1. INTRODUCTION
   1. BACKGROUND

Kernel development is not an easy task. This is a testament to your programming expertise: To develop a kernel is to say that you understand how to create software that interfaces with and manages the hardware. A kernel is designed to be a central core to the operating system - the logic that manages the resources that the hardware has to offer.

One of the most important system resources that you need to manage is the processor or CPU - this is in the form of allotting time for specific operations, and possibly interrupting a task or process when it is time for another scheduled event to happen. This implies multitasking. There is cooperative multitasking, in which the program itself calls a 'yield' function when it wants to give up processing time to the next runnable process or task. There is preemptive multitasking, where the system timer is used to interrupt the current process to switch to a new process: a form of forced switch, this more guarantees that a process can be given a chunk of time to run. There are several scheduling algorithms used in order to find out what process will be run next. The simplest of which is called 'Round Robin'. A more complicated scheduler involves 'priorities', where certain higher-priority tasks are allowed more time to run than a lower-priority task. Even more complicated still is a Real-time scheduler. This is designed to guarantee that a certain process will be allowed at least a set number of timer ticks to run.

The next most important resource in the system is fairly obvious: Memory. There are some times where memory can be more precious than CPU time, as memory is limited, however CPU time is not. The best approach would be a combination of the two: Strive for the best memory usage, while preserving CPU time.

The last resource that your kernel needs to manage are hardware resources. This includes Interrupt Requests (IRQs), which are special signals that hardware devices like the keyboard and hard disk can use to tell the CPU to execute a certain routine to handle the data that these devices have ready. Another hardware resource is Direct Memory Access (DMA) channels. A DMA channel allows a device to lock the memory bus and transfer it's data directly into system memory whenever it needs to, without halting the processor's execution.

* 1. PROBLEM DEFINATION

This project is targeted towards developing a functional microkernel with a command-line front end to its end-users. This microkernel along with the CLI is code named as “COLDWAVE OS” by it developers. The major challenges that one may face in developing a kernel from scratch is the support for any runtime libraries, since one cannot make use of any runtime libraries while making his own OS. The only alternative that is left with the developers is to write an entire C library from scratch that would aid the execution of C code that gets its implementation from libraries that has now have become a part of the kernel. Other major drawbacks that kernel developers may face is unable to debug the exact cause of an improper kernel build. This is a vital problem that often may last for weeks until carefully rectified by its developers. Our microkernel is proposed to be a kernel following a microkernel architecture with minimal functionality i.e. it includes the most features that makes it functional.

A microkernel in itself doesn’t have any prime functionality to perform unlike its monolithic kernel counterpart. Hence, we add extra functionality to our microkernel in the form of servers which we implement in the user mode and just the kernel being in the kernel or privileged mode.

The various problems associated with setting up our kernel are as follows:

i: Setting the kernel entry with a multiboot header for GRUB to identify it.

ii: Setting up a custom and basic Global Descriptor Table for memory access violation.

iii: Setting up a custom and basic Interrupt Descriptor Table for suitable to possible interrupts raised by the system or the user program.

iv:Setting Interrupt Service Routines (ISRs) to handle Interrupts and IRQs.  
v: Remapping Programmable Interrupt Controllers (PICs) to new IDT entries  
vi: Installing and servicing IRQs.  
vii: Managing the Programmable Interval Timer / System Clock (PIT).  
viii: Managing Keyboard IRQs and Keyboard Data.

ix: Creating a user-shell program to allow interaction between user and the kernel.

1.3 SOFTWARE

Compilers

* The Gnu C Compiler(GCC) [Unix]
* DJGPP (GCC for DOS/Windows) [Windows]

Assemblers

* Netwide assembler(NASM) [Unix/Windows]
* Microsoft assembler(MASM) [Windows]

Virtual Machines

* VMWare Workstation 4.0.5 [Linux/Windows NT/2000/XP]
* Microsoft Virtual PC [Windows NT/2000/XP]
* Bochs [Unix/Windows]

1.4 HARDWARE

* A 100% IBM compatible PC
* A Pentium 2 or K6 300MHz processor
* 32 Mbytes of RAM
* A VGA compatible video card with monitor
* A keyboard
* A Floppy Drive
* A hard disk with enough space for all development tools and space for

documents and source code

* Microsoft Windows or a flavor of Unix(Linux , Free BSD)
* An internet connect to look up documents(A mouse is highly recommended)

2. BOOTLOADER

* 1. BOOTLOADER FUNCTION

The bootloader plays a very crucial role right from loading the kernel into the primary memory to initializing various parameter that are needed by every kernel for the successful sum of the system.

A bootloader is thus a special program needed to successfully load our kernel to the memory location 0x7C00. The system BOIS loads a valid boot sector, also called the MBR (Master Boot Record), which is typically 512 bytes in size, and is the name given to the first sector of any device. However a the first sector of any device are not bootable by default, and we need to make use of some assembly routines to render them as bootable, which is detected by the bootstrap program in the system BIOS.

A typical bootloader can be divided into several stages. And its not mandatory for the entire bootloader to occupy only the bootsector. But, there is a provision for dividing the bootloader into several stages,by putting the basic bootloader in the first sector, and which in turn is responsible for loading the bootloader in the other sector of its containing device.

For our kernel,we are using GRUB[GRand Unified Bootloader], which is a multiboot loader i.e able to load several operating system either directly or through chain-loading.

The Multi-boot loader must accomplish the following tasks:

1. Locate the Multi-boot header in the preloaded kernel image.

2. Verify the Multi-boot header & flags.

3. Load the kernel image into high memory.

4. Write the Multi-boot information structure.

5. Return success or failure to boot1.

* 1. TYPES OF BOOTLOADER

Bootloaders have grown enormously in their features as well as their functionality.Bootloaders are largely classified based on the way they manage the loading and running several OS kernels simultaneously without any loss in performance.

They are basically of two types:

1.Single or dedicated Bootloaders

2.Multiboot Bootloaders

A single/dedicated bootloader is a bootloader that are targeted towards a specific kernel and can be used only for the same.

A multi-boot bootloader is targeted to boot several different kinds of kernel that either support multi-boot headers in its implementation definition, or use chain-loading functionality.Muti-bootloaders are used widely nowadays as compared to monolithic bootloaders and the most common bootloaders in this genre are GRUB,LILO,PUPA etc.

Here, is a typical illustration of a bootable floppy disk ,along with the kernel.



Fig: 2.2

The following are stages that a bootloader goes through during execution.

1. Disable interrupts.

2. Canonicalize %CS:%EIP.

3. Load segment registers (%DS, %ES, %FS, %GS, %SS).

4. Set the stack pointer.

5. Query the BIOS for the size of lower memory.

6. Query the BIOS for the size of upper memory.

7. Read kernel sectors from the floppy into lower memory.

8. Enable the A20 gate.

9. Disable interrupts.

10. Load the Global Descriptor Table.

11. Switch to protected mode.

12. Invoke the Multiboot loader.

13. Begin execution of the kernel.

In case, the above booting step fails , the booting fails as follows:

1. Notifying the user of a failure condition.

2. Disable interrupts (if not already disabled).

3. Permanently suspend progress in execution.

3. KERNEL

* 1. INTRODUCTION TO KERNEL

The term kernel is the name given to the program that is loaded by our bootloader into the primary memory, which intern loads the various additional libraries and ,modules that aid in the execution of an operating system.

An Operating System’s kernel can be called as the heart of the kernel as is certainly the program that runs at layer 0 ,and hence has full access to all the hardware resources that is available on a particular machine.

When the bootloader loads the kernel into the primary memory, we are said to be in the real mode,which is outside any human intervention, and this real-mode environment is helpful in setting up the interactive environment in protected mode ,where by the users are free to explore the features of specific kernels.

A typical kernel has the responsibilities of initializing all the segment registers of the microprocessors with the appropriate values, set up the values in the Global Descriptor Table, set up a Interrupt Descriptor Table, Interrupt Service Routines, Programmable Interrupt Timers, Video Graphics Adapter etc.

Also, we need to make use of assembly programming language to work with the specific segment registers and the main memory to store the values necessary to initialize this devices, also since the devices in Intel x86 architecture follows memory-mapping, we need to read or write specific instructions to these address that point to a particular specialized device in the memory.

An OS kernel can be either monolithic (self-contained) or possess a micro-kernel architecture whereby the devices drivers lie at a layer higher than the basic kernel layer or the layer 0, this helps in a more structured, faster but blotted kernel when compared to its monolithic counterpart.

* 1. TYPES OF KERNEL

• Monolithic Kernel

• Micro Kernel

• Exo Kernel

• Nano Kernel

• Macro Kernel

• Hybrid kernel etc.

* 1. KERNEL FUNCTIONALITY

The kernel is the core-component and it manages everything that relies on the kernel for its services.The primary purpose of the kernel is to manage the system resources and provide interface so that other program can access these resources.

The following are the functions that are handled by an operating systems kernel:

1. Memory Management

2. Program Management

3. Multitasking

4. Memory Protection

5. Interrupt Handling

6. Kernel

7. File Systems

8. Interrupt Handling

The above features are the chief and primary functionality that every good kernel provides to its users, and make working with the system much easier.

A micokernel in itself doesn’t have any prime functionality to perform unlike its monolithic kernel counterpart. Hence, we add extra functionality to our microkernel in the form of servers which we implement in the user mode and just the kernel being in the kernel or privileged mode.

4. LINKER AND ASSEMBLER

4.1 LINKER

The linker is a very useful utility, that helps us bind the object codes from various related objects or libraries to produce a single executable bundle, that forms the root to entire program. The linker being a small and powerful utility performs its functions by binding the object codes produced from the compilation and assembly phase respectively.

We are making use of the GNU LD program as our linker , along with a linker script to guide the linking process in accordance with the kernel executable.

The linker follows the following format for linking:

Cmd>ld <scr\_swt> <linker\_script> <out\_swt><out\_name> < … >

Eg: ld –T link.ld -o kernel.bin start.o main.o scrn.o console.o …

4.2 ASSEMBLER

The assembler is a program that is useful to assemble code written in assembly language to code that can be interpreted by a bare machine.

Assembly languages have a syntax very close to the instruction set of a typical micro-processor and hence is useful in writing the code for low-level programming for managing the hardware resources.

Assembly languages are always platform-dependent languages and are usually hard and difficult to program in.They, must be utilized to the minimum amount, and the rest must be handled by other high-level programming languages.

We make use of NASM[Netwide Assembler ] as our assembler program and the other alternatives are YASM,TASM,MASM,GAS etc.

Nasm follows the following format for assembling:

Cmd>nasm <fmt\_swt><out\_fmt><out\_swt><out\_file><src\_file\_name>

Eg: nasm -f aout –o start.o start.asm

5. MICRO KERNEL

* 1. INTRODUCTION TO MICRO KERNEL

Microkernel is a specialized kernel writing technique that is categorized by minimal set of hardware abstraction. Microkernels provide a small set of simple hardware abstraction and uses applications known as servers to provide additional functionality to the kernel program.

In this architecture, only a minimal part of the operating system will run in the kernel or supervisor mode. As compared to monolithic kernel, the primary task of any typical microkernel is resource management and the flexibility in which the above feature is achieved. Microkernel architecture has loosely coupled structure with a client-server model of communication between its layers.

Though monolithic kernel has a significant advantage over all other kernel architecture in terms of performance, but modularity is the key element that gives power to a microkernel. Microkernels are flexible and can be easily extended to our desired conditions.

Monolithic kernel has a significant advantage over all other kernel architecture in terms of performance, but modularity is the key element that gives power to a microkernel. Microkernels are flexible and can be easily extended to our desired conditions.

Minimalistic Microkernel Development

Following is an illustration on comparison between microkernel and monolithic kernel:

Table 5.1



* 1. BENEFITS OF MICROKERNEL

Microkernel architecture provides minimum functionality to the kernel, and all completely relies on user-space servers to provide functionality.

The functionality that can be realized in different manner by following microkernel architecture is:

* **Single Server Operating System**: In this model a microkernel runs an entire monolithic OS as an ordinary user program. This setup does not change any of the properties of the monolithic kernel, this feature helps us the preserve the UNIX like environment even while experimenting a microkernel. E.g.: BSD, OSF/1.
* **Multiserver Operating System**: In a multiserver OS , the operating system environment is caused by a number of cooperative servers. Depending on the functionality provided by these multiserver environments legacy software may still be provided backward compatibility.E.g: GNU Hurd on top of Mach microkernel.
* **Dedicated Systems**: This organization makes it possible to run specialized applications on top of the microkernel. This kind of organization is useful for mobile and embedded devices with reduced power consumption.

E.g.: Java Virtual Machine(JVM)



Monolithic Micro Exo

Fig: 5.2

6. COLDWAVE MICRO KERNEL

6.1 INTRODUCTION TO MICRO KERNEL

The cold-wave microkernel is a Toy OS that follows the microkernel architecture, and is developed as part of our entitled project. The cold-wave OS scope and functionality is very minimal in its approach and the developer of the project has taken utmost care to minimize the complexity of the system to be re-usable by students or researchers in Operating System Development.

The cold-wave microkernel does not make use of any inbuilt bootloader for loading the kernel to the main memory and makes use of GRUB, as its bootloader ,which is a multiboot compliant bootloader.Also,there are several added advantages when using GRUB as our bootloader since they have the default implementation of a file system, GDT table and many more.

The cold-wave microkernel makes use of a single assembly file to handle all the details required regarding initialization and storage in 8086 primary registers another additional registers.This is where we handle the GDT,IDT,ISR and decleration of a stack for the purpose of storing local varibles,function parameters and address off function themselves.This single assembly file also contains a multiboot header that provides the

necessary details to the bootloader which is GRUB in our case to load the kernel properly.

The kernel program constitutes of a typical C/C++ main function, and is the only block that should contain all the necessary elements to deal with our operating system. An exception in our kernel main function is that in its case the main shall not return a value. And, usually end up in an infinite loop in which context multitasking can be achieved.

The cold-wave kernel is very flexible and is provided with rich set of functions and headers files that are pre-build and can be used to run applications that are created as part of our kernel.

6.2 INTRODUCTION TO MICRO KERNEL

The cold-wave kernel was developed with the intension to provide its developers and user’s a basic understanding of how a typical kernel achieves all the tasks that a user assigns to it.

The cold-wave microkernel though being small and compact in size , is powerful and flexible. Its has several features that mimics the implementation of several standard operating systems.The cold-wave kernel comes with a command-line interface (CLI), that can be used by the end-users to interact with the system and explore the power of this kernel.

Following are the basic features that cold-wave kernel possess:

* Global Descriptor Table [GDT]
* Interrupt Descriptor Table [IDT]
* Interrupt Serveice Routines [ISR]
* Remapped Programmable Controller
* Programmable Interrupt Timer/System Clock
* Keyboard I/O and Miniature C/C++ library
* File System
* Memory Manager
* Command Line Interface [CLI]

The following is a illustration of the block diagram of cold-wave OS:

Keyboard

Video I/O

Memory Manager

User Space

User Shell

I/O Subsystem

File System

KERNEL [GDT, IDT, Timer, ISR, IRQs, PIC]

HARDWARE

Fig: 6.2

6.3 KERNEL ENTRY PROGRAM

**Source Code: start.asm**

The kernel's entry point is the piece of code that will be executed first when the bootloader calls your kernel. This chunk of code is almost always written in assembly language because some things, such as setting a new stack or loading up a new GDT, IDT, or segment registers, are things that you simply cannot do in your C code. In many beginner kernels as well as several other larger, more professional kernels, will put all of their assembler code in this one file, and put all the rest of the sources in several C source files.

As far as code goes, all this file does is load up a new 8KByte stack, and then jump into an infinite loop. The stack is a small amount of memory, but it's used to store or pass arguments to functions in C. It's also used to hold local variables that you declare and use inside your functions.

Any other global variables are stored in the data and BSS sections. The lines between the 'mboot' and 'stublet' blocks make up a special signature that GRUB uses to verify that the output binary that it's going to load is, infact, a kernel

This kernel program that is the program that is first visible to the GRUB bootloader, which identifies by checking the special multiboot signature in this file, and next it gives

a call to main function which is the beginning of the Cold-Wave kernel, after this call to the main function the control is explicitly transferred to the kernel program, that loads the important and vital modules needed for the running of our kernel.

This program is also useful for setting up the Global Descriptor Table as well as the Interrupt Descriptor Table, to be used as part of the kernel.

The kernel entry program is also the region where possible interrupt service routines are defined , to handle any software interrupts generated by the system.

6.4 GLOBAL DESCRIPTOR TABLE

**Source Code: gdt.c**

A vital part of the 386's various protection measures is the Global Descriptor Table, otherwise called a GDT. The GDT defines base access privileges for certain parts of memory. We can use an entry in the GDT to generate segment violation exceptions that give the kernel an opportunity to end a process that is doing something it shouldn't.

The GDT is also capable of defining what are called Task State Segments (TSSes), which is one of the ways to achieve multi-tasking.

The GDT itself is a list of 64-bit long entries. These entries define where in memory that the allowed region will start, as well as the limit of this region, and the access privileges associated with this entry. One common rule is that the first entry in your GDT, entry 0, is known as the NULL descriptor. No segment register should be set to 0, otherwise this will cause a General Protection fault, and is a protection feature of the processor.

Each GDT entry's Access and Granularity fields can be defined as follows:

|  |  |  |
| --- | --- | --- |
|  |  |  |

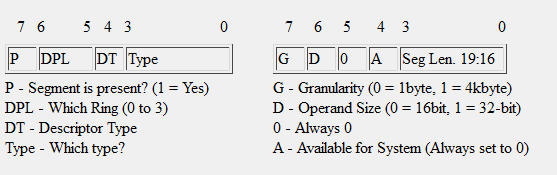


Fig: 6.4

In our kernel, we will create a GDT with only 3 entries.One 'dummy' descriptor, to act as our NULL segment for the processor's memory protection features. Another entry for the Code Segment, and finallyone entry try for the Data Segment registers. To tell the processor where our new GDT table is, we use the assembly opcode 'lgdt'. 'lgdt' needs to be given a pointer to a special 48-bit structure.

6.5 INTERRUPT DESCRIPTOR TABLE

**Source Code: idt.c**

The Interrupt Descriptor Table, or IDT, is used in order to show the processor what Interrupt Service Routine (ISR) to call to handle either an exception or an 'int' opcode (in assembly). IDT entries are also called by Interrupt Requests whenever a device has completed a request and needs to be serviced.

Each IDT entry is similar to that of a GDT entry. Both have hold a base address, both hold an access flag, and both are 64-bits long. The major differences in these two types of descriptors is in the meanings of these fields.

In an IDT, the base address specified in the descriptor is actually the address of the Interrupt Service Routine that the processor should call when this interrupt is 'raised' (called). An IDT entry doesn't have a limit, instead it has a segment that you need to specify. The segment must be the same segment that the given ISR is located in. This allows the processor to give control to the kernel through an interrupt that has occured when the processor is in a different ring.

The access flags of an IDT entry are also similar to a GDT entry's. There is a field to say if the descriptor is actually present or not. There is a field for the Descriptor Privilege Level (DPL) to say which ring is the highest number that is allowed to use the given

given interrupt. The major difference is the rest of the access flag definition. The lower 5-bits of the access byte is always set to 01110 in binary. This is 14 in decimal. Here is a table to give you a better graphical representation of the access byte for an IDT entry.

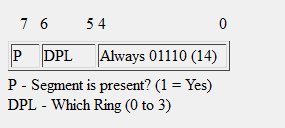


Fig: 6.5

6.6 INTERRUPT SERVICE ROUTINES

**Source Code: isrs.c**

Interrupt Service Routines, or ISRs, are used to save the current processor state and set up the appropriate segment registers needed for kernel mode before the kernel's C-level interrupt handler is called. This can all be handled in about 15 or 20 lines of assembly language, including calling our handler in C. We need to also point the correct entry in the IDT to the correct ISR in order to handle the right exception.

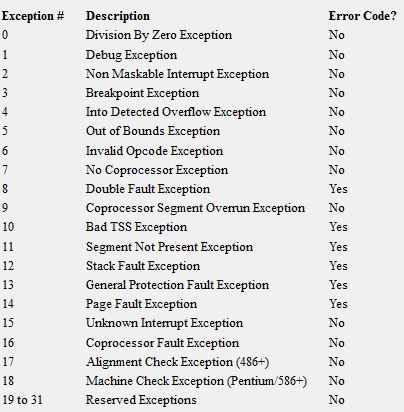
An Exception is a special case that the processor encounters when it cannot continue normal execution. This could be something like dividing by zero: The result is an unknown or non-real number, so the processor will cause an exception so that the kernel can stop that process or task from causing any problems. If the processor finds that a program is trying to access a piece of memory that it shouldn't, it will cause a General Protection Fault. When you set up paging, the processor causes a Page Fault, but this is recoverable: you can map a page in memory to the faulted address - but that's for another tutorial.

The first 32 entries in the IDT correspond to Exceptions that can possibly be generated by the processor, and therefore need to be handled. Some exceptions will push another value onto the stack: an Error Code value which is specific to the exception caused.

Some exceptions push an error code onto the stack. To decrease the complexity, we handle this by pushing a dummy error code of 0 onto the stack for any ISR that doesn't push an error code already. This keeps a uniform stack frame. To track which exception is firing, we also push the interrupt number on the stack. We use the assembler opcode 'cli' to disable interrupts and prevent an IRQ from firing, which could possibly otherwise cause conflicts in our kernel. To save space in the kernel and make a smaller binary output file, we get each ISR stub to jump to a common 'isr\_common\_stub'. The 'isr\_common\_stub' will save the processor state , the stack, push the current stack address onto the stack (gives our C handler the stack), call our C 'fault\_handler' function, and finally restore the state of the stack. Add this code to 'start.asm' in the provided space, filling out all 32 ISRs.

Here, is a listing of the first 32 types of interrupts that be raised by the processor, which if not handled will leave the system into triple fault stage.

Table: 6.6



6.7 PROGRAMMABLE INTERRUPT CONTROLLER

**Source Code: irq.c**

Interrupt Requests or IRQs are interrupts that are raised by hardware devices. Some devices generate an IRQ when they have data ready to be read, or when they finish a command like writing a buffer to disk, for example. It's safe to say that a device will generate an IRQ whenever it wants the processor's attention. IRQs are generated by everything from network cards and sound cards to your mouse, keyboard, and serial ports.

Any IBM PC/AT Compatible computer (anything with a 286 and later processor) has 2 chips that are used to manage IRQs. These 2 chips are known as the Programmable Interrupt Controllers or PICs. These PICs also go by the name '8259'. One 8259 acts as the 'Master' IRQ controller, and one is the 'Slave' IRQ controller. The slave is connected to IRQ2 on the master controller. The master IRQ controller is connected directly to the processor itself, to send signals. Each PIC can handle 8 IRQs. The master PIC handles IRQs 0 to 7, and the slave PIC handles IRQs 8 to 15. Remember that the slave controller is connected to the primary controller through IRQ2: This means that every time an IRQ from 8 to 15 occurs, IRQ2 fires at exactly the same time.

When a device signals an IRQ, remember that an interrupt is generated, and the CPU pauses whatever it's doing to call the ISR to handle the corresponding IRQ. The CPU then performs whatever necessary action (like reading from the keyboard, for example), and then it must tell the PIC that the interrupt came from that the CPU has finished executing the correct routine. The CPU tells the right PIC that the interrupt is complete by writing the command byte 0x20 in hex to the command register for that PIC. The master PIC's command register exists at I/O port 0x20, while the slave PIC's command register exists at I/O port 0xA0.

Before we get into writing our IRQ management code, we need to also know that IRQ0 to IRQ7 are originally mapped to IDT entries 8 through 15.

6.8 PROGRAMMABLE INTERRUPT TIMER

**Source Code: timer.c**

The Programmable Interval Timer (PIT, model 8253 or 8254), also called the System Clock, is a very useful chip for accurately generating interrupts at regular time intervals. The chip itself has 3 channels: Channel 0 is tied to is tied to IRQ0, to interrupt the CPU at predictable and regular times, Channel 1 is system specific, and Channel 2 is connected to the system speaker.

The only channels that you should every be concerned with are Channels 0 and 2. You may use Channel 2 in order to make the computer beep. We are only concerned with Channel 0 - mapped to IRQ0. This single channel of the timer will allow you to accurately schedule new processes later on, as well as allow the current task to wait for a certain period of time (as will be demonstrated shortly). By default, this channel of the timer is set to generate an IRQ0 18.222 times per second.

To set the rate at which channel 0 of the timer fires off an IRQ0, we must use our outportb function to write to I/O ports. There is a Data register for each of the timer's 3 channels at 0x40, 0x41, and 0x42 respectively, and a Command register at 0x43. The data rate is actually a 'divisor' register for this device. The timer will divide it's input clock of

1.19MHz (1193180Hz) by the number you give it in the data register to figure out how many times per second to fire the signal for that channel.

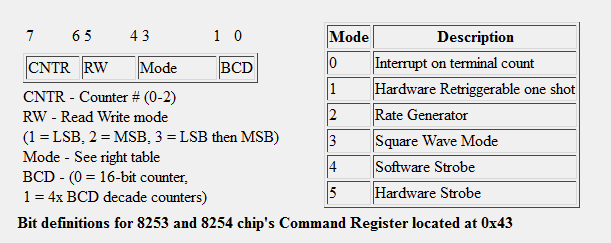


Fig: 6.8

6.8 KEYBOARD CONTROLLER

**Source Code: kb.c**

A keyboard is the most common way for a user to give a computer input, therefore it is vital that you create a driver of some sort for handling and managing the keyboard.

A scancode is simply a key number. The keyboard assigns a number to each key on the keyboard; this is your scancode. The scancodes are numbered generally from top to bottom and left to right, with some minor exceptions to keep layouts backwards compatible with older keyboards. You must use a lookup table (an array of values) and use the scancode as the index into this table. The lookup table is called a keymap, and will be used to translate scancodes into ASCII values rather quickly and painlessly. If bit 7 is set (test with 'scancode & 0x80'), then this is the keyboard's way of telling us that a key was just released.

Converting a scancode to an ASCII value is easy with this:

mychar = kbdus[scancode];

The keyboard is attached to the computer through a special microcontroller chip on your mainboard. This keyboard controller chip has 2 channels: one for the keyboard, and one for the mouse. Also note that it is through this keyboard controller chip that you would enable the A20 address line on the processor to allow you to access memory past the 1MByte mark (GRUB enables this, you don't need to worry about it). The keyboard controller, being a device accessible by the system, has an address on the I/O bus that we can use for access and control. The keyboard controller has 2 main registers: a Data register at 0x60, and a Control register at 0x64. Anything that the keyboard wants to send the computer is stored into the Data register. The keyboard will raise IRQ1 whenever it has data for us to read.

6.9 CONSOLE FUNCTIONS

**Source Code: cons.c and conf.c**

The above source programs contribute to the user console application, which provides a basic command-line interface for the end-users to interact with the cold-wave microkernel. The command line prompt provided by cold-wave right now is still in its development stage and will be enhanced in features in functionalities for the time to come.

The end user front-end console program is handled by the above two mentioned sources. The file “cons.c” is involved with the responsibility of setting up the basic environment needed to implement the command-line interface to be used by the users. The specific method that are implemented in this files are regarding declaring a global structure format for accepting specific command formats, and mapping them to specialized methods that will handle the execution of the command that it is originally intended to do.

The functions specific to particular command are defined in the function “conf.c”.The functions defined in this files corresponds to the command entry that exists globally for our user-shell application.It is mandatory that the commands need to be defined within these sources ,are declared in ascending alphabetical order, since the environment implementation makes use of binary search methodologies to compare the command typed by an user at the prompt matches any of the command defined as part of the cold-wave OS.

The command prompt in cold-wave OS has the following syntax:

Commands are:

about clear echo exit help

version day date time

wave>

Fig: 6.9

7. COLDWAVE LIBRARIES

* 1. CONF.H

This header file provides the complete set of functions and a user can avail by using command specific to this command implementation.

Functions:

extern void shellExit(short EXIT\_STATUS);

extern void CmdNotImplemented();

extern void listUCTCmd();

extern void echo(char \* str);

extern void version();

extern void about();

Functions Description:

void CmdNotImplemented()

This function displays interactive messages,when command was not found

void echo(char \* str)

This method echo’s the text you type following the echo command.

void listUCTCmd()

This function displays a list of available commands,used with help command.

void shellExit(short EXIT\_STATUS)

This function is useful to exit from the command-line.

* 1. CONF.H

This header file provides the complete set of functions needed to implement the command line interface and command repository.

Functions:

extern void initUCT();

extern void execUCTCmd(char \* cmd, char \* parameters);

extern void listUCTCmd();

extern void CmdNotImplemented();

extern char \*readline();

extern int LoadConsole();

Functions Description:

void initUCT()

initialize the UCT (user command table), be sure that all the commands in the UCT table are sorted by alphabetic order.

void execUCTCmd(char \* cmd, char \* parameters)

This function displays interactive messages,when command was not found.

char \*readline()

This function reads a line of input from the user.

int LoadConsole()

This function sets up the command-line interface to our kernel.

Void listUCTCmd()

This method lists the commands available to the terminal users.

* 1. IO.H

This header file provides the complete set of functions to perform I/O functionality.

Functions:

extern void add(char c);

extern char getkey(void);

extern void scan(char \* cmd);

Data Structure:

Struct iobuff;

Variable:

Struct iobuff kbbuff;

Functions Description:

void add(char c)

This function is used to append characters to the keyboard buffer.

char getkey()

This function pops out the character that is beginning of the buffer.

void scan(char\* cmd)

This function scans a string that is entered in the terminal.

* 1. KB.H

This header file provides the complete set of functions required to I/O using a PS/2 type keyboard.

Functions:

extern char \* Hexconvert(int val);

extern void KbHandler();

Functions Description:

char \* Hexconvert(int val)

This function is used to convert an integer to corresponding hexadecimal code

void KbHandler()

This function is needed to interface the keyboard to the user kernel.

* 1. MEMORY.H

This header file provides the complete set of functions required to perform memory related manipulation.

Functions:

extern void memcopy(void \*str1, void \*str2, int n);

extern void memset(void \*dest,int size,char car);

extern int writeMemToFile();

extern void \* malloc(uint32\_t size);

extern void free(void \* address);

extern void memory\_allocator\_init();

extern unsigned long int mem\_get\_free\_space();

Functions Description:

void\* malloc(uint32\_t size)

This function malloc , is used in dynamic allocation of free memory.

void memset(void \*dest,int size,char car)

This function is useful to set default values for specific memory locations.

void memcopy(void \*str1,void \*str2,int n)

This function is useful to copy a memory location to another.

void free(void \* address)

This method de-allocates memory that was allocated previously using malloc.

7.6 MEMORY\_ALLOC.H

This header file provides the complete set of functions to manipulate the memory locations.

Functions:

extern int mem\_firstFit\_init(void \* pool, unsigned long int mem\_siz);

extern int mem\_firstFit\_free(void \* p);

extern void mem\_firstFit\_fragment();

extern void \* mem\_firstFit\_alloc(unsigned long int size);

extern void \* mem\_firstFit\_set\_chunk\_size (void \* p,long int size);

extern unsigned long int mem\_firstFit\_get\_chunk\_size (void \* p);

Functions Description:

void \* mem\_firstFit\_alloc(unsigned long int size)

This function looks for free chunks,and provides with space more than needed.

extern void \* mem\_firstFit\_alloc(unsigned long int size)

This function checks that free and adjacent chunks are not grouped.

extern void \* mem\_firstFit\_set\_chunk\_size (void \* p,unsigned long int size)

This function specifies the size of the allocated space.

extern int mem\_firstFit\_init(void \* pool, unsigned long int \_\_mem\_size)

This method initializes the memory allocator.

7.7 STDIO.H

This header file provides the complete set of functions that implement I/O in standard I/O devices.

Functions:

extern void gotoxy(const char x, const char y);

extern void delline(int y);

extern void gotoxy(const char x, const char y);

extern void setTextColor(char color);

extern void clrscr();

Functions Description:

void gotoxy(const char x, const char y)

This function is useful to move to (x,y) co-ordinate in display device.

void settextcolor(const char x,const char y)

This function sets the back and foreground of the kernel environment.

void delline(int y)

This function deletes a line on pressing backspace.

void clrscr()

This function clears the screen , and prints the prompt at begin of line.

7.8 STR.H

This header file provides the complete set of functions needed to manipulate string related data.

Functions:

extern char \* strcat(char \* destination,char \* source);

extern char \* strchr(char \* s,int c);

extern int strcmp(char \* s1,char \* s2);

extern inline char \* strcpy(char s1[], const char s2[]);

extern inline char \* strcopy(char s1[], const char s2[]);

extern char \* Sreadline();

extern char \* readline();

Functions Description:

char \* strcat(char \* destination,char \* source)

This function is ued to bind the source and destination strings.

int strcmp(char\* s1,char\* s2)

This function is used to compare the values between two strings.

char \* strchr(char \* s,int c)

This function searches for the occurance of charcter in the string.

char \* Sreadline()

This method reads lines ending with a ‘/r’,and max limit is 255 characters.

char \* readline()

This method reads lines ending with a ‘/r’,and may cause an overflow.

7.9 SYSTEM.H

This header file combines the functions from all the constituent modules that as a whole make up the kernel.

Functions:

extern void logo();

/\* MAIN.C \*/

extern void \*memcpy(void \*dest, const void \*src, size\_t count);

extern void \*memset(void \*dest, char val, size\_t count);

extern unsigned short \*memsetw(unsigned short \*dest, unsigned short val, size\_t count);

extern size\_t strlen(const char \*str);

extern unsigned char inportb (unsigned short \_port);

extern void outportb (unsigned short \_port, unsigned char \_data);

/\* CONSOLE.C \*/

extern void init\_video(void);

extern void puts(unsigned char \*text);

extern void putch(unsigned char c);

extern void cls();

extern void printf (const char \*format, ...);

extern void itoa (char \*buf, int base, int d);

/\* GDT.C \*/

extern void gdt\_set\_gate(int num, unsigned long base, unsigned long limit, unsigned char access, unsigned char gran);

extern void gdt\_install();

/\* IDT.C \*/

extern void idt\_set\_gate(unsigned char num, unsigned long base, unsigned short sel, unsigned char flags);

extern void idt\_install();

/\* ISRS.C \*/

extern void isrs\_install();

/\* IRQ.C \*/

extern void irq\_install\_handler(int irq, void (\*handler)(struct regs \*r));

extern void irq\_uninstall\_handler(int irq);

extern void irq\_install();

/\* TIMER.C \*/

extern void timer\_wait(int ticks);

extern void timer\_install();

/\* KEYBOARD.C \*/

extern void keyboard\_install();

7.10 INCLUDE FILES DEPENDENCY HIERARCHY

The following hierarchy contains the dependency among several include header files that are part of the “COLDWAVE OS”.

SYSTEM.H

TYPES.H

TIME.H

CONSOLE.H

KB.H

MEMORY.H

STRING.H

IO.H

STDIO.H

MEM\_ALLOC.H

CONF.H

Fig: 7.10

8. INTEL x86 ARCHITECTUERE

8.1 INTEL x86 INTRODUCTION

The x86 architecture first appeared as the Intel 8086 CPU released in 1978, a fully 16-bit design based on the earlier 8-bit based 8008 and 8080. Although not binary compatible, it was designed to allow assembly language programs written for these processors (as well as the contemporary 8085) to be mechanically translated into equivalent 8086 assembly. This made the new processor a tempting software migration path for many customers. However, the 16-bit external databus of the 8086 implied fairly significant hardware redesign, as well as other complications and expenses. To address this obstacle, Intel introduced the almost identical 8088, basically an 8086 with an 8-bit external databus that permitted simpler printed circuit boards and demanded fewer (1-bit wide) DRAM chips; it was also more easily interfaced to already established (i.e. low-cost) 8-bit system and peripheral chips. Among other, non-technical factors, this contributed to IBM's decision to build a home computer / personal computer around the 8088, despite a presence of 16-bit microprocessors from Motorola, Zilog, and National Semiconductor (as well as several established 8-bit processors, which was also considered). The resulting IBM PC subsequently took over from Z80-based CP/M systems, Apple IIs, and other popular computers, and became a dominant de-facto standard for personal computers, thus enabling the 8088 and its successors to dominate this large branch of the microprocessor market.

* + 1. OVERVIEW

The x86 architecture is a variable instruction length, primarily two-address "CISC" design with emphasis on backward compatibility. The instruction set is not typical CISC, however, but basically an extended and orthogonalized version of the simple eight-bit 8008 and 8080 architectures. Memory access to unaligned addresses is allowed for all supported word sizes.

Dedicated floating point processor with 80-bit internal registers, the 8087, was developed for the original 8086. This chip subsequently developed into the extended 80387, and later processors incorporated a backward compatible version of this functionality on the same chip as the main processor.

* + 1. SEGMENTATION

The original 8086, developed from the simple 8080 microprocessor and primarily aiming at very small and inexpensive computers and other specialized devices, instead adopted simple segment registers which increased the memory address width by only 4 bits. By multiplying a 64-KB address by 16, the 20-bit address could address a total of one megabyte (1,048,576 bytes) which was quite a large amount for a small computer at the time. The concept of segment registers was not new to many mainframes which used segment registers to quickly swap to different tasks.

Data and/or code could be managed within "near" 16-bit segments within this 1 MB address space, or a compiler could operate in a "far" mode using 32-bit segment:offset pairs reaching (only) 1 MB.

* + 1. ADDRESSING MODES

Addressing modes for 16-bit x86 processors can be summarized by this formula:


\begin{Bmatrix}CS:\\DS:\\SS:\\ES:\end{Bmatrix}
\begin{bmatrix}\begin{Bmatrix}BX\\BP\end{Bmatrix}\end{bmatrix} +
\begin{bmatrix}\begin{Bmatrix}SI\\DI\end{Bmatrix}\end{bmatrix} +
\rm [displacement]


Addressing modes for 32-bit address size on 32-bit or 64-bit x86 processors can be summarized by this formula:


\begin{Bmatrix}CS:\\DS:\\SS:\\ES:\\FS:\\GS:\end{Bmatrix}
\begin{bmatrix}\begin{Bmatrix}EAX\\EBX\\ECX\\EDX\\ESP\\EBP\\ESI\\EDI\end{Bmatrix}\end{bmatrix} +
\begin{bmatrix}\begin{Bmatrix}EAX\\EBX\\ECX\\EDX\\EBP\\ESI\\EDI\end{Bmatrix}*\begin{Bmatrix}1\\2\\4\\8\end{Bmatrix}\end{bmatrix} +
\rm [displacement]


Addressing modes for 64-bit code on 64-bit x86 processors can be summarized by this formula: 
\begin{Bmatrix}
\begin{Bmatrix}FS:\\GS:\end{Bmatrix}
\begin{bmatrix}{\rm general\;register}\end{bmatrix} +
\begin{bmatrix}{\rm general\;register}*\begin{Bmatrix}1\\2\\4\\8\end{Bmatrix}\end{bmatrix}\\\\
RIP
\end{Bmatrix} +
\rm [displacement]


Instruction relative addressing in 64-bit code (RIP + displacement, where RIP is the instruction pointer register) simplifies the implementation of position-independent code (as used in shared libraries in some operating systems).

* + 1. X86 Registers

**16 bit registers:**

The original Intel 8086 and 8088 have fourteen 16-bit registers. Four of them (AX, BX, CX, DX) are general-purpose registers (GPRs; although each may have an additional purpose: for example only CX can be used as a counter with the loop instruction). Each can be accessed as two separate bytes (thus BX's high byte can be accessed as BH and low byte as BL). There are two pointer registers: SP which points to the top of the stack and BP (base pointer) which is used to point at some other place in the stack, typically above the local variables. Two registers (SI and DI) are for array indexing.

**32 bit registers:**

With the advent of the 32-bit 80386 processor, the 16-bit general-purpose registers, base registers, index registers, instruction pointer, and FLAGS register, but not the segment registers, were expanded to 32 bits. This is represented by prefixing an "E" (for Extended) to the register names in x86 assembly language. Thus, the AX register corresponds to the lowest 16 bits of the new 32-bit EAX register, SI corresponds to the lowest 16 bits of ESI, and so on.

**64 bit registers:**

Starting with the AMD Opteron processor, the x86 architecture extended the 32-bit registers into 64-bit registers in a way similar to how the 16 to 32-bit extension was done (RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, RFLAGS, RIP), and eight additional 64-bit general registers (R8-R15) were also introduced in the creation of x86-64. However, these extensions are only usable in 64-bit mode, which is one of the two modes only available in long mode.

* + 1. x86 Register Structure

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| General Purpose Registers (A, B, C and D) | | | | | | | |
| **64** | **56** | **48** | **40** | **32** | **24** | **16** | **8** |
| R?X | | | | | | | |
|  | | | | E?X | | | |
|  | | | | | | ?X | |
|  | | | | | | ?H | ?L |

Fig: 5.1.5(a)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 64-bit mode-only General Purpose Registers (R8, R9, R10, R11, R12, R13, R14, R15) | | | | | | | |
| **64** | **56** | **48** | **40** | **32** | **24** | **16** | **8** |
| ? | | | | | | | |
|  | | | | ?D | | | |
|  | | | | | | ?W | |
|  | | | | | | | ?B |

Fig: 5.1.5(b)

8.2 INTEL x86 MODES

8.2.1 REAL MODE

Real mode is an operating mode of 8086 and later x86-compatible CPUs. Real mode is characterized by a 20 bit segmented memory address space (meaning that only 1 MB of memory can be addressed), direct software access to BIOS routines and peripheral hardware, and no concept of memory protection or multitasking at the hardware level. All x86 CPUs in the 80286 series and later start up in real mode at power-on; 80186 CPUs and earlier had only one operational mode, which is equivalent to real mode in later chips.

In order to use more than 64 KB of memory, the segment registers must be used. This created great complications for compiler implementors who introduced odd pointer modes such as "near", "far" and "huge" to leverage the implicit nature of segmented architecture to different degrees, with some pointers containing 16-bit offsets within implied segments and other pointers containing segment addresses and offsets within segments.

8.2.2 PROTECTED MODE

In addition to real mode, the Intel 80286 supports protected mode, expanding addressable physical memory to 16 MB and addressable virtual memory to 1 GB, and providing protected memory, which prevents programs from corrupting one another. This is done by using the segment registers only for storing an index to a segment table. There were two such tables, the Global Descriptor Table (GDT) and the Local Descriptor Table (LDT), each holding up to 8192 segment descriptors, each segment giving access to 64 KB of memory. The segment table provided a 24-bit base address, which can be added to the desired offset to create an absolute address. Each segment can be assigned one of four ring levels used for hardware-based computer security.

The Intel 80386 introduced support in protected mode for paging, a mechanism making it possible to use paged virtual memory.

8.2.3 VIRTUAL 8086 MODE

There is also a sub-mode of operation in 32-bit protected mode, called virtual 8086 mode. This is basically a special hybrid operating mode that allows real mode programs and operating systems to run while under the control of a protected mode supervisor operating system. This allows for a great deal of flexibility in running both protected mode programs and real mode programs simultaneously. This mode is exclusively available for the 32-bit version of protected mode; virtual 8086 mode does not exist in the 16-bit version of protected mode, or in long mode.

8.3 INTEL x86 INSTRUCTION SET

This is the full 8086/8088 instruction set, but most, if not all of these instructions are available in 32-bit mode, they just operate on 32-bit registers (eax, ebx, etc.) and values instead of their 16-bit (ax, bx, etc.) counterparts.

Table 8.2

**Original 8086/8088 instructions**

|  |  |  |
| --- | --- | --- |
| **Instruction** | **Meaning** | **Notes** |
| AAA | ASCII adjust AL after addition | used with unpacked [binary coded decimal](http://en.wikipedia.org/wiki/Binary_coded_decimal) |
| AAD | ASCII adjust AX before division | 8086/8088 datasheet documents only base 10 version of the AAD instruction ([opcode](http://en.wikipedia.org/wiki/Opcode" \o "Opcode) 0xD5 0x0A), but any other base will work. Later Intel's documentation has the generic form too. NEC V20 and V30 (and possibly other NEC V-series CPUs) always use base 10, and ignore the argument, causing a number of incompatibilities |
| AAM | ASCII adjust AX after multiplication | Only base 10 version is documented, see notes for AAD |
| AAS | ASCII adjust AL after subtraction |
| ADC | Add with carry | destination := destination + source + [carry\_flag](http://en.wikipedia.org/wiki/Carry_flag) |
| ADD | Add |  |
| AND | [Logical AND](http://en.wikipedia.org/wiki/Logical_conjunction) |  |
| CALL | Call procedure |  |
| CBW | Convert byte to word |  |
| CLC | Clear [carry flag](http://en.wikipedia.org/wiki/Carry_flag) |  |
| CLD | Clear [direction flag](http://en.wikipedia.org/wiki/Direction_flag) |  |
| [CLI](http://en.wikipedia.org/wiki/CLI_%28x86_instruction%29) | Clear [interrupt flag](http://en.wikipedia.org/wiki/IF_%28x86_flag%29) |  |
| CMC | Complement carry flag |  |
| CMP | Compare operands |  |
| CMPSB | Compare bytes in memory |  |
| CMPSW | Compare words |  |
| CWD | Convert word to doubleword |  |
| [DAA](http://en.wikipedia.org/wiki/Intel_BCD_opcodes) | Decimal adjust AL after addition | (used with packed [binary coded decimal](http://en.wikipedia.org/wiki/Binary_coded_decimal)) |
| [DAS](http://en.wikipedia.org/wiki/Intel_BCD_opcodes) | Decimal adjust AL after subtraction |  |
| DEC | Decrement by 1 |  |
| DIV | Unsigned divide |  |
| ESC | Used with [floating-point unit](http://en.wikipedia.org/wiki/Floating-point_unit) |  |
| [HLT](http://en.wikipedia.org/wiki/HLT) | Enter halt state |  |
| IDIV | Signed divide |  |
| IMUL | Signed multiply |  |
| IN | Input from port |  |
| INC | Increment by 1 |  |
| [INT](http://en.wikipedia.org/wiki/INT_%28x86_instruction%29) | Call to [interrupt](http://en.wikipedia.org/wiki/Interrupt) |  |
| INTO | Call to interrupt if overflow |  |
| IRET | Return from interrupt |  |
| Jxx | Jump if condition | (*JA, JAE, JB, JBE, JC, JCXZ, JE, JG, JGE, JL, JLE, JNA, JNAE, JNB, JNBE, JNC, JNE, JNG, JNGE, JNL, JNLE, JNO, JNP, JNS, JNZ, JO, JP, JPE, JPO, JS, JZ*) |
| [JMP](http://en.wikipedia.org/wiki/JMP_%28x86_instruction%29) | Jump |  |
| LAHF | Load flags into AH register |  |
| LDS | Load pointer using DS |  |
| LEA | Load Effective Address |  |
| LES | Load ES with pointer |  |
| LOCK | Assert BUS LOCK# signal | (for multiprocessing) |
| LODSB | Load string byte |  |
| LODSW | Load string word |  |
| LOOP/LOOPx | Loop control | (*LOOPE, LOOPNE, LOOPNZ, LOOPZ*) |
| [MOV](http://en.wikipedia.org/wiki/MOV_%28x86_instruction%29) | Move |  |
| MOVSB | Move byte from string to string |  |
| MOVSW | Move word from string to string |  |
| MUL | Unsigned multiply |  |
| NEG | Two's complement negation |  |
| [NOP](http://en.wikipedia.org/wiki/NOP) | No operation | opcode (0x90) equivalent to XCHG EAX, EAX |
| NOT | Negate the operand, logical NOT |  |
| OR | [Logical OR](http://en.wikipedia.org/wiki/Logical_disjunction) |  |
| OUT | Output to port |  |
| POP | Pop data from [stack](http://en.wikipedia.org/wiki/Stack_%28data_structure%29) | POP CS (opcode 0x0F) works only on 8086/8088. Later CPUs use 0x0F as a prefix for newer instructions. |
| POPF | Pop data into [flags register](http://en.wikipedia.org/wiki/FLAGS_register_%28computing%29) |  |
| PUSH | Push data onto stack |  |
| PUSHF | Push flags onto stack |  |
| RCL | Rotate left (with carry) |  |
| RCR | Rotate right (with carry) |  |
| REPxx | Repeat MOVS/STOS/CMPS/LODS/SCAS | (*REP, REPE, REPNE, REPNZ, REPZ*) |
| RET | Return from procedure |  |
| RETN | Return from near procedure |  |
| RETF | Return from far procedure |  |
| ROL | Rotate left |  |
| ROR | Rotate right |  |
| SAHF | Store AH into flags |  |
| SAL | [Shift Arithmetically](http://en.wikipedia.org/wiki/Arithmetic_shift) left (signed shift left) |  |
| SAR | Shift Arithmetically right (signed shift right) |  |
| SBB | Subtraction with borrow | alternative 1-byte encoding of SBB AL, AL is available via [undocumented](http://en.wikipedia.org/wiki/X86_instruction_listings#Undocumented_instructions) SALC instruction |
| SCASB | Compare byte string |  |
| SCASW | Compare word string |  |
| SHL | [Shift](http://en.wikipedia.org/wiki/Logical_shift) left (unsigned shift left) |  |
| SHR | Shift right (unsigned shift right) |  |
| STC | Set carry flag |  |
| STD | Set direction flag |  |
| [STI](http://en.wikipedia.org/wiki/STI_%28x86_instruction%29) | Set interrupt flag |  |
| STOSB | Store byte in string |  |
| STOSW | Store word in string |  |
| SUB | Subtraction |  |
| [TEST](http://en.wikipedia.org/wiki/TEST_%28x86_instruction%29) | Logical compare (AND) |  |
| WAIT | Wait until not busy | Waits until BUSY# pin is inactive (used with [floating-point unit](http://en.wikipedia.org/wiki/Floating-point_unit)) |
| XCHG | Exchange data |  |
| XLAT | Table look-up translation |  |
| XOR | [Exclusive OR](http://en.wikipedia.org/wiki/Exclusive_or) |  |

9. FURTHER EXTENSIONS

9.1 VIRTUAL FILE SYSTEM AND PAGING

In addition to our existing kernel we may add additional functionality to our kernel by adding a virtual file system service on top of our kernel that will allow us to mount devices with different file formats as part of our local file system transparently, which will result into better storage and sharing of resources.

A virtual file system (VFS) or virtual filesystem switch is an abstraction layer on top of a more concrete file system. The purpose of a VFS is to allow client applications to access different types of concrete file systems in a uniform way. A VFS can, for example, be used to access local and network storage devices transparently without the client application noticing the difference. It can be used to bridge the differences in Windows, Mac OS and Unix filesystems, so that applications can access files on local file systems of those types without having to know what type of file system they are accessing.

A VFS specifies an interface (or a "contract") between the kernel and a concrete file system. Therefore, it is easy to add support for new file system types to the kernel simply by fulfilling the contract.

In addition to VFS, we may implement the concept of paging as part of our OS.In computer operating systems, paging is one of the memory-management schemes by which a computer can store and retrieve data from secondary storage for use in main memory. In the paging memory-management scheme, the operating system retrieves data from secondary storage in same-size blocks called pages. The main advantage of paging over memory segmentation is that it allows the physical address space of a process to be noncontiguous. Before the time paging was used, systems had to fit whole programs into storage contiguously, which caused various storage and fragmentation problems.

Paging is an important part of virtual memory implementation in most contemporary general-purpose operating systems, allowing them to use disk storage for data that does not fit into physical random-access memory (RAM).

9.2 I/O INTERFACING OF MOUSE AND USB

Till now we have added only keyboard integration as part of our kernel, that helps us manage the interaction of the command- line of the user kernel. Mouse Integration is necessary for an operating system that comes with a rich graphical user interface to interact with its end users.

Mouse integration with our present kernel can be tricky if not implemented carefully, since all the devices in the x86 architecture, are memory-mapped devices, we need to read and write at this location to work with the mouse. A mouse works by mapping the x-coordinate and the y-coordinate against the user screen.

Moreover, we can custom make the pointer for our mouse that we make use as a pointing device. Also, we may add support for USB devices to our OS .

Universal Serial Bus (USB) is an industry standard developed in the mid-1990s that defines the cables, connectors and communications protocols used in a bus for connection, communication and power supply between computers and electronic devices.

USB was designed to standardize the connection of computer peripherals, such as keyboards, pointing devices, digital cameras, printers, portable media players, disk drives and network adapters to personal computers, both to communicate and to supply electric power. It has become commonplace on other devices, such as smartphones, PDAs and video game consoles.USB has effectively replaced a variety of earlier interfaces, such as serial and parallel ports, as well as separate power chargers for portable devices.

9.3 INTEGRATED GRAPHICS LIBRARY

A graphics library is a program library designed to aid in rendering computer graphics to a monitor. This typically involves providing optimized versions of functions that handle common rendering tasks. This can be done purely in software and running on the CPU, common in embedded systems, or being hardware accelerated by a GPU, more common in PCs. By employing these functions, a program can assemble an image to be output to a monitor. This relieves the programmer of the task of creating and optimizing these functions, and allows them to focus on building the graphics program.

The following are the standard graphics libraries available:

Cairo (graphics)

Direct3D

MiniGL

OpenGL

WebGL

OpenGL ES

Open Inventor

We may make use of the above libraries or develop one from scratch to add interactive graphics to the kernel.

CONCLUSION

Thus, we have successfully developed a basic kernel with a user shell as an aim towards educating young minds on kernel development and system programming or as a reference for Operating System researchers who want to study the structure of the implemented system. Our Operating System was developed keeping minimum functionality in mind, and hence comes with a basic set of commands .The operating system was implemented with a modular architecture and hence is very flexible and reliable to add new features to it without much complexity.

The next thing should think of writing is a memory manager. A memory manager will allow you to grab chunks of memory so that you can dynamically allocate and free memory as you need it. Using a memory manager, you can use more complicated data structures such as linked lists and binary trees to allow for more efficient storage and manipulation of data. It's also a way of preventing applications from writing to kernel pages, which is a feature of protection.

It's possible to write a VGA driver, also. Using a VGA driver, you can set up different graphics modes in your kernel, allowing higher resolutions and graphical display options such as buttons and images. If you want to go further, you could eventually look into VESA video modes for high color and higher resolutions.

You could eventually write a device interface which would allow you to load or unload kernel 'modules' as you need them. Add support for filesystems and disk drives so that you can access files off disks and open applications.

BIBLIOGRAPHY

Web Resources:

http://www.wiki.osdev.org

http://www.minix3.org

http://www.osdever.net

http://www.nondot.org/sabre/os/articles

http://www.linuxfromscratch.org/lfs/view/6.5/index.html

http://www.helenos.org

http://oss.sgi.com/LDP/HOWTO/Kernel-HOWTO/index.html

http://www.helenos.org/

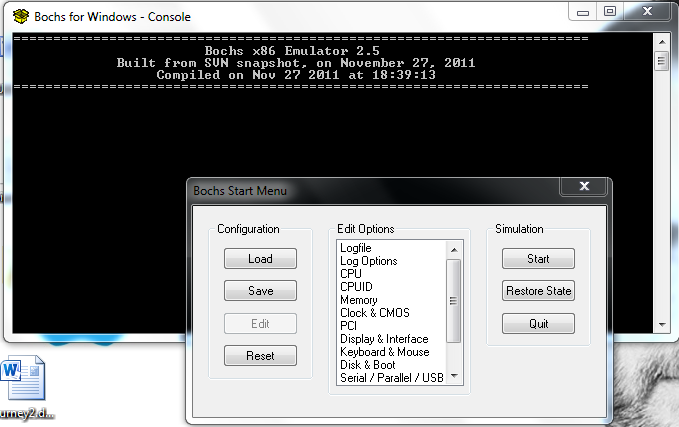
Text-Book References:

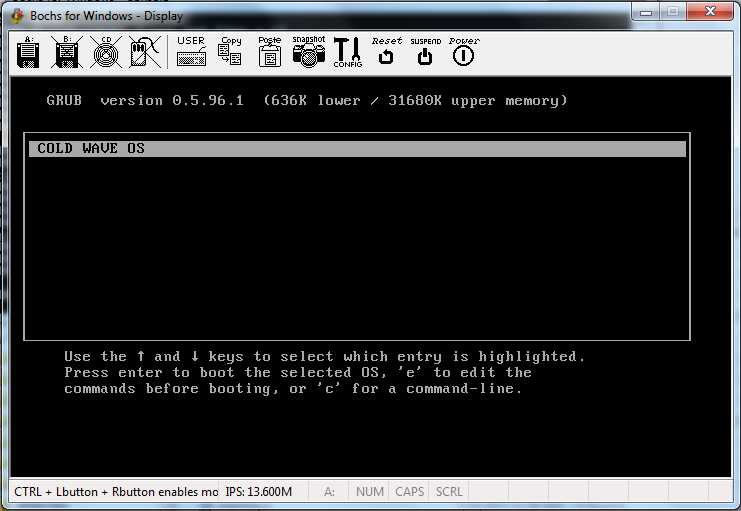
* Operating System Implementation and Design by Andrew S. Tanenbaum
* The Design and Implementation of the FreeBSD Operating System
* Modern Operating Systems by Andrew S. Tanenbaum
* Intel x32 Architecture vol 1 & 2 for developers

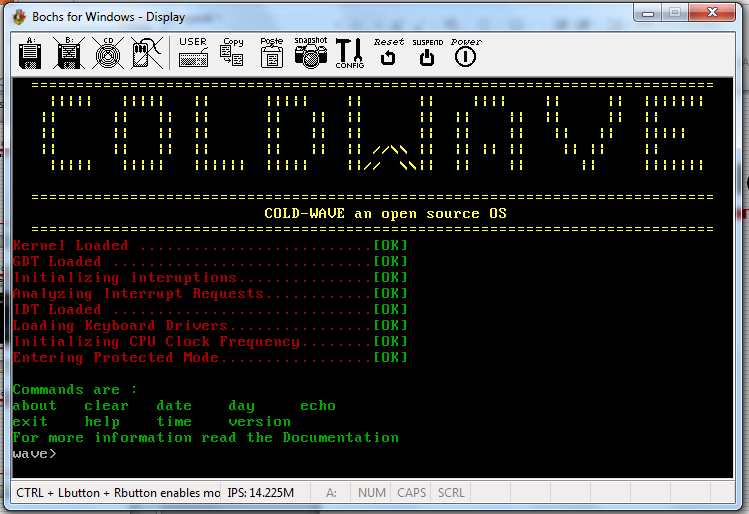
APPENDIX

Snapshots:

1. Starting Bochs emulator on Windows Platform



1. Booting COLDWAVE OS from bochs
2. COLDWAVE OS Home Screen



1. Executing commands from CLI of COLDWAVE OS

