Contents:

1. Protected Mode
2. Kernel: Basic Concepts -Part1
3. Kernel: Basic Concepts –Part2
4. Preparation of Kernel

Protected Mode:

As we know, protected mode is supposed to offer memory protection. By defining how the memory is used, we can insure certain memory locations cannot be modified, or executed as code. The 80x86 processor maps the memory regions based off the **Global Descriptor Table (GDT). The processor will generate a General Protection Fault (GPF) exception if you do not follow the GDT. Because we have not set up interrupt handlers, this will result in a triple fault.**

Descriptor Tables

A Descriptor Table defines or maps something--in our case, memory, and how the memory is used.

There are three types of descriptor tables: **Global Descriptor Table (GDT), Local Descriptor Table (LDT), and Interrupt Descriptor Table (IDT); each base address is stored in the GDTR, LDTR, and IDTR x86 processor registers.** Because they use special registers, they require special instructions.

Note: Some of these instructions are specific to Ring 0 kernel level programs. If a general Ring 3 program attempts to use them, a General Protection Fault (GPF) exception will occur.

Global Descriptor Table

The Global Descriptor Table (GDT) defines the global memory map. It defines what memory can be executed (The **Code Descriptor**), and what area contains data (**Data Descriptor**).

Remember that a descriptor defines properties--i.e., it describes something. In the case of the GDT, it describes starting and base addresses, segment limits, and even virtual memory.

The GDT usually has three descriptors--a **Null descriptor** (Contains all zeros), a **Code Descriptor**, and a **Data Descriptor**

Local Descriptor Table

The Local Descriptor Table (LDT) is a smaller form of the GDT defined for specialized uses. It does not define the entire memory map of the system, but instead, only up to 8,191 memory segments. We will go into this more later, as it does not have to do with protected mode. Cool?

Interrupt Descriptor Table

THIS will be important. Not yet, though. The Interrupt Descriptor Table (IDT) defines the Interrupt Vector Table (IVT). It always resides from address 0x0 to 0x3ff. The first 32 vectors are reserved for hardware exceptions generated by the processor.

For example, a **General Protection Fault** ,or a **Double Fault Exception**. This allows us to trap processor errors without triple faulting. More on this later, though.

The other interrupt vectors are mapped through a **Programmable Interrupt Controller** chip on the motherboard. We will need to program this chip directly while in protected mode.

PMode Memory Addressing

Remember that **PMode** (Protected Mode) uses a different addressing scheme then real mode. Real mode uses the **Segment:Offset** memory model, However PMode uses the **Descriptor:Offset** model.

This means, in order to access memory in PMode, we have to go through the correct descriptor in the GDT. The descriptor is stored in CS. This allows us to indirectly refrence memory within the current descriptor.

For example, if we need to read from a memory location, we do not need to describe what descriptor to use; it will use the one currently in CS. So, this will work:

```
mov bx, byte [0x1000]
```

This is great, but sometimes we need to reference a specific descriptor. For example, Real Mode does not use a GDT, While PMode requires it. Because of this, when entering protected mode, **We need to select what descriptor to use** to continue execution in protected mode. After all, because Real Mode does not know what a GDT is, there is no guarantee that CS will contain the correct descriptor in CS, so we need to set it.

Entering Protected Mode

To enter protected mode is fairly simple. At the same time, it can be a complete pain. To enter protected mode, we have to load a new GDT which describes permission levels when accessing memory. We then need to actually switch the processor into protected mode, and jump into the 32 bit world.

Step 1: Load the Global Descriptor Table

Remember that the GDT describes how we can access memory. If we do not set a GDT, the default GDT will be used (Which is set by the BIOS--Not the ROM BIOS). As you can imagine, this is by no means standard among BIOS's. And, **if we do not watch the limitations of the GDT (ie. if we access the code selector as data), the processor will generate a General Protection Fault (GPF). Because no interrupt handler is set, the processor will also generate a second fault exception--which will lead to a triple fault.**

Step 2: Entering Protected Mode

- **Bit 0 (PE) : Puts the system into protected mode**
- **Bit 1 (MP): Monitor Coprocessor Flag** This controls the operation of the **WAIT** instruction.
- **Bit 2 (EM): Emulate Flag**. When set, **coprocessor instructions will generate an exception**
- **Bit 3 (TS): Task Switched Flag** This will be set when the processor switches to another **task**.
- **Bit 4 (ET): Extension Type Flag. This tells us what type of coprocessor is installed.**
    - 0 - 80287 is installed
    - 1 - 80387 is installed.
- **Bit 5:** Unused.
- **Bit 6 (PG): Enables Memory Paging.**

The important bit is bit 0. **By setting bit 0, the processor continues execution in a 32 bit state.** That is, **setting bit 0 enables protected mode.**

Kernel: Basic Concepts-Part 1

Kernel: Basic Definition

Dictionaries define "kernel" as "core", "essential part", or even "The body of something". When applying this definition to an Operating System environment, we can easily state that:

The Kernel is the core component of an operating system.

There is no rule that states a kernel is mandatory. We can easily just load and execute programs at specific addresses without any "kernel". In fact, all of the early computer systems started this way. Some modern systems also use this. A notable example of this are the early console video game systems, which required **rebooting** the system in order to execute **one** of the games designed for that console.

So, what is the point of a Kernel? In a computing environment, it is impractical to restart every time to execute a program. This will mean that each program itself would need its own bootloaders and direct hardware controlling. After all, if the programs need to be executed at boot up, there would be no such thing as an operating system.

What we need is an abstraction layer to provide the capability of executing multiple programs, and manage their memory allocations. It also can provide an abstraction to the hardware, which will not be possible if each program had to start on boot up without an OS. After all, the software will be running on raw hardware.

The need for Kernels

The Kernel provides the primary abstraction layer to the hardware itself.

It has direct control over every little thing.

It is impossible to have multitasking, or multiprocessing as there is no common ground to switch between programs and processes. Only one program can execute at a time.

The basic idea is that a Kernel is a necessity. Not only do we want to **prevent** other software direct control over everything, but we want to create an **abstraction layer** for it.

Abstraction Layers of Software

Software has a lot of abstractions. All of these abstractions is meant to provide a core and basic interfaces to not only hide implementation detail, but to **shield** you from it. Having direct control over everything might seem cool--but imagine how much problems would be caused by doing this.

The Kernel

Because the Kernel is the Core component, it needs to provide the management for everything that relies on it. **The primary purpose of the Kernel is to manage system resources, and provide an interface so other programs can access these resources.** In a lot of cases, the Kernel itself is unable to use the interface it provides to other resources. **It has been stated that the Kernel is the most complex and difficult tasks in programming.**

This implies that designing and implementing a good Kernel is very difficult.

Memory Management

This is quite possibly the most important part of any Kernel. And rightfully so--all programs and data require it. As you know, in the Kernel, because we are still in **Supervisor Mode** (Ring 0), **We have direct access to every byte in memory**. This is very powerful, but also produces problems, especially in a multitasking environment, where multiple programs and data require memory.

One of the primary problems we have to solve is: What do we do when we run out of memory?

Another problem is **fragmentation**. **It is not always possible to load a file or program into a sequential area of memory**.

Virtual Address Space (VAS)

A **Virtual Address Space** is a **Program's Address Space**. One needs to take note that this does **not** have to do with **System Memory**. The idea is **so that each program has their own independent address space. This insures one program cannot access another program, because they are using a different address space.**

Because **VAS** is **Virtual** and not directly used with the physical memory, it allows the use of other sources, such as disk drives, as if it was memory. That is, **It allows us to use more memory then what is physically installed in the system.**

Virtual Addresses are mapped by the Kernel trough the MMU.

Virtual Memory: Abstract

Virtual Memory is a special **Memory Addressing Scheme** implemented by both the hardware and software. It allows noncontiguous memory to act as if it was contiguous memory.

Virtual Memory is based off the **Virtual Address Space** concepts. It provides every program its own Virtual Address Space, allowing memory protection, and decreasing memory fragmentation.

Virtual Memory also provides a way to indirectly use more memory then we actually have within the system. One common way of approaching this is by using **Page files**, stored on a **hard drive**.

Memory Management Unit (MMU): Abstract

The MMU, Also known as **Paged Memory Management Unit (PMMU)** is a component inside the microprocessor responsible for the management of the memory requested by the CPU. It has a number of responsibilities, including **Translating Virtual Addresses to Physical Addresses, Memory Protection, Cache Control, and more.**

Segmentation: Abstract

Segmentation is a method of **Memory Protection**. In Segmentation, we only allocate a certain address space from the currently running program. This is done through the **hardware registers**.

Segmentation is one of the most widely used memory protection scheme. On the x86, it is usually handled by the **segment registers**: CS, SS, DS, and ES.

We have seen the use of this through Real Mode.

Paging: Abstract

THIS will be important to us. Paging is the process of managing program access to the virtual memory pages that are not in RAM.

Supervisor Mode

Ring 0 is known as **supervisor mode**. It has access to every instruction, register, table, and other, more privileged resources that no other applications with higher ring levels can access.

Ring 0 is also known as **Kernel level**, and is **expected** never to fail. If a ring 0 program crashes, it will take the system down with it.

Supervisor Mode utilizes a hardware flag that can be changed by system level software. System level software (Ring 0) will have this flag set, while application level software (Ring 3) will not.

Only Ring 0 programs have direct access to hardware via software ports. Because of this, we will need to switch back to Ring 0 often.

Kernel Space

Kernel Space refers to a special region of memory that is reserved for the Kernel, and Ring 0 device drivers. In most cases, **Kernel Space should never be swapped out to disk, like virtual memory**.

If operating software runs in **User Space**, it is often known as **"Userland"**.

User Space

This is normally the **Ring 3 application programs**. Each application usually executes in its own **Virtual Address Space (VAS)** and can be swapped from different disk devices. **Because each application is within their own virtual memory, they are unable to access another programs memory directly.** Because of this, they will be required to go through a Ring 0 program to do this. This is necessary for **Debuggers**.

Applications are normally the least privileged. Because of this, they usually need to request support from a ring 0 Kernel level software to access system resources.

Switching Protection Levels

What we need is a way so that these applications can query the system for these resources. However, to do this, we need to be in Ring 0, Not Ring 3. Because of this, we need a way to switch the processor state from Ring 3 to Ring 0, and allow applications to query our system.

So...In order for an application to execute a system routine (while switching to Ring 0), the application must either **far jump**, execute an **Interrupt**, or use a special instruction, such as **SYSENTER**.

Interrupts

A **Software Interrupt** is a special type of interrupt implemented in software. Interrupts are used quite often, and rely on the use of a special table--the **Interrupt Descriptor Table (IDT)**. We will look at Interrupts a lot more closer later, as it is the first thing we will implement in our Kernel.

Interrupts are the most portable way to implement system calls. Because of this, we will be using interrupts as the first way of invoking a system routine.

Error Handling

Normally this is done by means of **Exception Handling**. Whenever the processor enters an invalid state caused by an invalid instruction, divide by 0, etc; the processor triggers an **Interrupt Service Routine (ISR)**. If you have mapped our own ISR's, it will call our routines.

The **ISR** called depends on what the problem was. This is great, as we know what the problem is, and can try finding the program that originally caused the problem.

Kernel: Basic Concepts-Part 2

Hardware Abstraction

Hardware Abstraction is very important. A HAL is a software abstraction layer used to provide an interface to the physical hardware. It is an abstraction layer. These abstractions provide a way to interact with devices, while not needing to know the details of a particular device or controller.

Normally in modern OSs, the HAL is a basic **Motherboard chipset driver**. It provides a basic interface between the kernel and the hardware of the machine, including the processor. This is great, as the Kernel can interact with the HAL whenever it needs access to the hardware. This also means that the kernel can be completely hardware independent.

Kernel: Putting everything together

Memory Management

Memory Management refers to:

- Dynamically giving and using memory to and from programs that request it.
- Implementing a form of Paging, or even Virtual Memory.
- Insuring the OS Kernel does not read or write to unkown or invalid memory.
- Watching and handling Memory Fragmentation.

Memory Protection refers to:

- Accessing an invalid descriptor in protected mode (Or an invalid segment address)
- Overwriting the program itself.
- Overwriting a part or parts of another file in memory.

Processor Management

As we know, the BIOS ROM initializes and starts up the primary processor. It only starts a single core. If you are running your OS on a system with a multicore processor, or a system with multiple processors, you will need to start up the other processors and cores manually.

Letting applications play with the different processors at any time can cause fatal system problems. Because of this, we should never allow applications the ability to do this.

I/O Device Management

Similar to physical memory, allowing applications direct access to controller ports and registers can cause the controller to malfunction, or system to crash. With this, depending on the complexity of the device, some devices can get surprisingly complex to program, and uses several different controllers. Because of this, providing a more abstract interface to

manage the device is important. This interface is normally done by a **Device Driver** or **Hardware Abstraction Layer**.

Frequently, applications will require access to these devices. The Kernel must maintain the list of these devices by querying the system for them in some way. This can be done through the BIOS, or through one of the various system buses (Such as PCI/PCIE, or USB.) When an application requests an operation on a device (Such as, displaying a character), the kernel needs to send this request to the current active video driver. The video driver, in turn, needs to carry out this request. This is an example of **Inter Process Communication (IPC).**

Process Management

Program Management is responsible for:

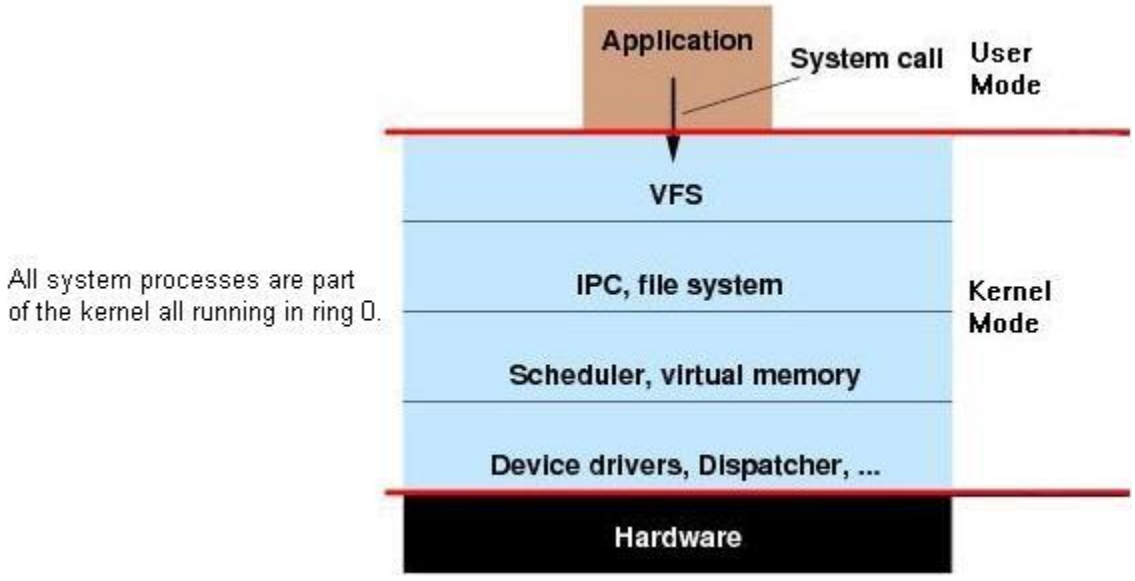
- Insuring the program doesn't write over another program.
- Insuring the program does not corrupt system data.
- Handle requests from the program to complete a task (such as allocate or deallocate memory).

Multitasking refers to:

- Switching and giving multiple programs a certain timeframe to execute.
- Providing a Task Manager to allow switching (Such as Windows Task Manager).
- TSS (Task State Segment) switching. Another new term!
- Executing multiple programs simultaneously.

Kernel Designs - Abstract: Primary Design Models

Monolithic kernel Design



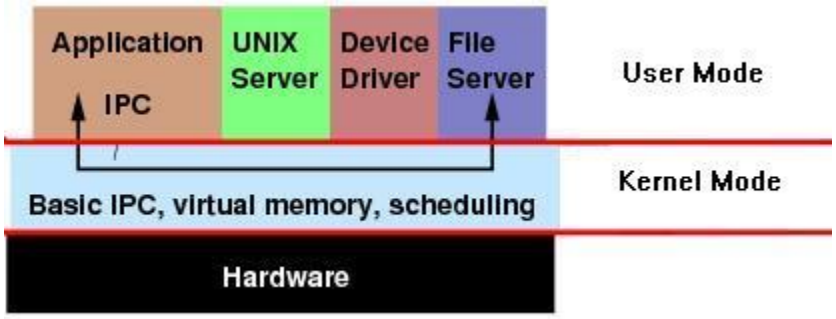
In a Monolithic Kernel, all system process run as part of the Kernel at Ring 0

The first part--"Mono" means "one". The second part--"lithic" means "it is of or like stone".

In a Monolithic kernel, the entire Kernel executes in Kernel space at Ring 0. It provides a higher level interface over computer resources and hardware. It also provides a basic set of system calls via the System API.

In a monolithic kernel, most (if not all) Kernel services are part of the kernel, itself.

Microkernel Design



In a Microkernel design, the Kernel only provides basic functionality needed for user mode system services.

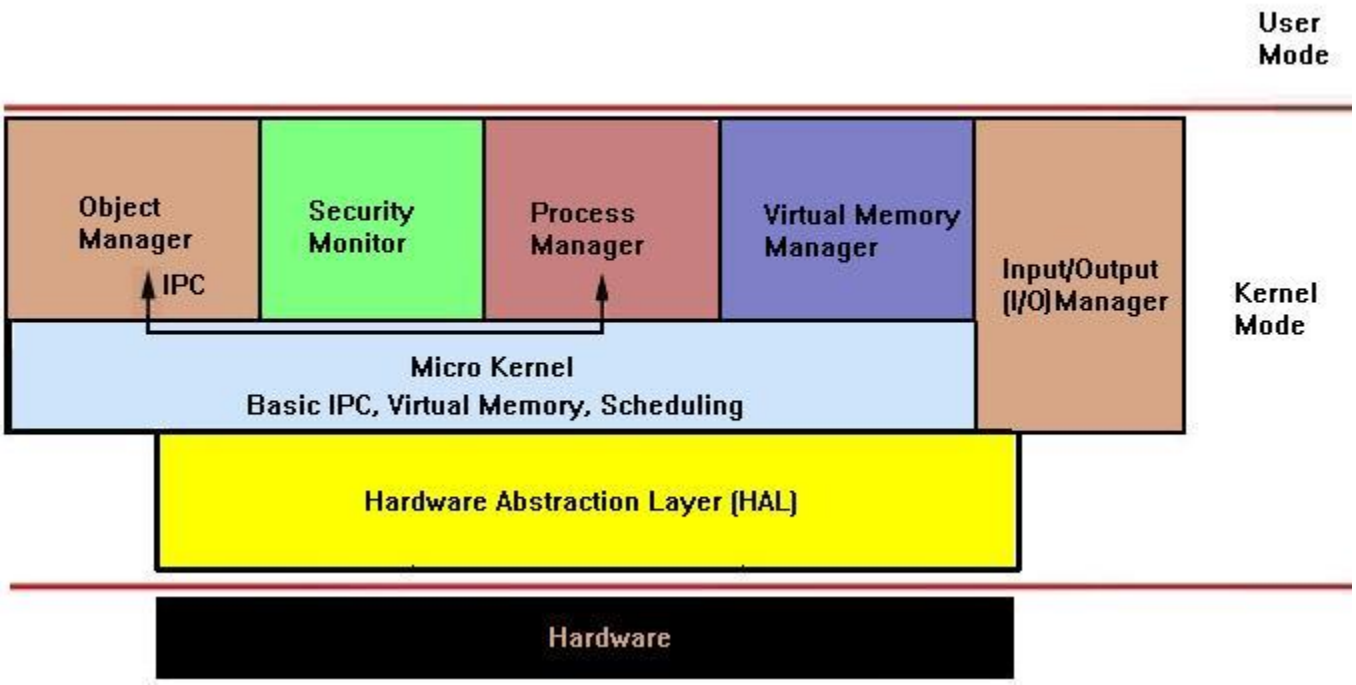
A Microkernel is a kernel design that provides no OS services at all, only the mechanisms needed to implement those services. Because of this, the Kernel itself is usually quite small compared to Monolithic kernels. For example, a microkernel may implement low level memory management and thread management, and Inter Process Communication (IPC).

The Kernel would use external user mode services, such as Device Drivers and Filesystems, rather then everything implemented as part of the kernel (As with Monolithic kernels.)

Kernel Designs - Abstract: Secondary Design Models

Remember that there are countless of ways that kernels may be designed. The following are common design models that are based off of the primary design models (Monolithic and Microkernels).

Hybrid kernels



Hybrid kernels are Microkernels with aspects from Monolithic kernels

A Hybrid kernel is a kernel combining aspects from both Monolithic and Microkernel designs.

Hybrid kernels usually has a structure similar to microkernels, but implemented as a monolithic kernel. Lets look at this another way for better understanding.

Hybrid Kernels, similar to microkernels, use separate sever programs for filesystems, device drivers, etc. However, like Monolithic Kernels, these severs execute as part of the Kernel, instead of user space.

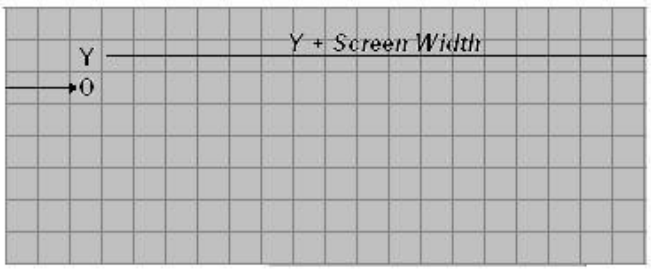
Nano kernel

Nano kernels, also known as **Pico kernels**, are a very small kernel. Normally, this would be a minimal microkernel structure. As the kernel itself is very small, it must rely on other software and drivers for the basic resources within the system.

Preparation for the Kernel

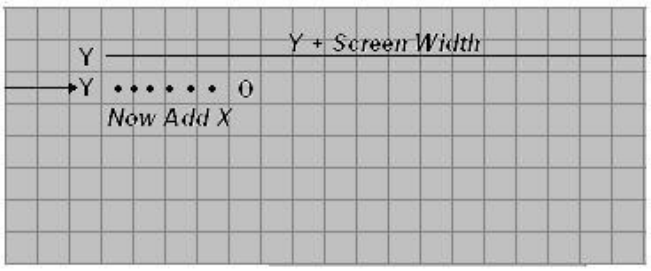
Printing characters

A special property about memory is how it is linear. If we reach the end of a line being displayed, the next byte is on the line right below it. Because of linear addressing, we have to be able to convert an x/y location to a linear address to render it to screen. And, a special formula to do that is: **x + y * screen width**.



Notice by multiplying screen width by a value of Y, we effectivly move down one row on the screen within video memory.

Because of this, multiplying it by a value of Y means that, by incrementing the value of Y, we effectivly go down Y rows.



Now, we add the value of "X" to this location, giving us a forumla: ***Location = x + (y * screen width)***

Remember that memory is linear!

.

The attribute byte provides a way of supplying color, as well as certain attributes, such as blinking. The values can be...

- **0** - Black
- **1** - Blue
- **2** - Green
- **3** - Cyan
- **4** - Red
- **5** - Magenta
- **6** - Brown
- **7** - Light Gray
- **8** - Dark Gray
- **9** - Light Blue
- **10** - Light Green
- **11** - Light Cyan
- **12** - Light Red
- **13** - Light Magenta
- **14** - Light Brown
- **15** – White

Setting up

Printing characters is a little complex because we have to track where we are, both in current x/y location and when writing to video memory. We also need to track certain characters, such as the newline character, and to watch for the end of line. And yet, we still need to update the hardware cursor to this position as well.

Working with strings

To print actual information, we will need a way to print full strings. Because we already have a routine that tracks current position (and updates it), and prints the characters, all we need to do to print a string is a simple loop.

```
Puts32:

        ;--------------------------------;
        ;   Store registers              ;
        ;--------------------------------;

        pusha                           ; save registers
        push    ebx                     ; copy the string address
        pop     edi
```

Okay, Here's our Puts32() function. It takes one parameter: EBX, which contains the address of a null terminated string to print. Because out Putch32() function requires that BL store the character to print, we need to save a copy of EBX, so we do it here.

Updating the hardware cursor

So we can print characters and strings out now. You might notice something though: the cursor does not move! Because of this, it just stays no matter what we do. This cursor is a simple underline that the BIOS uses to indicate the current position when printing text.

Clearing the screen

Because we already have a way to display text, just loop, and reset the current position to 0!