# DaxOS

PROJECT REPORT

submitted by

*By*

Nihal Narayan (MBT17CS081)

Antony S. Chirayil (MBT17CS023)

Mathew Koshy (MBT17CS068)

R Midhun Suresh (MBT17CS095)

to

**the APJ Abdul Kalam Technological University**

**in partial fulfillment of the requirements for the award of the Degree**

of

**BACHELOR OF TECHNOLOGY**

IN

*COMPUTER SCIENCE AND ENGINEERING*

**June, 2021**


**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEEERING**

MAR BASELIOS COLLEGE OF ENGINEERING & TECHNOLOGY

Mar Ivanios Vidya Nagar, Nalanchira

Thiruvananthapuram 15

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

MAR BASELIOS COLLEGE OF ENGINEERING & TECHNOLOGY

Nalanchira, Thiruvananthapuram.


## CERTIFICATE

*This is to certify that the report entitled* ***DaxOS*** *submitted by* ***Nihal Narayan (MBT17CS081)***, ***Antony S. Chirayil (MBT17CS023)***, ***Mathew Koshy (MBT17CS068)***, ***R Midhun Suresh (MBT17CS095)*** *to the APJ Abdul Kalam Technological University in partial fulfillment of the requirements for the award of the Degree of Bachelor of Technology in Computer Science and Engineering and Technology is a bonafide record of the project work carried out by him/her under my/our guidance and supervision.This report in any form has not been submitted to any other University or Institute for any purpose*

| **Ms. Gayathri K. S.** | **Mr. V. S. Shibu** | **Dr. Tessy Mathew** |
|---|---|---|
| Project Coordinator | Guide | Head of the Department |

***Place: Thiruvananthapuram***

***Date: 12/01/2021***

# Acknowledgement

First and foremost we would like to thank our Principal, **Dr. Abraham T Mathew** and our HOD, **Dr. Tessy Mathew** for providing us with this opportunity.

We would like to take this opportunity to extend our sincere gratitude to **Mr. Shibu VS** for guiding us through the process of this project. We would also like to thank the seminar coordinator, **Mrs. Gayathri KS**, for her help in bringing this project to fruition.

**Nihal Narayan**

**Mathew Koshy**

**Antony S. Chirayil**

**R Midhun Suresh**

# ABSTRACT

Every computer science enthusiast is proficient in their operating system of choice. They may even know its underlying working from an old operating system course they took in college. However, it is usually the case that their knowledge and understanding is limited to theory and writing low-level system code is often considered an insurmountable challenge. This project hopes to change this attitude by developing a minimal yet functional 32-bit operating system that can be used in conjunction with theoretical teaching to promote and introduce systems programming. A minimal kernel guarantees easier to read source code (as opposed to the 27 million SLOC Linux kernel) and provides a gentler introduction to kernel development.

# Contents

# List of Figures

# Nomenclature

| | |
|---|---|
| $GDT$ | Global Descriptor Table |
| $IDT$ | Interrupt Descriptor Table |
| $ISR$ | Interrupt Service Routine |
| $VGA$ | Video Graphics Array |
| $PIC$ | Programmable Interrupt Controller |
| $VESA$ | Video Electronics Standards Association |

# Chapter 1

# Introduction

The Kernel is the fundamental interconnect between hardware and software of a computer system. Writing a kernel (kernel programming) is considered to be a difficult endeavor because development has to start from a bare metal state. DAX OS is a minimal 32-bit hobbyist operating system that can be used to provide a gentle introduction to students who wish to explore the domain of systems programming.

The project is open source and licensed under GNU General Public License v3.0 to ensure unrestricted access and complete transparency. DAX OS comes with a terminal driver, keyboard and mouse driver and basic memory management. The project also uses appropriate development practices such as unit-testing and version control.

Some key facts about this project are:

- Source code is hosted at https://github.com/DaxKernel/OS.
- About 2500 lines of code only.
- 32-bit
- Low resource consumption.

## 1.1 Objectives

We propose to build a 32-bit kernel that has the following functionality:

1. **Keyboard Driver**

   Dax-OS includes a fully functional PS/2 keyboard driver. The PS/2 keyboard driver will convert scan-codes generated when the user presses a key on the keyboard to an integer character code. It must be noted that all keyboards practically used in modern day utilize the USB standard. We stick with the PS/2 protocol because the USB protocol is massive and difficult to implement. However there are practically no disadvantages from such a decision because most motherboards will emulate USB keyboards as PS/2 keyboards.

2. **Terminal Display Driver**

   DaxOS is a terminal based operating system i.e it does not support windowing. The display support will be implemented using VGA text-mode and real-mode. It will later be re-implemented in Graphics.

3. **Interrupts**

   DaxOS makes heavy use of interrupts for device drivers. It also uses interrupts for handling CPU exceptions.

4. **Memory Management**

   DaxOS uses a flat memory model. Since it is an 32-bit operating system, it will support at most 4GB of addressable memory. A custom memory allocator has also be implemented.

5. **Graphics Support**

   DaxOS will provide graphics capability using VESA mode. The supported resolution is 1280x720.

6. **Font Rendering**

   With graphics support, the vga text-mode based terminal driver is depreciated in favor of the higher resolution VESA mode. Font rendering needed for this scenario is also implemented.

7. **Image Rendering**

   DaxOS will natively support rendering images in TGA format.

## 1.2 Limitations

Some functionality DaxOS does not implement are:

- Process Management
- GUI
- Paging

## 1.3 Technology Stack

The bulk of the operating system is written in the C programming language. Certain functionality such as writing data to ports and loading tables (IDT, GDT) are implemented using either GCC inline assembly or using normal x86 assembly.

The project uses the GCC cross-compiler & binutils to target the generic i686 platform; which is a generic 32-bit Intel P6 architecture. The GNU Assembler and Linker are also used. The assembly syntax style used is AT&T.

The entire compilation process is driven using GNU Make which uses Makefiles to build the project. The compilation is initiated using BASH shell scripts. The kernel is tested using the qemu-i386 emulator and is developed on a stable Xubuntu distribution.

DAX OS uses git as its version control system of choice. The git repository is uploaded on Github and every new feature is developed on its own separate branch. Team communication and coordination are actualized using discord with Github integration enabled.

# Chapter 2

# Design Diagrams

## 2.1 Use-case diagram

The use case diagram illustrates how the user interacts with the operating system. There is a total of three interactions between user and the operating system in this case. The first interaction that the user initiates is the boot process.
This involves:

1. **Setting up interrupts**

   The concept behind interrupts is that when a piece of hardware or software wants to interrupt the CPU to do something important, an interrupt is raised. The CPU executes a program called the ISR (Interrupt Service Routine) and returns back to what is was doing before.

2. **Loading VGA driver**

   Using VGA to print text to the screen is quite simple and is achieved by using VGA text-mode. The user can directly write to the video memory located at address 0xB8000. Each character that is to be printed requires a two-byte representation:
   One byte, called the code-point is used to represent the character in ASCII. The other is used to set the background and foreground colors. There are 16 colors that can be used. In VGA text-mode a maximum of 80x25 characters can be printed on the screen at a time.

3. **Load keyboard driver**

   The keyboard driver uses the PS/2 interface to enable communication between the keyboard and the computer. When a key is pressed on the keyboard, the PIC raises IRQ1. This triggers an ISR which on execution will store the pressed key on a circular buffer. The data from the circular buffer is read by primitives from the C standard library such as scanf in stdio. The data from keyboard is read from port 0x60 using inline assembly function **inportb()** defined in io.c file.

4. **Setting up memory**

   This involves ensuring that multiboot is working as intended and confirming that the **multiboot_info_t** structure contains the fields neccessary for memory management. Other intermediate datastructures needed for memory management such as the stack and global variables are also initialized at this stage.

The second interaction involves the user issuing a command to the terminal.
The command may require dynamic memory to be allocated. Dynamic memory is allocated by implementing the malloc C function. The malloc function acts as a wrapper around the kernels memory manager.

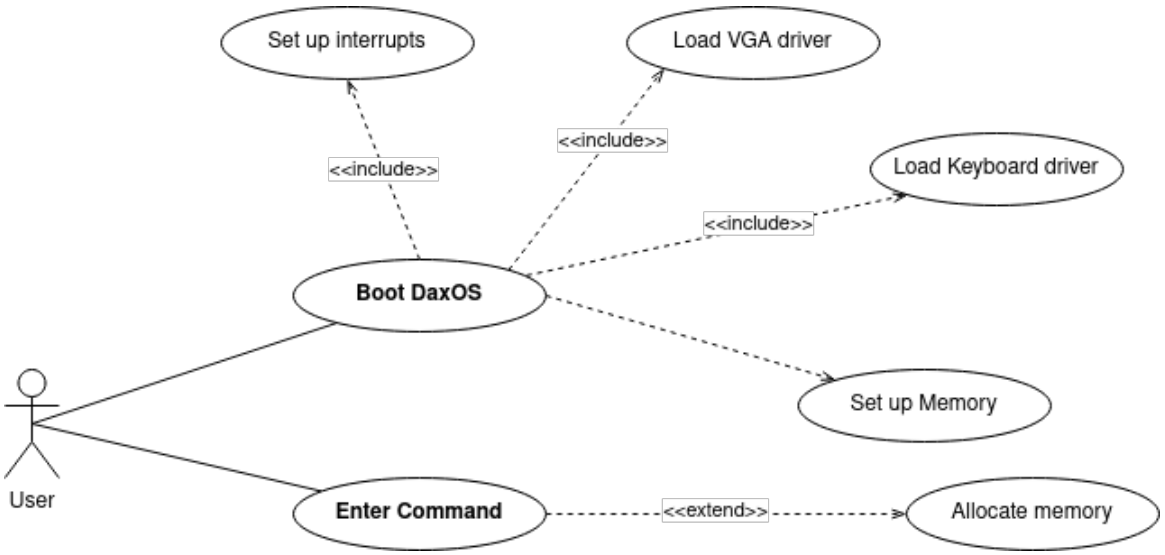


Figure 2.1: Use Case Diagram

## 2.2 Activity Diagram

The activity diagram illustrates the shell scripts that are used to build and compile the kernel.
The most important script here is the **build.sh** shell script which dives the make program to compile our source code. The build script relies further on two other shell scripts:

1. **headers.sh** script

   Since in this project a version of C standard library is implemented the compilation of kernel is done by instructing the GCC cross-compiler to look for system headers in the SYSROOT directory. This script compiles the custom C std-lib into an archive named libk.a and places it in the SYSROOT/usr/lib directory. Also it copies all the .h header files into SYSROOT/usr/include directory. Now compiling the kernel is done by passing the –sysroot=SYSROOT parameter.

2. **config.sh** script

   This script sets up all the environment variables used by GNU Make.This allows the user to easily change the core aspects and tooling used in the projects later. For example, we could change the compiler from GCC to

CLANG by simply changing the environment variable CC to CLANG. Likewise SYSROOT directory can be changed by changing a single variable. Once these two scripts are executed the build script runs the make-install command on each of the folder directory. It copies the output DaxOS.kernel file into the boot directory.

The **iso.sh** script builds a bootable .iso image file from our kernel. It first calls the previously described build script and then produced an iso file by using the grub-mkrescue program. This iso file can be loaded into any emulator to run the operating system. The iso files are built into the isodir directory.

The **clean.sh** script removes all the build artifacts from the compilation. Build artifacts are files that are produced by the compilation process that can always be reproduced by the compiler. Since it can be reproduced it is usually removed to maintain a clean project structure.
There are three kinds of build artifacts:

1. **Object files** - .o files
2. **Make dependencies** - .d files
3. **Unwanted directories** - sysroot, isodir

The clean script works by executing the make clean commands in each of the project directories.The make clean commands use the rm bash command to delete the build artifacts.

The last script is the **create-bootable-usb.sh** script. This script creates a bootable USB of the operating system. It needs sudo permission because it accesses the UNIX block file of the USB device. This of the form /dev/sda or something similar. This script formats the USB device and copies the kernel along with the GRUB bootloader into the USB device. Therefore there is a chance that the data in the USB device can be destroyed if the user is not careful. To prevent this a safety check that checks if the device has more than 10 GB capacity is done. If this is true

we do not format the device and exits with error. If the capacity is less than 10GB, the USB is made bootable.

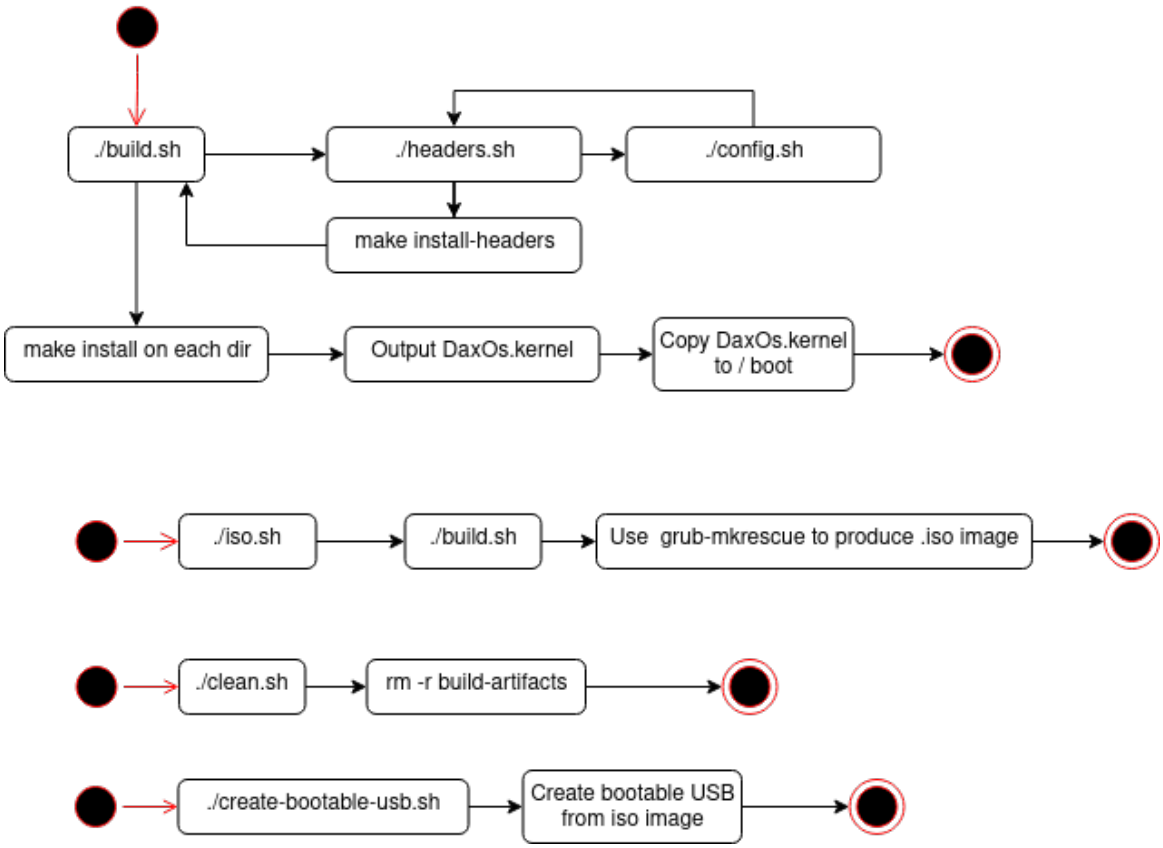


Figure 2.2: Activity Diagram

## 2.3 Sequence Diagram

This diagram illustrates the time dependent and sequential interaction between the user, the OS and the keyboard driver. During boot, the Operating System initializes the keyboard driver. This involves:

1. **Self Test**

   (a) The keyboard device is disabled by sending command 0xAD and 0xA7 to the PS/2 controller.

   (b) The PS/2 controller's output buffer is flushed by reading from port 0x60.

   (c) Initiate the PS/2 controller self test by sending command 0xAA to it. A response of 0x55 indicates success.

   (d) Enable the keyboard device by sending commands 0xAE and 0xA8 to the PS/2 controller.

   (e) Reset the device by sending 0xFF to the keyboard.

   (f) Set the LED states by sending appropriate commands.

2. **Handling Keypress**

   When the user presses a key the keyboard device initiates an interrupt. This done by activating IRQ1 which is the standard interrupt line used by keyboards. In response to this interrupt, the CPU executes an ISR which updates a buffer with the read characters. The I/O functions in stdio.h like scanf will read from this buffer to perform the necessary computation and show the results back to the user.

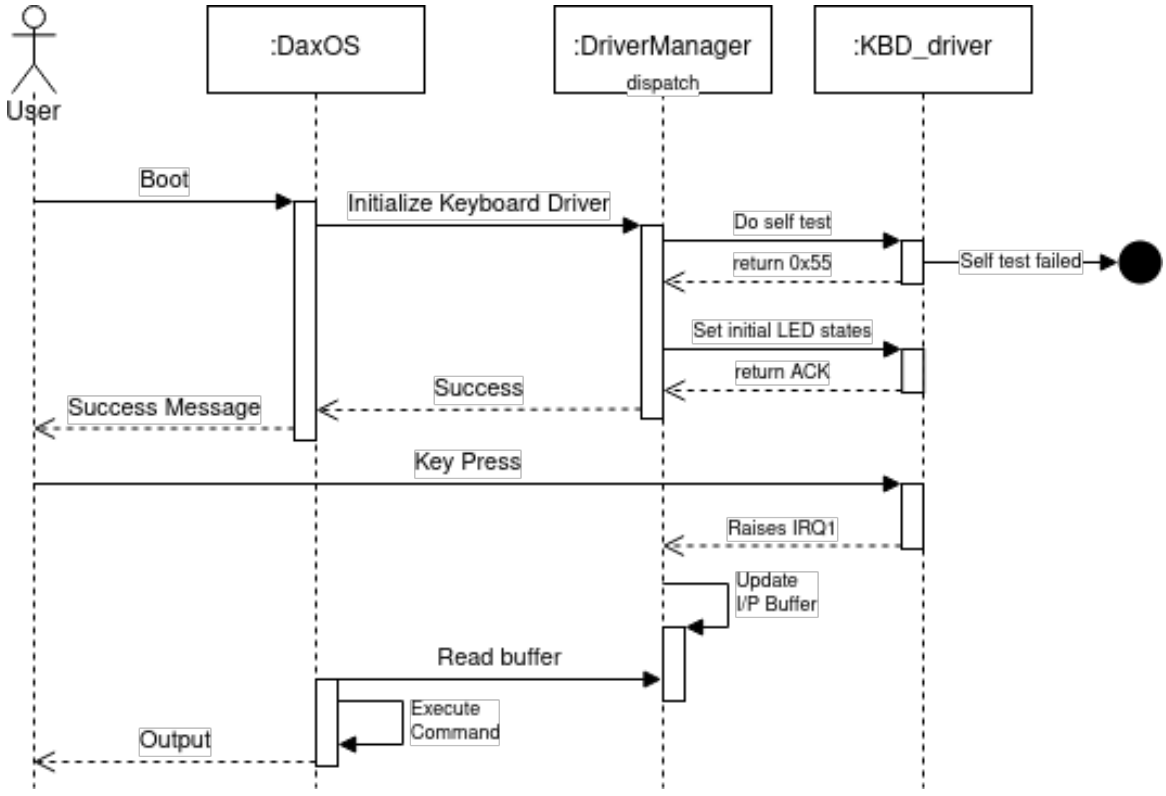


Figure 2.3: Sequence Diagram for keyboard driver

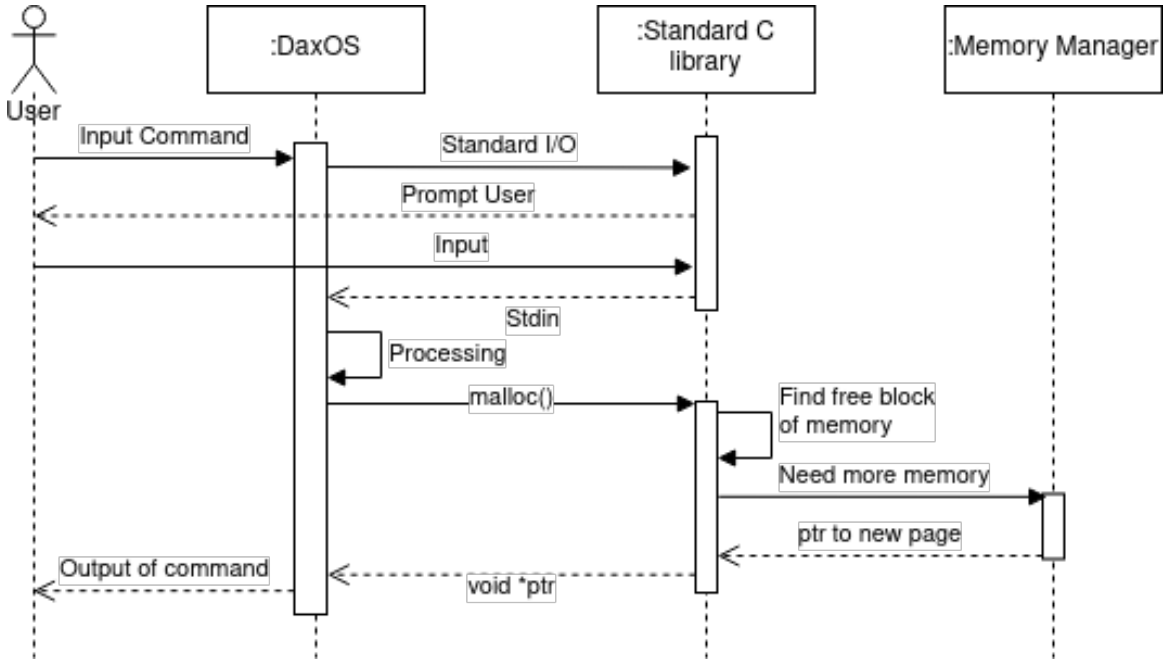


Figure 2.4: Sequence Diagram for user command

# Chapter 3

# Build Process

## 3.1 Overview

DaxOS requires a certain number of tools for the build process:

- **Cross-compiler**:
  This is a special kind of compiler that can output binaries for a different kind of computer architecture than the one in which it resides.

- **Binutils**:
  This contains other tools such as the assembler and linker.
  It also contains an archiver used to produce libraries.

- **Emulator**:
  Since an OS cannot be run as easily as other types of software, an emulator is used to accelerate the code-build-debug cycle.

- **Linux**:
  Although this is not a strict requirement, most of the tools specified above are easily available on Linux.
  Other OS such as Window can still be used although it is more time-consuming and non-intuitive.

## 3.2 Building the cross-compiler and binutils

These tools cannot be downloaded as a binary package. Instead they must be compiled from source. The build process for these tools are somewhat complicated. However in summary, the steps to be followed are:

1. Download respective source files from the GNU website.
2. Install the necessary dependencies using the package manager. For example on debian based distributions, the *apt* package manager is used as follows:

```
sudo apt install build-essentials, bison, flex, libgmp3-dev ...
```

3. Add path variables:

```
export PREFIX="$HOME/opt/cross"
export TARGET=i686-elf
export PATH="$PREFIX/bin:$PATH"
```

4. Run configure scripts.
5. Issue make-install commands as follows:

```
make all-gcc
make all-target-libgcc
make install-gcc
make install-target-libgcc
```

6. Add newly compiled compiler binaries to $PATH variable.

## 3.3 Setting up the emulator

DaxOS uses *qemu* as its default and preferred emulator.
Qemu is fast, open-source and can be easily run from the terminal. It can be easily installed using:

```
sudo apt install qemu-system-i386
```

## 3.4 Overview of shell-scripts

Building DaxOS is a complicated process that involves running many different commands with a large number of optional parameters and flags. Therefore to make the build process easier, DaxOS comes with a set of shellscript wrappers to automate the heavy build process. With the shellscript, compiling the OS is as easy as invoking:

```
.\build.sh
```

Other available shellscripts and their functionalities were introduced in the Design Diagrams chapter.

## 3.5 Makefile

A Makefile is a particular type of file that specifies the steps to be taken to build a particular piece of software.
DaxOS uses Makefiles to drive the compilation process.
Makefiles are written by specifying a target and the steps needed to produce that target. Additionally dependencies needed to build a target can also be specified, in which case, the steps needed to produce the dependencies also need to be written.

Consider the following snippet of a Makefile that is used in DaxOS:

```
DaxOS.kernel: $(OBJS) $(ARCHDIR)/linker.ld
$(CC) -T $(ARCHDIR)/linker.ld -o $@ $(CFLAGS) $(LINK_LIST)
grub-file --is-x86-multiboot DaxOS.kernel

.c.o:
    $(CC) -MD -c $< -o $@ -std=gnu11 $(CFLAGS) $(CPPFLAGS)

.S.o:
    $(CC) -MD -c $< -o $@ $(CFLAGS) $(CPPFLAGS)
```

This makefile can be interpreted as follows:

- To produce **DaxOS.kernel**, the files in *OBJS* and *linker.ld* in *ARCHDIR* needs to be produced first.
- Object files(*.o*) are produced from C source files (*.c*) files by executing $(CC) command.
- Object files(*.o*) are produced from Assembly source files (*.S*) files by executing $(CC) command.
- $(foo) indicates a variable named *foo*.

**Note**: $(CC) here is a variable that is set to the GCC cross-compiler by the shellscript.

# Chapter 4

# VGA Display Driver

## 4.1 Overview

DaxOS initially used VGA text mode for its display. VGA text-mode is a part of the VGA standard and is available on most hardware. However it is depreciated on newer systems in favor of the higher resolution VESA or GOP buffers on newer UEFI based systems. The primary advantages of using VGA text-mode are:

1. **No need to worry about font-rendering**
   VGA text-mode abstracts away most of the complexity involved in writing a display driver including font management and rendering.

2. **Easy to program**
   Writing code to use VGA text-mode is as easy as writing characters to a particular memory address.

3. **No additional dependencies**
   You do not need an existing malloc() for instance.

However there are also a couple of disadvantages:

1. **Low resolution**
   Text resolution is limited to about 25 lines of 80 character each.

2. **No drawing capability**

   It is not possible to draw pixels onto the screen in this mode.

3. **16 color palette**

   Only 16 colors are available in this mode.

## 4.2 Printing text to the screen

It is very simple to print text to the screen using VGA-text mode. The process basically involves writing the necessary character onto the VGA text-buffer. This buffer is located at **0xB8000**.

Every character to be displayed is represented by two bytes:

- First byte determines the background and foreground color of the character.
- Second byte is the codepoint; i.e the character itself.

Therefore to print the character 'A' to the screen:

```
char *buffer = (char *)(intptr_t)0xB8000;
*buffer++ = 12; \\bg is black, fg is white
*buffer = 'A';
```

The actual display driver is much more sophisticated and involves a standardized API that the other parts of the OS can use.

| Attribute | | | | | | | | Character | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Blink | Background color | | | Foreground color | | | | Code point | | | | | | | |

Figure 4.1: VGA representation for a single character

## 4.3 Display Driver API

The display driver supports the following API calls:

```
void tty_initialize(void);

void tty_put_char(char c);

void tty_write(const char *data, size_t size);

void tty_write_string(const char *data);

void tty_write_string_centered(const char *string);

void tty_print_horizontal_rule(const char symbol);

void tty_setcolor(uint8_t color);

void tty_print_success(const char *string, const char *success_string);
```

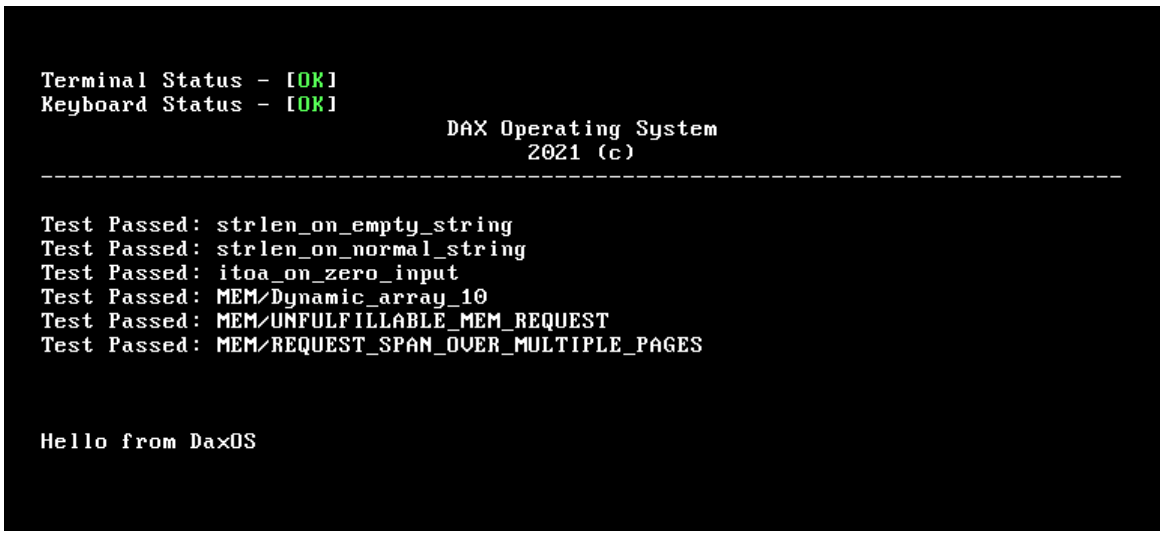


Figure 4.2: VGA display driver running in DaxOS

# Chapter 5

# Interrupts

Interrupts provide a quick and easy way for different parts of the OS to signal that important events that need attention have occurred. The basic idea here is that when something important that needs to be immediately handled occurs, the CPU stops what it is currently doing and calls another function. This function is called an Interrupt Service Routine.

There are a total of 256 interrupts in the x86 hardware architecture. Of this, the first 32 are CPU exceptions. The interrupts from 32-47 deal with various parts of the system such as keyboard and mouse. The remaining interrupts are user-defined.

When an interrupt occurs the CPU looks up the corresponding ISR from a table called Interrupt Descriptor Table. It then branches to that function. ISRs are different from other functions in that they use the **iret** instruction to return instead of the commonly used **ret** instruction. The difference is that while the latter only restores the IP, the former also restores the flags and CS registers.

## 5.1 Programming Interrupts

### 5.1.1 Creating the IDT

An overview of the implementation is as follows:

1. Each value in the IDT is represented by a C struct.
2. The IVT itself is simply an array of C structs.

Thus the C code for the IDT is as follows:

```
struct IDT_entry
{
    uint16_t offset_lowerbits;
    uint16_t selector;
    unsigned char zero;
    unsigned char type_attr;
    uint16_t offset_higherbits;
} IDT[IDT_SIZE];
```

The struct fields are described as follows:

- **offset_higherbits, offset_lowerbits**:

  These together form the memory address of the ISR.

- **selector**:

  This is a code segment selector defined in the Global Descriptor Table (GDT).

- **zero**:

  This has to be zero by convention.

- **type_attr**:

  This decides the type of interrupt gate (interrupt gate or task gate).

### 5.1.2 Creating the ISR

Now that the IDT has been created interrupt handlers can be created.
There are two options to proceed:

- Use x86 assembly
- Use C with GCC extensions

The second choice is preferred because it gives the opportunity to write cleaner code.
However it must be noted that using C by itself is not an option because functions in C always end with the ret instruction. GCC compiler provides additions to the C programming language that can sometimes be helpful in scenarios such as this.

Some C code to make the implementation easier is written:

```
struct interrupt_frame
{
    uint32_t error_code;
    uint32_t ip;
    uint32_t cs;
    uint32_t flags;
};
#define IRQ_HANDLER __attribute__((interrupt)) void
#define IRQ_ARG __attribute__((unused)) struct interrupt_frame *frame
```

The interrupt attribute is an GCC extension that will force the function to return with the **iret** instruction.

This setup makes it easy to create new ISRs in a very clean and readable fashion. For instance to create a ISR handler we can simply do:

```
IRQ_HANDLER gameControllerISR(IRQ_ARG)
{
    // Logic here
}
```

## 5.2 Loading interrupts

The final step is to tell the CPU where to find the Interrupt Descriptor Table.

This is done by the load_idt() function:

```
static inline void load_idt()
{
    typedef struct __attribute__((packed))
    {
        uint16_t limit;
        uintptr_t start;
    } IDT_REPRESENTATION;

    const IDT_REPRESENTATION rep = {(sizeof(struct IDT_entry) *
        (IDT_SIZE - 1)), (uintptr_t)IDT};

    asm("lidt %0; sti;"
        : /*No Output*/
        : "m"(*(IDT_REPRESENTATION *)&rep));
}
```

The **lidt** instruction indicates the location of the IDT table to the CPU. This information if encoded as a C struct which contains the starting address and the offset that must be added to reach the last IDT entry. Extended asm is used to inline assembly within the C code. The packed attribute instructs GCC to not add any extra padding to the struct.
Finally **sti** instruction is used to enable interrupts.

# Chapter 6

# Keyboard Driver

## 6.1 Writing and reading from ports

Before writing a device driver, the functionality to communicate with different I/O ports needs to be implemented. This is done in assembly using the **outb** and **inb** instructions. The code can be written using extended asm:

```
// Read a byte from port
inline unsigned char inportb(unsigned int port)
{
    unsigned char ret;
    asm volatile("inb %%dx,%%al"
                 : "=a"(ret)
                 : "d"(port));
    return ret;
}

// Write value to port
inline void outportb(unsigned int port, unsigned int value)
{
    asm volatile("outb %%al,%%dx"
                 :
                 : "d"(port), "a"(value));
}
```

## 6.2 Overview

When the user presses a key, an interrupt is raised which causes the keyboard ISR to be called.
This ISR instructs the keyboard driver to read the character from the keyboard port. This character is then printed to the screen and saved in a circular buffer. During scanf(), this input is read from the circular buffer.

## 6.3 Understanding Keyboard ports

The PS/2 keyboard uses two main ports:

- **0x60** - This is the data port from which characters are read.
- **0x64** - This is the command port to which commands are issued.
  Reading from this port gives status information.

| IO Port | Access Type | Purpose |
|---|---|---|
| 0x60 | Read/Write | Data Port |
| 0x64 | Read | Status Register |
| 0x64 | Write | Command Register |

Figure 6.1: Summary of keyboard ports

| Bit | Meaning |
|---|---|
| 0 | Output buffer status (0 = empty, 1 = full)<br>(must be set before attempting to read data from IO port 0x60) |
| 1 | Input buffer status (0 = empty, 1 = full)<br>(must be clear before attempting to write data to IO port 0x60 or IO port 0x64) |
| 2 | System Flag<br>Meant to be cleared on reset and set by firmware (via. PS/2 Controller Configuration Byte) if the system passes self tests (POST) |
| 3 | Command/data (0 = data written to input buffer is data for PS/2 device, 1 = data written to input buffer is data for PS/2 controller command) |
| 4 | Unknown (chipset specific)<br>May be "keyboard lock" (more likely unused on modern systems) |
| 5 | Unknown (chipset specific)<br>May be "receive time-out" or "second PS/2 port output buffer full" |
| 6 | Time-out error (0 = no error, 1 = time-out error) |
| 7 | Parity error (0 = no error, 1 = parity error) |

Figure 6.2: Keyboard Status

## 6.4 Scan code to character conversion

The data read from the keyboard data port is not an ASCII representation of the character associated with a key press. Instead the data is a scan code which must further be converted to an ASCII character. There are three scan code sets that are used by keyboards. DaxOS uses scan code set one.

As an example consider that a user presses the 'A' key on the keyboard. The data from the keyboard will be 0x1E which corresponds to 'A' on scan code set 1.

DaxOS does this conversion using a lookup table in **kbd_table.h**.

```
static const char kbd_scan_tbl[236] = {
    '\0','\0','1','2','3','4','5','6','7','8','9',
    '0','\0','\0','\b','\0','q','w','e','r','t',
    'y','u','i','o','p','\0','\0','\n','\0','a',
    's','d','f','g','h','j','k','l','\0','\0',
    '\0','\0','\0','z','x','c','v','b','n','m',
    ...
};
```

The '\0' here are for characters that are ignored.

## 6.5 Keyboard Driver API

The keyboard driver API is as follows:

```
uint8_t kbd_read_status();
uint8_t kbd_read_data();
void kbd_data_send_cmd(uint8_t cmd);
void kbd_ack();
void kbd_ctrl_send_cmd(uint8_t cmd);
void kbd_enable_interrupts();
```

# Chapter 7

# Unit Testing

Testing is an integral practice to ensure and verify that the code written solves the problem that it is trying to solve. Unit testing refers to testing the units of a program - in most cases these units are functions.

## 7.1 Challenge

An operating system is an example of a system software. It is difficult to write tests for such software. Some related difficulties are:

- Most unit testing frameworks depend on an existing standard C library.
- Lack of mature debugging utilities.
- Difficulty in getting test results across to developers.

## 7.2 Introducing DUnit

To make testing easier, DaxOS uses a custom unit testing framework called **DUnit**. The implementation is minimal and based on two assertions.

The API can summarized as follows:

```
// Assert that a condition is true
void D_UNIT_ASSERT(bool condition, const char *t_name);

// Assert that a condition is false
void D_UNIT_ASSERT_FALSE(bool condition, const char *t_name);
```

An example of an unit test written using DUnit is given below:

```
// Ensure that malloc returns NULL if we're out of memory
void D_TEST_unfulfillable_request()
{
    char *p = malloc(/* Quick way to get SIZE_MAX*/ (size_t)-1);
    D_UNIT_ASSERT_FALSE(p, "MEM/UNFULFILLABLE_MEM_REQUEST");
}
```

# Chapter 8

# Standard C Library

The C library which includes all the familiar functions such as printf, scanf, strlen and so on is not actually a part of the C language. It is instead the responsibility of the OS to provide these primitives.
DaxOS provides a partial implementation of the C library.

## 8.1 Implementation Summary

The parts of the C library that have been implemented are listed below:

- **stdio.h**
  - int **printf**(const char *, ...);
  - int **putchar**(int);
  - int **puts**(const char *);
  - int **scanf**(const char *, ...);

- **stdlib.h**
  - char ***itoa**(int, char *);
  - char ***uitoa**(unsigned int, char *);
  - void **abort**(void) __attribute__((__noreturn__));
  - void ***malloc**(size_t);
  - void ***realloc**(void *, size_t);
  - void ***calloc**(size_t, size_t);
  - void **free**(void *);

- **string.h**
  - int **memcmp**(const void *, const void *, size_t);
  - void ***memcpy**(void *__restrict, const void *__restrict, size_t);
  - void ***memmove**(void *, const void *, size_t);
  - void ***memset**(void *, int, size_t);
  - size_t **strlen**(const char *);
  - char ***strcpy**(char *destination, const char *source);
  - char ***strncpy**(char *destination, const char *source, size_t num);
  - int **strcmp**(const char *p1, const char *p2);

# Chapter 9

# Memory Management

The objective here is to support **malloc()** and **free()**. The implementation of memory management is split into two phases:

- Implementing Physical Memory allocator
- Porting liballoc

## 9.1 Physical Memory Allocator

This idea here is to allocate memory in 4kb chunks. Liballoc will use this allocator internally to provide malloc() functionality. A simplified overview of the implementation is as follows:

1. Split physical memory into blocks of size 4kb each.
2. Use a data structure to keep track of which regions are free/used.
3. Provide an API to allocate/free blocks.

These steps are elaborated in the following subsections.

### 9.1.1 Memory Map

A memory map is a data structure that gives indication about the status of different regions of physical memory. DaxOS uses multiboot to fetch the memory map.

The following multiboot struct represents information about a single memory region:

```
typedef struct multiboot_memory_map {
    uint32_t size;
    uint32_t base_addr_low, base_addr_high;
    uint32_t length_low, length_high;
    uint32_t type;
};
```

- **size**: Size of this structure.
- **base_addr_low, base_addr_high**: Starting address of this memory region.
- **length_low, length_high**: Length of this memory region in bytes.
- **type**: Status of this region. A value of '1' here indicates available.

We can configure multiboot to pass this information to our kernel by modifying **boot.S** as follows:

```
.set MEMINFO, 1<<1  # provide memory map
.set FLAGS,   ALIGN | MEMINFO
```

We can get the information of the next region by adding *size* bytes to the memory address of this record.

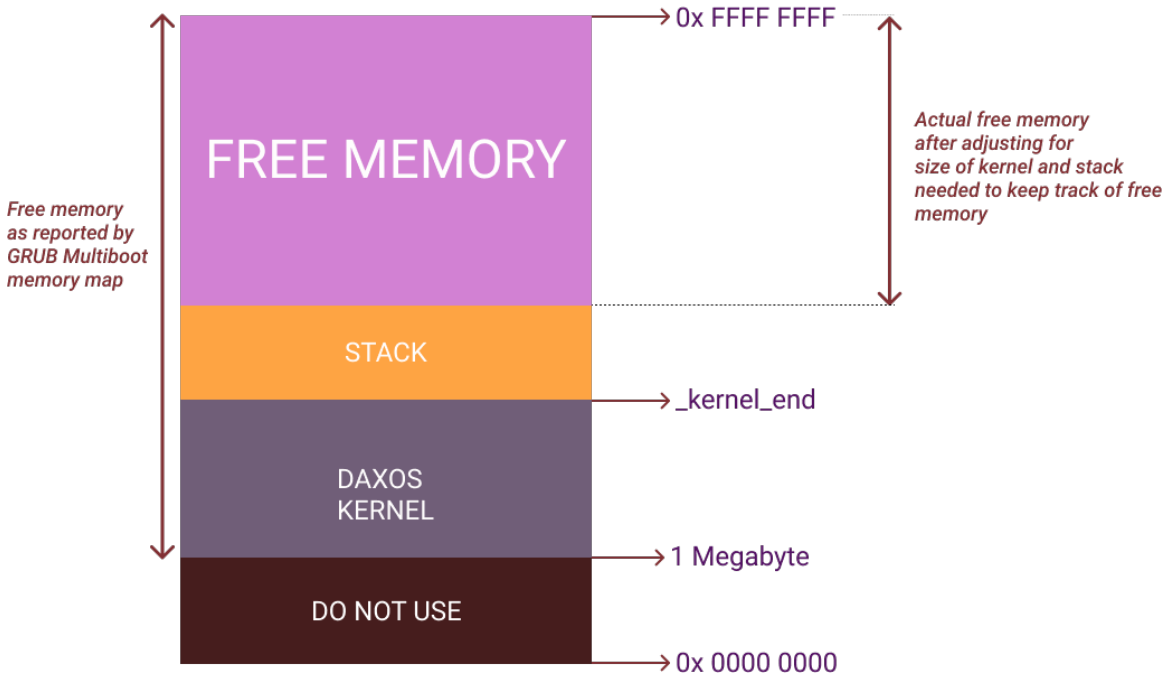


Figure 9.1: Memory Architecture in DaxOS

### 9.1.2 Keeping Track of Memory

With the memory map in possession, the next objective is to implement a data structure to keep track of this memory. There are two common options:

- **Bitmap**
- **Stack**

The disadvantage of using a bitmap is that in order to find a free 4kb block of memory, the entire data structure needs to be scanned. This incurs a run-time complexity of O(n). Using a stack is O(1) since all that is required to find a free block is simply a pop operation. Therefore DaxOS uses the stack approach.

The stack implementation supports two calls:

```
void push(uint32_t mem_addr);
int32_t pop();
```

### 9.1.3 API

The API consists of two calls:

```
uint32_t* get_page();
void free_page(uint32_t *page);
```

Their description is as follows:

- **get_page()**: returns the memory address of a free 4kb block of memory.
- **free_page()**: returns the block to the free list.

## 9.2 Porting Liballoc

Liballoc is a free and open-source memory allocator for use in hobbyist OS. It provides malloc and other related functions. However to use liballoc, DaxOS must implement four hooks. Hooks are functions that liballoc uses internally that must be provided by the OS.

The hooks are:

```
extern int liballoc_lock();
extern int liballoc_unlock();
extern void *liballoc_alloc(size_t);
extern int liballoc_free(void *, size_t);
```

Their descriptions are as follows:

- **Lock and unlock** are used to provide mutex locks.
  Although DaxOS is not multi-threaded it still has concurrency in the form of interrupts. Therefore the lock function simply disables interrupts whereas the unlock function enables interrupts. This is done using the *cli* and *sti* instructions respectively.
- **alloc** function allocates multiple blocks of memory as specified in the parameter.
- **free** function frees multiple blocks of memory starting from the address specified in parameter.

These functions are implemented using the functions in the physical memory allocator. At this point, implementation of memory management is complete.

# Chapter 10

# CPU Exceptions

These are a kind of interrupt that signals that an exception or error has occurred. Most CPU exceptions are faults i.e they are recoverable. However some CPU exceptions are aborts i.e the OS cannot continue after they occur.

In DaxOS, every CPU exception leads to a kernel panic. Kernel panic involves showing a screen with the exception information and then stopping the OS.

## 10.1 Supported Exceptions

DaxOS supports the following CPU exceptions:

- **Divide By Zero**

  This occurs when a number is divide by zero.

- **Debug**

  This is triggered when single step mode is enabled.

- **Non Maskable Interrupt**

  This occurs in response to a non-recoverable hardware fault.

- **Breakpoint**

  This is triggered by running INT3 instruction.

- **Overflow**

  This occurs when the overflow bit in flag is set and INTO instruction is given.

- **Bound Range Exceeded**

  Occurs in response to a BOUND instruction.

- **Invalid Opcode**

  Triggered when a nonexistent opcode is used.

- **Device Not Available**

  Triggered when a Floating Point Unit is not available.

## 10.2 Code Generation

Code for handling two cpu exceptions are given below:

```
IRQ_HANDLER cpu_exception_0(IRQ_ARG)
{
    send_eoi();
    k_panic("Divide By Zero");
}

IRQ_HANDLER cpu_exception_1(IRQ_ARG)
{
    send_eoi();
    k_panic("Debug");
}
```

One obvious problem is that the code is duplicated. The two interrupt handlers are similar except for the string argument in k_panic function. This is a clear violation of the DRY principle which states that code must not be duplicated.

DaxOS solves this issue by generating the duplicated code using a python script. The python script produces the code needed for exception handling in two files: exception.c and exception.h, both of which can be configured to be produced at any location.

# Chapter 11

# Graphics Support

Graphics support involves having the ability to draw pixels to the screen. Graphics in DaxOS includes:

- 1280x720p resolution
- Font rendering
- Image rendering

## 11.1 Enabling Graphics

DaxOS initially used VGA text mode for printing text to the screen. Therefore to get graphics mode, multiboot must be configured as follows:

```
.set FLAGS, ALIGN | MEMINFO | VBEINFO
.set MODE_TYPE, 0
.set WIDTH, 1024
.set HEIGHT, 768
.set DEPTH, 32
```

The **MODE_TYPE** indicates that we want to be able to draw pixels instead of text and **depth** indicates that each pixel requires 32 bits for representation. Thus each pixel follows the ARGB form where:

- A is alpha transparency.
- R, G and B are red, green and blue components respectively.

Now we can draw pixels to the screen by writing to the framebuffer address.

## 11.2 Font Rendering

With VGA text-mode, the complexity of rendering fonts were handled by the BIOS. However, in graphics mode, DaxOS must implement its own font renderer. The idea here is that in order to print a letter to the screen, a bitmap of the letter must be generated from the font and it must be copied into the framebuffer.

The sequence of steps to be followed to get the font bitmap is as follows:

- Choose any bitmap based font.
- Convert the font to .ssfn format.
- Convert the .ssfn file into object file.
- Link the font file with the OS.

To render fonts to the screen DaxOS uses a modified verson of a font renderer called scalable screen font renderer. The modifications introduced include:

- Refactoring to split ssfn.h into corresponding .h and .c file.
- Removing unwanted functionality from the source.

## 11.3 Image Rendering

DaxOS support rendering images in Truevision TGA format. The image is converted to .tga format and linked with the OS as described previously. Rendering images is then a two step process:

- Decode the TGA header to get the image metadata.
- Iterate and copy pixel data of image to framebuffer.

The tga header is as follows:

```
typedef struct
{
    unsigned char magic1;          // must be zero
    unsigned char colormap;        // must be zero
    unsigned char encoding;        // must be 2
    unsigned short cmaporig, cmaplen; // must be zero
    unsigned char cmapent;         // must be zero
    unsigned short x;              // must be zero
    unsigned short y;              // image's height
    unsigned short w;              // image's width
    unsigned short h;              // image's height
    unsigned char bpp;             // must be 32
    unsigned char pixeltype;       // must be 40
} __attribute__((packed)) tga_header_t;
```

Of all the fields, the image width and height are most important. This header is located at the start of the image. Therefore skipping over this header gives the pixel data of the image. Copying pixel data to the framebuffer is then done using memcpy function.


Figure 11.1: Image rendering in DaxOS

# Chapter 12

# Conclusion

This report has attempted to provide a sufficient and elaborate introduction to DaxOS.
The implementation has been discussed along with code samples where applicable.
The authors of this report hope that the explanations were clear and succinct.

Interested readers can find the source code of this project at
https://github.com/DaxKernel/OS.
You can also check out our website at https://daxkernel.github.io/.

# REFERENCES


[1] https://wiki.osdev.org/Bare_Bones

[2] http://www.osdever.net/tutorials/

[3] https://wiki.osdev.org/Interrupts_tutorial

[4] https://wiki.osdev.org/User:Omarrx024/VESA_Tutorial

[5] https://wiki.osdev.org/Exceptions

[6] https://wiki.osdev.org/Scalable_Screen_Font

[7] https://wiki.osdev.org/PS/2_Keyboard

[8] https://www.gnu.org/software/grub/manual/multiboot/multiboot.html