of the called kernel.bin.
* Then we will dd that into the actual OS file which we call bin. Right now it is called boot.bin.
* So boot.bin and kernel.bin will create OS.bin. and from then out bootloader and kernel are separate.
* Now lets improve Makefile to build both kernel.asm and boot.asm using linker file and ELF file to create a one big OS.bin file which can be directly load it into our system.
* Now use the build script to build one os.bin file and try to run that using this cmd
  1. Cmd – ‘qemu-system-x86\_64 -had ./os.bin’
     1. Here the system will stay in the boot phase as we had set it to jump forever inside boot.asm.
  2. So use gdb to go one by one cmd – ‘gdb’
     1. Here we need to add the symbol files to our files so we will add symbols to the system inside gdb.
        1. Cmd – ‘add-symbol-file ../build/kernelfull.o 0x100000’ then ‘y’.
        2. Cmd – ‘break \_start’.
        3. Cmd – ‘target remote | qemu-system-x86\_64 -S -gdb stdio -had ./os.bin’.
        4. Then press ‘c’ for continue.
        5. Cmd - ‘layout asm’
        6. Cmd – ‘stepi’ to go step by step inside regs.
     2. Now we will be able to write our code in C lang.
* Now we need to align everything as working with both asm and C we need to align them otherwise they will not work properly.
* So we need to add .asm section at the end of kernel.asm and linker script. Because, the first thing which runs is bootloader and then kernel and they both are in asm lang so we do not need to worry about the aligning and when we introduce C we will align them.
* If we align them anywhere except at the end, we might change the instructions uwillingly so we need to align them at the end.

Creating kernel.c and kernel.h files

* Now, to call this files we need to update the kernel.asm, we will add ‘extern kernel\_main’ at the global and we can add kernel\_main in the asm code.
* Here we need to use ‘extern’ word as it is not mentioned in the file but is present in the file which we are going to merge.

Text mode

* You write ASCII chars into mem starting at address 0xB8000 for coloursed display or for monochrome displays address 0xB0000.
* Each ascii char written to this mem has its pixel equivalent outputted to monitor.
* Each cahr takes up two bytes
  1. Byte 0 = ascii char eg ‘A’
  2. Byte 1 = colour code
  3. 0xb8000 = ‘A’, 0xb8001 = 0x00
  4. 0xb8002 = ‘B’, 0xb8003 = 0x00

The logo during booting process:-

* During booting, we can see a logo of Lenovo or the the companies logo, we can do that; we just need to start writing to the mem address, it starts taking input during the boot process.
* In most CPUs the mem address “0xB8000” is the starting place to write the during the booting stage.
* So, calculate the screen size (height and width) and set that as macro or variable in the code.
* Now, suppose the system is 8 bit arch, then the first will take the input you want to place and the next input to displaying which colour you want.
* Eg:-

Void print\_screen(){

Char\* video\_mem = (char\*)(0xB8000);

Video\_mem[0]= ‘A’;

video\_mem[1]= 1;

}

* So the A will be displayed with the colour respective to 1. Here we need to see if the input is little edian or big.
* And also to move to next line, the system does not know that ‘\n’ means new line, so we need to place it in if sentence and then tell the system to go to new line.

Interrupt Descriptor Table

* Describes how interrupts are invoked in protected mode.
* Can be mapped anywhere in memory.
* Different from the interrupt vector table.
* Similarly to the interrupt vector tale the interrupt descriptor table describes how interrupts are setup in the CPU so that if someone causes an ‘int 5’ it will ivode the code for tinterrupt 5 as described by the interrupt descriptor table.
* A screenshot of a computer

  Description automatically generated
* A screenshot of a computer

  Description automatically generated
* A screenshot of a computer

  Description automatically generated
* A screenshot of a computer

  Description automatically generated
* A screen shot of a computer

  Description automatically generated

## **Gate Types**

There are basically two kinds of interrupts: ones that occur when code execution has encountered an [**Exception**](https://wiki.osdev.org/Exceptions) due to bad code, or ones that occur to handle events unrelated to currently executing code. In the first case it is pertinent to save the address of the *currently* executing instruction so that it can be retried, these are called **Traps**. In the second case it is pertinent to save the address of the *next* instruction so that execution can be resumed where it left off. These could be caused by an IRQ or other hardware event, or by use of the **INT** instruction. Another difference to note is that with **Traps**, new interrupts might occur during the service routine, but when the CPU is serving an IRQ, further interrupts are masked until an **End of Interrupt** signal is sent. How a certain interrupt is served depends on which kind of gate you put in the IDT entry.

### Interrupt Gate

An **Interrupt Gate** is used to specify an [**Interrupt Service Routine**](https://wiki.osdev.org/Interrupt_Service_Routines). For example, when the assembly instruction **INT 50** is performed while running in protected mode, the CPU looks up the 50th entry (located at 50 \* 8) in the **IDT**. Then the Interrupt Gate's **Selector** and **Offset** values are loaded. The **Selector** and **Offset** are used to call the **Interrupt Service Routine**. When the **IRET** instruction is performed, the CPU returns from the interrupt. If the CPU was running in 32-bit mode and the specified selector is a 16-bit gate, then the CPU will go in 16-bit Protected Mode after calling the **ISR**. To return in this case, the **O32 IRET** instruction should be used, or else the CPU will not know that it should do a 32-bit return (reading 32-bit values off the [stack](https://wiki.osdev.org/Stack) instead of 16 bit).

### Trap Gate

A **Trap Gate** should be used to handle [**Exceptions**](https://wiki.osdev.org/Exceptions). When such an exception occurs, there can sometimes be an error code placed on the stack, which should be popped before returning from the interrupt.

**Trap Gates** and **Interrupt Gates** are similar, and their descriptors are structurally the same, differing only in the **Gate Type** field. The difference is that for **Interrupt Gates**, interrupts are automatically disabled upon entry and reenabled upon **IRET**, whereas this does not occur for **Trap Gates**.

### Task Gate

A **Task Gate** is a gate type specific to IA-32 that is used for hardware task switching. For a **Task Gate** the **Selector** value should refer to a position in the [**GDT**](https://wiki.osdev.org/GDT) which specifies a [**Task State Segment**](https://wiki.osdev.org/Task_State_Segment) rather than a code segment, and the **Offset** value is unused and should be set to zero. Rather than jumping to a service routine, when the CPU processes this interrupt, it will perform a hardware task switch to the specified task. A pointer back to the task which was interrupted will be stored in the **Task Link** field in the **TSS**.

* And struct for the IDT is
  1. Struct idt\_desc{

Uint16\_t offset\_1; //offset bits 0-15

Uint16\_t selector; // a ode seg selector in GDT or LDT

Uint16\_t zero; // used, set to 0

Uint16\_t type\_attr; // type and attributes, see below

Uint16\_t offset\_2; // offset bits 16-31

}\_\_attribute\_\_((packed));

* Notice we need to pack this struct so that it does not pad itself with the system as it will not work properly if it gets pad.
* Interrupt descriptors are stored in an array with index 0 defining “int 0”, index 1 defining “int 1” and so on.
  1. Struct idt\_des idt\_desc[COS32\_MAX\_INTERRUPTS];
* There is another struct idtr\_desx
  1. Struct idtr\_des{

Uint16\_t limit; //the len of the interrupt des table minus one

Uint31\_t base; //the add of the int des table.

}\_\_attribute\_\_((packed));

* IDT can be defined where we linke in mem. They are setup dfifferently than “Interrupt Vector Table”.
* During an interrupt certain properties can be pushed to the stack. The rules involved with this are quite complicated.

Input/Output: -

* Intel does not provide support to input output functions in C lang so we have to create a io.asm file to manage the io.
* To use proper input hardware like keyboard we will use ISR (Interrupt Service Routine).
* There are many IRQs and each basic hardware is given a specific mem location for IRQ and IRS.
* The start of ISR is from 0x20 and each IRQs takes 1 byte, so 0th IRQ at 0x20, 1st at 0x21, and so on.
* Timer is given 0x20 or 0th IRQ and the keyboard is given 0x21 or 1st IRQ.
* So we will mention this location for the IRQ handling.

Heap allocation and freeing

* Memory limits for a 32 bit kernel, whilst in protected mode we have certain restrictions, the processor is in a 32 bit state.
* As we are running in a 32 bit mode we have access only to 32 bit memory addresses allowing us to address to a max of 4.29GB of RAM regardless of how much system RAM is installed.
* Video and hardware takes up portions of Ram.
* Unused parts are available and an array uninitialized mem is available to us from address “0x01000000”
* Address 0xC0000000 is reserved this means the memory array we have at address 0x01000000 can give us a max of 3.22 GB for a machine with 4GB or higher installed RAM.

Heap implementation

* Our heap implementation will consist fo a giant table which describes a giant piece of free memory in the system. This table will describe which memory is taken, which memory is free and so on. We will call this the “Entry Table”.
* Will have another pointer to a gaint piece of free memory, this will be the actual heap data its self that users of “alloc” can use. We will call this the “data pool”. If out heap can allocate 100 MB of RAM then the heap data pool will be 100 MB insize.
* Our heap implementation will be block based and aligned to 4096 and will atleast be 4096 in size. So if you request to have ’50 bytes’, 4096 bytes will be returned.
  1. Entry table: - composes of an array of 1byte values that represents an entry in our heap data pool.
     1. We want a 100MB heap then, 100MB/4096 = 25600 bytes in our entry table.
     2. Entry 1 – 0x01001000, entry 2 – 0x01002000, and so on.
  2. Entry structure: - this will be 8 bit entry in or table
     1. Upper 4 bits are flags and lower 4 bits are the entry type.
     2. 7th – HAS\_N – set if the entry to the right od us is part of our allocation.
     3. 6th – IS\_FIRST – set if this is the first entry of our allocation
     4. 5th – 0 ‘zero’
     5. 4th - 0 ‘zero’
     6. 3rd – ET\_3
     7. 2nd – ET\_2
     8. 1st – ET\_1
     9. 0th – ET\_0
  3. Entry types
     1. HEAP\_BLOCK\_TABLE\_ENTRY\_TAKEN – the entry is taken and the address cannot be used
     2. HEAP\_BLOCK\_TABLE\_ENTRY\_FREE – the entry is free and may be used.

Malloc implementation example: -

* Take the size from malloc and cal how many blocks we need to allocate for this size. If user asks for ‘5000’ bytes then we will aloocate 4096\*2=8192 bytes.
* Check the entry table for the first entry we can find that has a type of HEAP\_BLOCK\_TABLE\_ENTRY\_FREE.
* Since we require 2 blocks we also need toensure the next entry is also free for use other wise we will need to discard the first block we found and look further in our table.
* Now return the absolute address that the starting block represents. Calculation for the absolute address is – (heap\_data\_pool\_start\_address + (block\_number\*block\_size))
* The data in the block will look like this –
  1. 0xC1 – block taken, first block, has more blocks for this allocation
  2. 0x41 – block taken, first block, no more blocks for this allocation
  3. 0x81 – block taken we have more blocks for this allocation, we are not he first block
  4. 0x01 – block taken, we are not the first block, no more blocks for this allocation
  5. 0x00 – block free.

Freeing the memory

* Go through the entry table starting at the block number we have calculated and set each entry to ‘0x00’ until we reach the last block of the allocation.
* We know how many blocks we need to free because the current block we are freeing will not have the “HAS\_N” bit set in the entry byte.
  1. Advantages
     1. Fast to allocate and free.
     2. Can be written in under 200 lines of code
  2. Disadvantages
     1. We assign block of mem so memory wasted.
     2. Memory fragmentation is possible.

Paging: -

* It allows us to remap mem addresses to point to other mem addresses.
* Can be used to provide the illusion we have max amount of RAM installed.
* Can be used to hide mem from other processes.
* If need to find anything related to OS development, go to osdev.org/paging.
* Remapping mem: -
  1. It allows to remap one mem addres to another.
  2. It works in 4096 byte block sizes by default. They are called pages.
  3. When it is enabled the MMU will look at your aloocated page tables to resolve virtual addresses into physical addresses.
  4. It allows us to pretend mem exists when it does not.
* Virtual add vs Physical add: -
  1. VA are addresses that are not pointing to address in mem that their value says they are. VA 0x10000000 might point to physical address 0x20000000 as an example.
  2. PA are absolute addresses in mem whose value points to same address in mem. For eg: - if PA 0x100 points to add 0x100.
  3. Essentially VA and PA are just terms we used to explain how a piece of mem is being accessed.
* Structure Overview: -
  1. 1024 page dir that point to 1024 page tables.
  2. 1024 page table entries per page table.
  3. Each page table entry covers 4096 bytes of mem.
  4. Each “4096” byte block of mem is called a page.
  5. 1024\*1024\*4096 = 4GB of addressable mem.
* Page Dir Structure
  1. Holds a pointer to a page table
  2. Holds attributes.
* Page Fault Exeption
  1. The CPU will call the page falult interrupt 0x14 when their was a pronlem with paging.
  2. The exception is invoked:-
     1. If you access a page in mem that does not have its “P (Present)” bit set.
     2. Invoked if you access a page that is for supt=ervisor but you are not supervisor.
     3. Invoked if you write to a page thai is read only and you are not supervisor.
* Hiding mem from processes
  1. If we give each process its own page dir table then we can map the mem for the process however we want it to be. We can make it so the process can only see itself.
  2. Hiding mem can be achieved by switching the page dirs. When moving between processes.
  3. All processes can access the same virtual mem addresses but they will point to different physical add.
* Illusion of more mem
  1. We can pretend we have max amount of mem even if we don not have.
  2. This is achieved by creating page tables that are not present. Once a process accesses this non-present address a page fault will occur. We can then load the page back into mem and the process had no idea.
  3. 100MB sys can act as if it has access to full 4GB on 32 bit arch.
* Benefits to paging
  1. Each process can access that same virtual mem add, never wrtining over each other.
  2. Security is an added benefit as we can map out physical mem that we don’t want processes to see.
  3. Can be used to prevent overwriting of sens data such as program code.

A computer screen shot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screen shot of a computer

Description automatically generated

PCI IDE controller: -

* IDE refers to electrical spec of cables which connect ATA drivers to another device. It allows upto 4 drives to be connected
  1. ATA (serial): - used for modern hard drives.
  2. ATA (parallel): - used for hard drives.
  3. ATAPI (serial): - used for modern optical drives.
  4. ATAPI (parallel): - commonly used for optical drives.
* Kernel programmers do not have to care if the drive is serial or parallel.
* Possible Drive types: -
  1. Primary master drive
  2. Primary slave drive
  3. Secondary master drive
  4. Secondary slave drive
* ATA Read Example

Int disk\_read\_sector(int lba, int total, void\* buf){

Outb(0x1f6, (lba >> 24) | 0xE0); // select master drive and pass part of the lba

Outb(ox1f2, total); // send the total number of sectors we want to read

Outb(0x1F3, (unsigned char)(lba & 0xff)); // send more of the lba

Outb(0x1F4, (unsigned char)( lba >> 8)); // send more of the lba

Outb(0x1F5, (unsigned char)(lba >> 16)); // send more of the lba

Outb(0x1F7, 0x20); // 0x20 = read command

}

Unsigned short\* ptr = (unsigned short\*) buf;

For(int b=0; b< total; b++){

//wait until buffer is ready

// poll until we are ready to read.

// You can also use interrupts if you prefer.

Char c = insb(0x1F7);

While(!(c & 0x08)){ c = insb(0x1F7) ;}

//copy from hard disk to memory 2 bytes at a time

// read 3 butes at a time into buffer from the ATA controller.

For(int i=0; i<256; i++){

\*ptr = insw(0x1F0);

Ptr++;

}

Return 0;

}

* Filesystem: -
* A fs is a structure that describes how information is laid on a disk.
* Disks are not aware of files. The OS knows the fs struct so knows how to read files form the disk.
* Without implementing a fs in your OS you cannot have files.
* Disks: -
  1. Hard disks can be thought of as just a gaint array of info split into sectors.
  2. Each sector can be read into memory and is given a LBA (Logical Block Address) number.
  3. Files do not exist on the disk.
  4. Disk have no concept of files.
* FS structure:-
  1. Contains raw data for files.
  2. Contains fs struct header which can explain things such as how many files are on the disk, where the root directory is located and so on.
  3. The way files are laid out on disk is different depending on the fs you are using for example, “FAT16”, “FAT32”, and more.
  4. Without fs we would be forced to read and write data through the use of sector numbers, struct wourld not exist and corruption would be likely.
* FAT16 (File Allocation Table) 16 bits
  1. The first sector in this fs format is the boot sector on a disk. Fields also exist in this first sector that describe the fs such as how many reserved sectors follow it.
  2. Then follows the reserved sectors these are sectors ignored by the fs. There is a filed in the boot sector tha specifies how many reserved sectors there are. Remember the OS must ignore these its not automatic! The disk has no idea.
  3. Now we have our first file allocation table, this table contains values that represent which sluster on the disk are taken and which are free. A cluster isjust a certain number of sectors joined together to represent on cluster.
  4. Next comes our second file aloocation table it’s optional though and depends on the FAT16 header in the boot sector.
  5. Now comes our root directory this explains what files/ directories are in the root directory of the fs. Each entry has a relative names that represents the file or directory name, attributes susch as read only, the add of the first cluster representing the data on the disk, and many more.
  6. Finally, we have our data region, all the data is here.
* A screen shot of a computer

  Description automatically generated

FAT16 fs: -

* Uses cluster to represent data ndand subdirectories.
* Each cluster uses a fixed amount of secotes which is specidied in the boot sector.
* Every file in FAT16 needs to use atleast one cluster for its data this means a lot of storage is waterd for small diles.
* FAT16 cannot store files larfer than 2GB without large file support. With large file support 4GB is the max.
* FAT 16 explained: -
  1. Each entry in the eable is 2 bytes long and represents a cluster in the data clusters region that is available or taken.
  2. Clusters can chain tofether, for eg a file larfer thn one cluster will use 2 clusters. The value that represents the first cluster in the file allocation table will contain the calue of the next cluster. The final cluster will contain a value of 0xffff signifying that there are no more clusters.
  3. The size of a cluster is represented in the boot sector. The position specified as the data cluster is this cluster
* A computer screen with a computer screen

  Description automatically generatedA screenshot of a computer

  Description automatically generated
* FAT16 root dir: -

A computer screen shot of a computer program

Description automatically generated

* 1. FS have directories/folders. This is no different. It has whats known as root directory, this is the top most directory in the system.
  2. Directories contain directory entries of a fixed size.
  3. Attributes filed contains flags that determine if this dir item is a dile or a directory. If its read only and so on.
  4. In the boot sector contains the max number of root dir entries we should not exceed this value when iterating rhrough the root dir.
  5. We know when we have finished iterating through rhe root dir or a subdir because the first byte of the filename will be equal to zero.
  6. Dir entry attribute flags: -
     1. 0x01 – read only
     2. 0x02 – file hidden
     3. 0x04 – system file do not move the cluster
     4. 0x08 – volume label
     5. 0x10 – this is not a regular file, it’s a subdir
     6. 0x20 – archived
     7. 0x40 – device
     8. 0x80 – reserved must not be changed by disk tools.
  7. Filename and extension
     1. The filename is 8 bytes wide and unused bytes are padded with spaces (0x20).
     2. The extension is 3 bytes wide and unused bytes are padded with spaces (0x20).
* Clusters: -
  1. Each cluster represents a certain amount of sectors linearly to each other.
  2. The amount of sectors that represents a cluster is stored in the boot sector.
  3. The data cluster section in the fs contains all the cluster that make up the subdir and files data of files throughout the FAT fs.
* Things to do to make syre that the linux or kernel understand the fs: -
  1. In the linux terminal, write cmd – “sudo mkdir /mnt/d”.
  2. Cmd - “sudo mount -t vfat ./bin/os.bin /mnt/d”.
  3. Now move to that location using cmd – “cd /mnt/d”.
  4. Create a text file cmd – “sudo touch ./hello.txt”.
  5. Open the text file – “sudo gedit hello.txt”.
  6. In that file write – “hello world” and save that.
  7. Now unmount using cmd - “sudo unmount /mnt/d”.
  8. To check it, go to cmd - “cd /home/PeachOS/bin” and use this cmd – “bless ./bin/os.bin”. You will see “hello world” written there.
  9. Now remount the harddisk or OS again using cmd – “sudo mount -t vfat ./bin/os.bin /mnt/d”.
  10. Add the above cmd in the Makefile and also create a hello.txt file. By running this we can see that kernel is now able to read the file.
* Understanding the VFS layer: -
  1. The VFS layer allows a kernel to support an infinite amount of fs.
  2. The VFS layer allows to abstract out low-level fs code.
  3. Allows fs functionality to be loaded or unloaded to the kernel at any time. Fs drivers can be loaded or unloaded on demand.
  4. The programming interface to the fs remains the same for all fs.
* What happens when a disk gets inserted?
  1. We poll each fs and ask if the disk holds a fs it can manage.
  2. We call this resolving the fs.
  3. When a fs that can be used with the disk is found then the disk binds itself to its implementation.
* Communication:-
  1. User program <-> Kernel <-> FAT16, NTFS, FAT32,..
* FOPEN communication: -
  1. Suppose in user prog we use fopen(“0:/test.txt”, “r”); this things happens
     1. Kernel uses “path parser” to extart the location of the file. Thencommunications happens in this way
     2. User <-> kernel <-> path parser/root <-> disk(0) <-> FAT16 <-> fopen.
     3. Here the user calls kernel, and from their using path parser it finds on which disk it is stored, all the disk has a fs head which helps to understand how to read the file and follow the ops like fopen.
     4. We use file descriptor to store this information from path parser to disk(0).
* FREAD comms: -
  1. Kernel <-> fd <-> FAT16 <-> fread <-> buf
  2. As kernel is not directly accessing the fs, kernel does not need to know everytime the fs which make is more efficient and it can easily read the files.
  3. The caller of the file routine does not have to care about which fs to use.

User land/space: -

* User land is a term used to describe when the processor is in a limited privileged state.
* It is what OS processes run in.
* User land is safe because if something goes wrong the kernel is able to intervene.
* It is when the processor is in ring 3.
* Restrictions
  1. Access to certain locations in memory can be restricted for user land processes.
  2. Access to certain CPU instructions are restricted from user land.
  3. Using paging the kernel can ensure all processes cannot access each others memory. User land code is unable to override this because its running in an unpriviled state. The instructions for switching pages are disables.
  4. Attempting to run privileged instructions whilst in user land will cause a protections fault. The protection fault exception interrupt handler will then be responsible for solving the problem.
* Kernel land is hen the processor is in its max privileged state.
* Whilst in kernel land any area in memory can be changed, any CPU instruction can be run.
* There is also a high risk of damage to the system if things go wrong.
* Kernel land is when the processor is in a privileged protection ring such as ring 0.

User land setup: -

* Setup user code and data segments.
* Setup a TSS(Task Switch Segment).
* Pretend we are returning from an interrupt pushing appropriate flags, and data to the stack before executing an “iret” instruction to change the processors privilege state.
* Code: -
  1. Struct gdt gdt\_eral[COS32\_TOTAL\_GDT\_SEGMENTS];
  2. Struct gdt\_structred gdt\_structured[COS32\_TOTAL\_GDT\_SEGMENTS] = {

(.base = 0x00, .limit = 0x00, .type = 0x9a), //null segment

(.base = 0x00, .limit = 0xfffffffff, .type = 0x9a), //kernel code segment

(.base = 0x00, .limit = 0xfffffffff, .type = 0x92), //kernel data segment

(.base = 0x00, .limit = 0xfffffffff, .type = 0xf8), // user code segment

(.base = 0x00, .limit = 0xfffffffff, .type = 0xf2), // user data segment

(.base = (uint32\_t)&tss, .limit = sizeof(tss), .type = 0xE9) // TSS segment

};

* TSS (task switch segment): -
  1. It is a way for the processor to get back to kernel land when we have a system interrupt.
  2. The TSS explains things such as where the kernel stack is located.
  3. Upon receiving an interrupt when the processor is in user land state, the processor will switch to kernel code and data segments. It will then restore the stack pointer located in the TSS before then invoking the kernel interrupt handler.
  4. In TSS struct we only have to care about the ESP0 and SS0 variables for our TSS.
* Example of returning from an interrupt
  1. We should set our segment register to the user data segment that we created in previous steps. This is likely 0x23. “ds”, “es”, “fs” and “gs” regs should be changed, but not stack segment.
  2. Next we save our stack pointer in the EAX red as we are about to modify the stack.
  3. Now we push our user data segment to the stack “0x23”.
  4. Now we push our stack pointer we saved in EAX earlier.
  5. Next we push our current flags to the stack but not before we bitwise OR the bit that re-enables interrupts. This is important as our interrupts are cleared at this moment in time and we only want to re-enable them when we “iret”.
  6. Now we push the user code segment which should ne “0x18”.
  7. Finally we psuh the address of the function we want to run in user land.
  8. The last step is to call an “iret” which should force the process into a user land unprivileged state.
* Getting back to user land when in a kernel interrupt: -
  1. When an interrupt is invoked whilst the processor is in the user land state the processor will push the same regs that we pushed to get to the user land in the first plave. This way getting back to user land is very easy you just invoke “iret” at the end of your kernel interrupt routine, causing the kernel to go back to the user program and just after the user programs interrupt instruction.
  2. In a multi-tasking system user land regs will need to be salvaged when entering kernel land, this is imp so we can switch to the next process task if we want too. When ever we want to switch back to the old task we just swap the old regs of the task back to the real CPU regs again and then finally we drop the processor back into user land the task will then continue executing as if nothing happened.

Talking with the kernel from userland: -

* Communication with the kernel form a process
  1. User program calls an interrupt using the interrupt instructions.
  2. Kernel interrupt routine is executed and extracts arguments prushed by the user program.
  3. Kernel interrupt routine returns the result.
* User program calls an interrupt using the interrupt instruction: -
  1. Lets we have a kernel operation that’s represented by code 1. It simply prints a messafe to the screen.
  2. Telling the kernel to print is easy from userland we do the following: -
     1. The user program begins by setting the “EAX” regs to 1, this is the kernel operation code for print operations.
     2. The user program then pushes the address of the message that should be printed to the screen.
     3. The user program issues an interrupt to the kernel. The interrupt number used is 0x80 as in this hypothetical kernel implementation we have decided to use interrupt 0x80 for handing cmds to the kernel.

Print:

Push ebp

Mov ebp, esp

Mov eax, 1 ; cmd 1 =print

Mov ebx, [ebp+8] ; string to print

Push dword ebx ; push it to the stack

Int 0x80 ; invoke kernel to print

Add esp, 4 ; restore stack

Pop ebp

ret

* Calling the kernel overview: -
  1. The processor pushes the same info we pushed to get into user land in the first place to the stack.
  2. Interrupt 0x80 kernel routine begins execution, the cmd number is extracted from the EAX reds.
  3. The C interrupt handler for 0x80 is called.
  4. The task that executed the interrupt has its sate saved, all regs for that task are saved in the tasks regs structure. These are extracted from the interrupt fram.
  5. Execution flow is passed to the correct handler for the cmd number that was provided to the kernel in the EAX regs.
  6. Kernel does the action that it was instructed to do from the user land. i.e print to the screen.
  7. Kernel cmd handler returns a value.
  8. Execution continues after the user lands “int 0x80” instruction.
  9. The EAX reg is populated with the return result from the kernel.

Understanding keyboard access in protected mode: -

* Keyboard access is interrupt driven.
  1. In protected mode we make use of the keyboard with interrupts.
  2. Each time a key is pressed a CPU interrupt is called that causes our interrupt handler to run.
  3. We mapped our PIC to interrupt 0x20, so the interrupt the keyboard uses is interrupt 0x21.
  4. We are responsible for reading the scan code and issuing an acknowledgement to the PIC.
* Each process has its own keyboard buffer.
  1. This keyboard buffer will be inside the process structure
  2. Tail will point to an index in the keyboard buffer array that represents the tail and head will point to head buffer.
* Keyboard buffer can be pushed and popped from.
* We must parse scan codes.
  1. The PIC doesn’t tell us which key was pressed. It gives us a scan code which we must parse into an ASCII value.
  2. An array of characters can be used to assist with this.
  3. If we care about lowercase and uppercase letters we must also keep track of the caps lock key and weather it’s a down state.
* Understanding the keyboard buffer
  1. Imagine a keyboard buffer that’s 1024 bytes in size.
  2. We have a head variable equal to zero signifying that the head is at index 0.
  3. We have a tail variable equal to zero signifying that the tail is at index 0.
  4. Before we do anything the keyboard buffer’s tail and head variables both point to index zero of the keyboard buffer.
  5. So lets assume keyboard buffer 1024, Initially both head and tail will be pointing to 0th index,  then a keyboard input is given and will occupy 0th index, and now the tail will move to 1st index and head will remain at 0th index. Suppose another input buffer provided, so tail at 2nd and head at 0th. Now when read from keyboard, tail will be on 2nd index, but head will move to 1st index, and so on. And after 1023, the tail will move to 0th index and this goes on.
* We will have a virtual keyboard layer
  1. Just like we did with filesystems we will have a virtual keyboard layer.
  2. This will allow us to create drivers for any type of keyboard that we wish.
  3. Keyboard will be able to inserted into the system by calling the “keyboard\_insert” function.