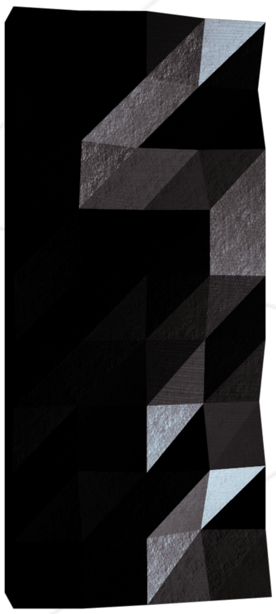

KFS_2

GDT & Stack

Louis Solofrizzo louis@ne02ptzero.me
42 Staff pedago@42.fr

Summary: Let's code the stack !

Contents

Chapter I

Forewords

I.0.1 Hunter 6.2.4 Rotation

- Barrage
- Aspect of the pack
- Die
- Blame the heal

I.0.2 Rogue 6.2.4 Rotation

- Pull without the tank
- Die
- Blame the heal

I.0.3 Warrior tank 6.2.4 Rotation

- Pull the entire dungeon
- Die
- Blame the heal

I.0.4 Druid Heal 6.2.4 Rotation

- Blame the other heal

I.0.5 Death Knight 6.2.4 Rotation

- Roll
- Your
- Head
- On
- The
- Keyboard

Chapter II

Introduction

Welcome in `Kernel from Scratch`, second subject. This time, you will code a stack, and integrate it with the `GDT`.

What is a `GDT` ? Let's see:

The `GDT` ("Global Descriptor Table") is a data structure used to define the different memory areas: the base address, the size and access privileges like execute and write. These memory areas are called "segments".

In a `GDT`, you can find:

- Kernel code, used to store the executable binary code
- Kernel data
- Kernel stack, used to store the call stack during kernel execution
- User code, used to store the executable binary code for user programs
- User program data
- User stack, used to store the call stack during execution in userland

I think you can see why this thing is really important in a Kernel !

Next, the stack. I'm sure you all know what a stack is, but here's a friendly reminder:

> *In computer science, a stack is an abstract data type that serves as a collection of elements, with two principal operations: push, which adds an element to the collection, and pop, which removes the most recently added element that was not yet removed. The order in which elements come off a stack gives rise to its alternative name, LIFO (for last in, first out).*

Get it ? Good. Now, let's move on.

Chapter III

Goals

In this subject, you will have to `create`, `fill` and `link` a Global Descriptor Table into your Kernel.

Yup, that's it. Not so much, eh ?

Actually, you will have to understand how "memory" really works in a system, how the Stack and RAM works, how to use it, how to fill it and how to link it with the `BIOS`.

Yeah, the `BIOS`. Thanks to `GRUB`, it will help you a lot !
Good thing you already installed it.

Chapter IV

General instructions

IV.1 Code and Execution

IV.1.1 Emulation

The following part is not mandatory, you're free to use any virtual manager you want to, however, i suggest you to use `KVM`. It's a `Kernel Virtual Manager`, and have advanced execution and debugs functions. All the example below will use `KVM`.

IV.1.2 Language

The `C` language is not mandatory, you can use any language you want for this suit of projects.
Keep in mind that all language are not kernel friendly, you could code a kernel with `Javascript`, but are you sure it's a good idea ?
Also, a lot of the documentation are in `C`, you will have to 'translate' the code all along if you choose a different language.

Furthermore, all of the features of a language cannot be used in a basic kernel. Let's take an example with `C++` :
This language uses 'new' to make allocation, class and structures declaration. But in your kernel, you don't have a memory interface (yet), so you can't use those features now.

A lot of language can be used instead of `C`, like `C++`, `Rust`, `Go`, etc. You can even code your entire kernel in `ASM` !

IV.2 Compilation

IV.2.1 Compilers

You can choose any compilers you want. I personnaly use `gcc` and `nasm`. A Makefile must be added to your repo.

IV.2.2 Flags

In order to boot your kernel without any dependencies, you must compile your code with the following flags (Adapt the flags for your language, those ones are for `C++`, for instance):

- `-fno-builtin`
- `-fno-exception`
- `-fno-stack-protector`
- `-fno-rtti`
- `-nostdlib`
- `-nodefaultlibs`

Pay attention to `-nodefaultlibs` and `-nostdlib`. Your Kernel will be compiled on a host system, yes, but cannot be linked to any existing library on that host, otherwise it will not be executed.

IV.3 Linking

You cannot use an existing linker in order to link your kernel. As written above, your kernel will not boot. So, you must create a linker for your kernel.
Be carefull, you `CAN` use the 'ld' binary available on your host, but it is **FORBIDDEN** to use the .ld file of your host.

IV.4 Architecture

The `i386` (x86) architecture is mandatory (you can thank me later).

IV.5 Documentation

There is a lot of documentation available, good and bad. I personnaly think the `OSDev` wiki is one of the best.

IV.6 Base code

In this subject, you have to take your precedent `KFS` code, and work from it !
Or not... and rewrite all from the beginning. Your call !

Chapter V

Mandatory part

Let's sum this up:

- You must create a `Global Descriptor Table`.
- Your `GDT` must contain:
    - Kernel Code
    - Kernel Data
    - Kernel stack
    - User code
    - User data
    - User stack
    - Your work should not exceed 10 MB.
- You must declare your `GDT` to the `BIOS`.
- The `GDT` must be set at address `0x00000800`.

When this is done, you have to code a tool to print the kernel stack, in a human-friendly way. (Tip: If you haven't made a `printk` yet, now is a good time !)

Chapter VI

Bonus part

Assuming your keyboard work correctly in your Kernel, and you able to catch an entry, let's code a Shell !
Not a POSIX Shell, just a minimalistic shell with a few commands, for debugging purposes.

For example, you could implement the print-kernel-stack-thing in this shell, and some other things like `reboot`, `halt` and such.

Have fun !

Chapter VII

Turn-in and peer-evaluation

Turn your work into your `GiT` repository, as usual. Only the work present on your repository will be graded in defense.

Your must turn in your code, a Makefile and a basic virtual image for your kernel. Side note about that image, your kernel does nothing with it yet, SO THERE IS NO NEED TO BE SIZED LIKE AN ELEPHANT.