RV College of Engineering®

# (Autonomous Institution Affiliated to VTU, Belagavi)



**SUBJECT: BUILDING OS FROM SCRATCH**

Submitted by:

|  |  |
| --- | --- |
| SAI VARU KONDA | 1RV22CS171 |
| SAHIL S NAIK | 1RV22CS169 |

Submitted to

# Prof. JYOTHI SHETTY

# DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

# RV COLLEGE OF ENGINEERING, BANGALORE-5

**CONTENT**

* Problem statement
* Tools used
* Goals
* Procedure
* Addressing
* Operating System hardware
* Code
* Conclusion
* References

**PROBLEM STATEMENT**

Design and implement a minimal operating system capable of performing basic tasks such as process management, memory management, and file system operations. The operating system should be able to boot on standard PC hardware and provide a command-line interface for user interaction. Key functionalities to include bootstrapping, printing, grub, memory management.

**TOOLS**

* QEMU
* NASM
* BUILD
* GCC
* VSCODE
* MAKEFILE

‘

**GOALs**

* To build an basic cli interface os
* It should have basic grub control
* Is should perfor basic functions such as printing, reading.
* Drivers such as keyboard.
* PCI

**PROCEDURE**

Step1:

Using basic bios interrupt functionalities, such as print on tty, reading in assembly, waiting, changing from text to vga, vice versa.

Step2:

Creating gdt

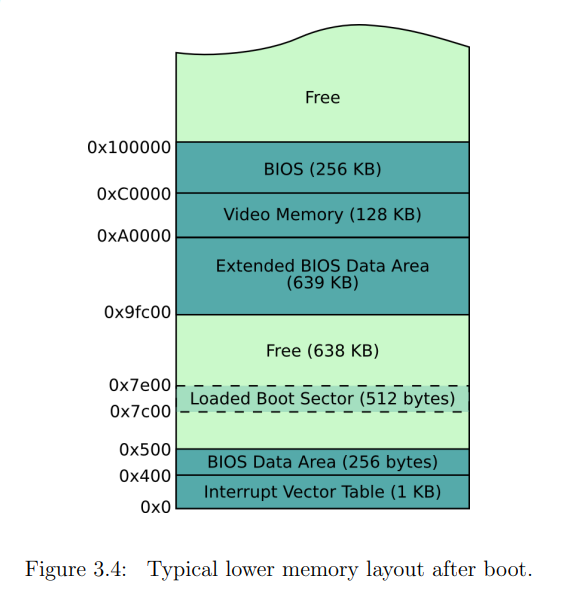
Step3:

loading kernel at location 1000. Handing over the control to the kernel.

Step4:

Define various functionalities in kernel such as drivers, stdlib where we can use function for allocating. Designing drivers, configuring pcis

**ADDRESSING**

****

**OPERATING SYSTEM HARDWARE**

The framework we have used here is x86 64 bit.

It contains registers such as

EAX (Accumulator): Used for arithmetic operations.

EBX (Base): Used as a pointer to data.

ECX (Counter): Used in shift/rotate instructions and loops.

EDX (Data): Used in arithmetic and I/O operations.

ESI (Source Index): Pointer to a source in stream operations.

EDI (Destination Index): Pointer to a destination in stream

CS (Code Segment): Points to the code segment.

DS (Data Segment): Points to the data segment.

SS (Stack Segment): Points to the stack segment.

ES (Extra Segment): Additional data segment.

EFLAG

IP

ALU

**CODE**

include "stdlib.h"

// Base method for itoa

char\* \_itoa(int integer, char\* result, int base) {

int num\_digits = 0;

int val = 0;

// Push each digit

while (integer > 0) {

// Mod 10 gets just the last digit

// Adding '0' turns it into an ASCII character

val = integer % base;

if (val < 10) {

result[num\_digits] = val + '0';

} else {

result[num\_digits] = val + 'a';

}

num\_digits++;

// Divide by 10 to get the next character in line

integer /= base;

}

// Reverse it and return it

char tmp = 0;

for (int i = 0; i < num\_digits/2; i++) {

// Swap at each end of the array

tmp = result[i];

result[i] = result[num\_digits-i-1];

result[num\_digits-i-1] = tmp;

}

// Null terminate the string

if (num\_digits == 0) {

num\_digits = 1;

result[0] = '0';

}

result[num\_digits] = '\0';

return result;

}

// Integer to ASCII

char itoa\_buffer[256];

char\* itoa(int integer) {

return \_itoa(integer, \*itoa\_buffer, 10);

}

// Integer to ASCII HEX

char\* itoah(int integer) {

return \_itoa(integer, \*itoa\_buffer, 16);

}

// Integer to ASCII Binary

char\* itoab(int integer) {

return \_itoa(integer, \*itoa\_buffer, 2);

}

bool streq(char\* string1, char\* string2) {

int i = 0;

while (true) {

// Char mismatch? false

if (string1[i] != string2[i]) {

return false;

}

// They're equal, AND it's null char? We're done

if (string1[i] == '\0') {

break;

}

i++;

}

return true;

}

bool safe\_streq(char\* string1, int str1len, char\* string2, int str2len) {

if (str1len != str2len) return false;

for (int i = 0; i < str1len; i++) {

if (string1[i] != string2[i]) return false;

}

return true;

}

void terrible\_sleep\_impl(int ticks) {

volatile int i = 0;

while (i < ticks\*100000) i++;

}

GDT

GDT - Global Descriptor Table

gdt\_start:

gdt\_null: ; Entry 1: Null entry must be included first (error check)

dd 0x0 ; double word = 4 bytes = 32 bits

dd 0x0

gdt\_code: ; Entry 2: Code segment descriptor

; Structure:

; Segment Base Address (base) = 0x0

; Segment Limit (limit) = 0xfffff

dw 0xffff ; Limit bits 0-15

dw 0x0000 ; Base bits 0-15

db 0x00 ; Base bits 16-23

; Flag Set 1:

; Segment Present: 0b1

; Descriptor Privilege level: 0x00 (ring 0)

; Descriptor Type: 0b1 (code/data)

; Flag Set 2: Type Field

; Code: 0b1 (this is a code segment)

; Conforming: 0b0 (Code w/ lower privilege may not call this)

; Readable: 0b1 (Readable or execute only? Readable means we can read code constants)

; Accessed: 0b0 (Used for debugging and virtual memory. CPU sets bit when accessing segment)

db 10011010b ; Flag set 1 and 2

; Flag Set 3

; Granularity: 0b1 (Set to 1 multiplies limit by 4K. Shift 0xfffff 3 bytes left, allowing to span full 32G of memory)

; 32-bit default: 0b1

; 64-bit segment: 0b0

; AVL: 0b0

db 11001111b ; Flag set 3 and limit bits 16-19

db 0x00 ; Base bits 24-31

gdt\_data:

; Same except for code flag:

; Code: 0b0

dw 0xfffff ; Limit bits 0-15

dw 0x0000 ; Base bits 0-15

db 0x00 ; Base bits 16-23

db 10010010b ; Flag set 1 and 2

db 11001111b ; 2nd flags and limit bits 16-19

db 0x00 ; Base bits 24-31

gdt\_end: ; Needed to calculate GDT size for inclusion in GDT descriptor

; GDT Descriptor

gdt\_descriptor:

dw gdt\_end - gdt\_start - 1 ; Size of GDT, always less one

dd gdt\_start

; Define constants

CODE\_SEG equ gdt\_code - gdt\_start

DATA\_SEG equ gdt\_data - gdt\_start

kernel

bits 32

section .multiboot

dd 0x1BADB002 ; Magic number

dd 0x0 ; Flags

dd - (0x1BADB002 + 0x0) ; Checksum

section .text

; Include the GDT from previous tutorials

; Set this as our GDT with LGDT

; insetad of relying on what the bootloader sets up for us

%include "src/kernel/gdt.asm"

; Make global anything that is used in C files

global start

global load\_gdt

global load\_idt

global keyboard\_handler

global ioport\_in

global ioport\_out

global inl

global outl

global enable\_interrupts

extern main ; Defined in kernel.c

extern handle\_keyboard\_interrupt

load\_gdt:

lgdt [gdt\_descriptor] ; from gdt.asm

ret

load\_idt:

mov edx, [esp + 4]

lidt [edx]

ret

enable\_interrupts:

sti

ret

keyboard\_handler:

pushad

cld

call handle\_keyboard\_interrupt

popad

iretd

ioport\_in:

mov edx, [esp + 4] ; PORT\_TO\_READ, 16 bits

; dx is lower 16 bits of edx. al is lower 8 bits of eax

; Format: in <DESTINATION\_REGISTER>, <PORT\_TO\_READ>

in al, dx ; Read from port DX. Store value in AL

; Return will send back the value in eax

; (al in this case since return type is char, 8 bits)

ret

ioport\_out:

mov edx, [esp + 4] ; port to write; DST\_IO\_PORT. 16 bits

mov eax, [esp + 8] ; value to write. 8 bits

; Format: out <DST\_IO\_PORT>, <VALUE\_TO\_WRITE>

out dx, al

ret

inl:

mov edx, [esp + 4]

in eax, dx

ret

outl:

mov edx, [esp + 4]

mov eax, [esp + 8]

out dx, eax

ret

start:

lgdt [gdt\_descriptor]

jmp CODE\_SEG:.setcs ; Set CS to our 32-bit flat code selector

.setcs:

mov ax, DATA\_SEG ; Setup the segment registers with our flat data selector

mov ds, ax

mov es, ax

mov fs, ax

mov gs, ax

mov ss, ax

mov esp, stack\_space ; set stack pointer

cli ; Disable interrupts

mov esp, stack\_space

call main

hlt

section .bss

resb 8192 ; 8KB for stack

stack\_space:

**CONCLUSION**

We have learnt from this how the main os interracts with the hardware, how the low level language works. How we can use the interrupt to create various functions. Usage of register and stack as a parameters.

**REFERENCES**

Writing a Simple Operating System — from Scratch by Nick Blundel

[Creating an Operating System - OSDev Wiki](https://wiki.osdev.org/Creating_an_Operating_System)