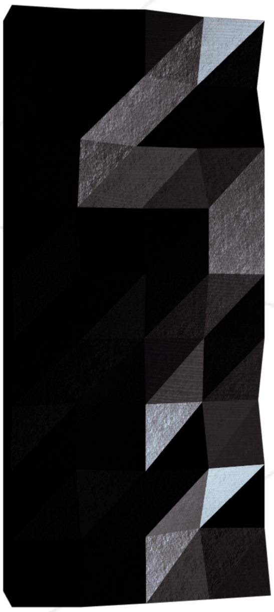


# KFS_3

## Memory

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

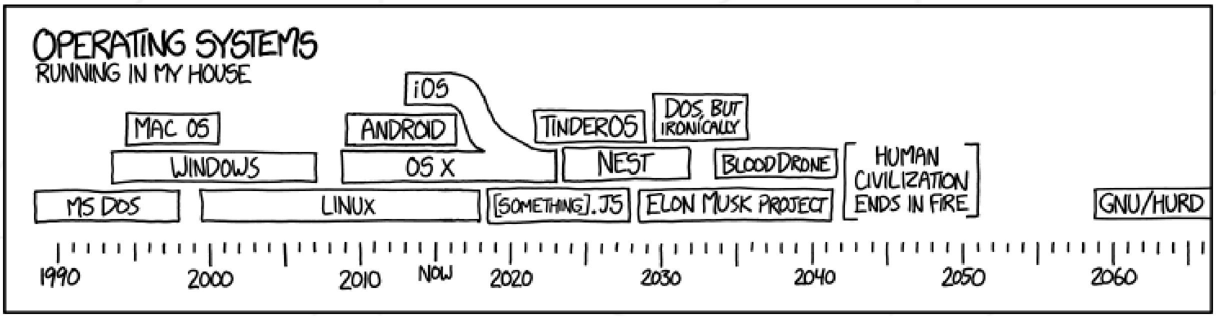
*Summary: The sweet world of memory*

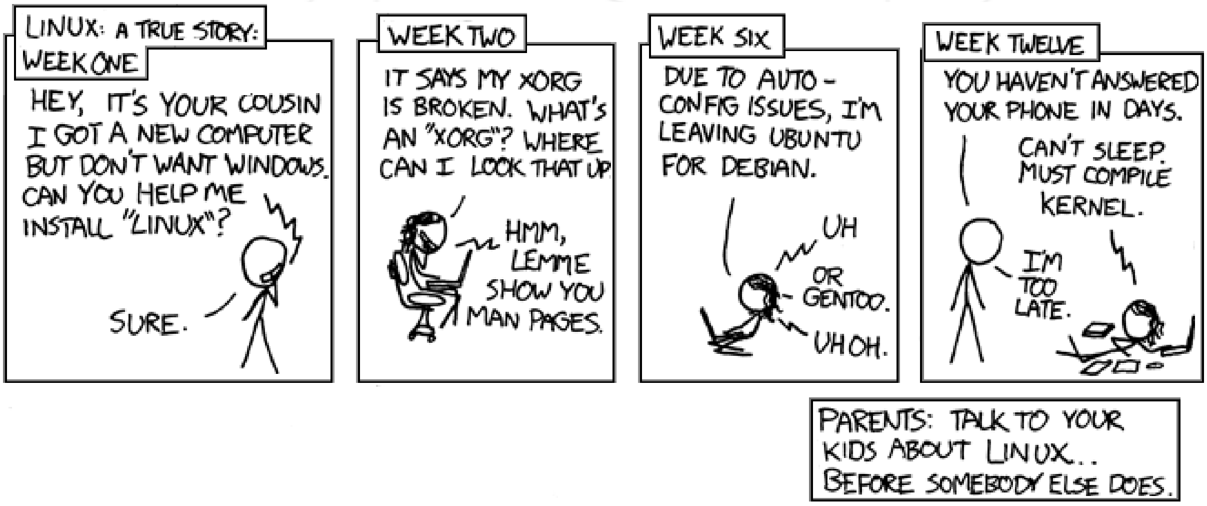
*Version: 1*

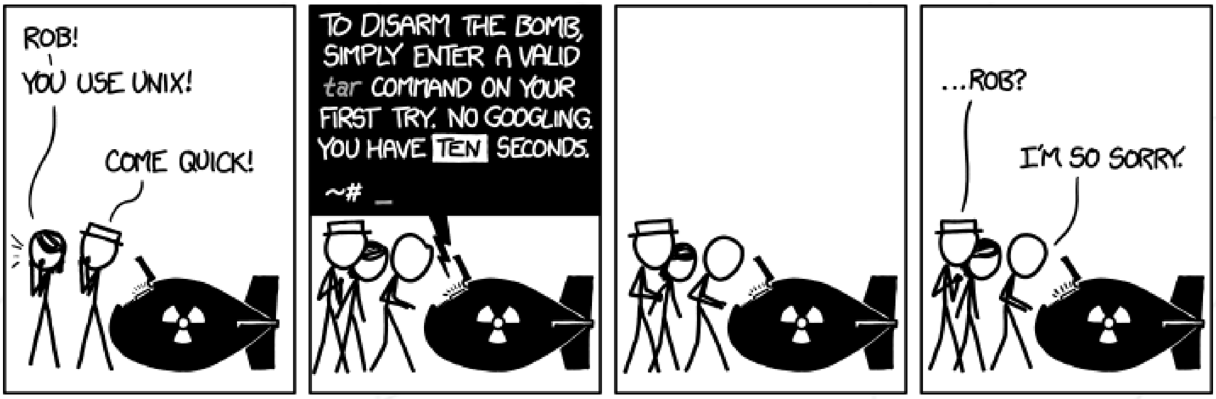
# Contents

# Chapter I

# Foreword




LINUX: A TRUE STORY:
WEEK ONE
HEY, IT'S YOUR COUSIN I GOT A NEW COMPUTER BUT DON'T WANT WINDOWS. CAN YOU HELP ME INSTALL "LINUX"?
SURE.
WEEK TWO
IT SAYS MY XORG IS BROKEN. WHAT'S AN "XORG"? WHERE CAN I LOOK THAT UP
HMM, LEMME SHOW YOU MAN PAGES.
WEEK SIX
DUE TO AUTO-CONFIG ISSUES, I'M LEAVING UBUNTU FOR DEBIAN.
UH
OR GENTOO.
UH OH.
WEEK TWELVE
YOU HAVEN'T ANSWERED YOUR PHONE IN DAYS.
CAN'T SLEEP. MUST COMPILE KERNEL.
I'M TOO LATE.
PARENTS: TALK TO YOUR KIDS ABOUT LINUX... BEFORE SOMEBODY ELSE DOES.


ROB!
YOU USE UNIX!
COME QUICK!
TO DISARM THE BOMB, SIMPLY ENTER A VALID tar COMMAND ON YOUR FIRST TRY. NO GOOGLING. YOU HAVE TEN SECONDS.
~# _
...ROB?
I'M SO SORRY.

# Chapter II

# Introduction

## II.1 Foreword

Welcome to `Kernel from Scratch`, third subject. This time, we will make a proper memory !

In the last subject, you have declared your memory space to the `BIOS`, and now it's the time to use it. In other words, you will code `memory paging`, `allocation`, and `free`.

First, let's take a look of what a dynamic memory allocation is:

> *The task of fulfilling an allocation request consists of locating a block of unused memory of sufficient size. Memory requests are satisfied by allocating portions from a large pool of memory called the heap or free store. At any given time, some parts of the heap are in use, while some are "free" (unused) and thus available for future allocations.*

I'm sure you all know how to use a `malloc`, and what's behind it. But this subject is more than a `malloc`, it's about `memory paging`, the difference between `physical` and `virtual` memory, and memory code structures.

This subject is one of the more important of the `Kernel from Scratch` series. Take your time, write proper code, cause if you don't, you will regret it !

## II.2 Theory

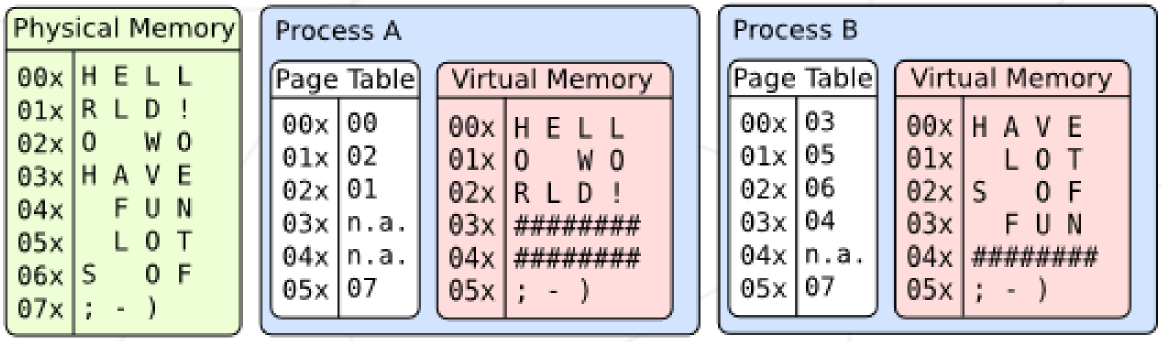
Note: This section is from the Samy Pesse's book: How to write an operating system. A must read !

### II.2.1 Why do we need paging ?

Paging will allow your kernel to:

- Use the hard-drive as a memory and not be limited by the machine ram memory limit
- To have a unique memory space for each process
- To allow and unallow memory space in a dynamic way

In a paged system, each process may execute in its own 4gb area of memory, without any chance of effecting any other process's memory, or the kernel's.
It simplifies multitasking.



Physical Memory

| | |
|---|---|
| 00x | H E L L |
| 01x | R L D ! |
| 02x | O   W O |
| 03x | H A V E |
| 04x | F U N |
| 05x | L O T |
| 06x | S   O F |
| 07x | ; - ) |

Process A

Page Table

| | |
|---|---|
| 00x | 00 |
| 01x | 02 |
| 02x | 01 |
| 03x | n.a. |
| 04x | n.a. |
| 05x | 07 |

Virtual Memory

| | |
|---|---|
| 00x | H E L L |
| 01x | O   W O |
| 02x | R L D ! |
| 03x | ######## |
| 04x | ######## |
| 05x | ; - ) |

Process B

Page Table

| | |
|---|---|
| 00x | 03 |
| 01x | 05 |
| 02x | 06 |
| 03x | 04 |
| 04x | n.a. |
| 05x | 07 |

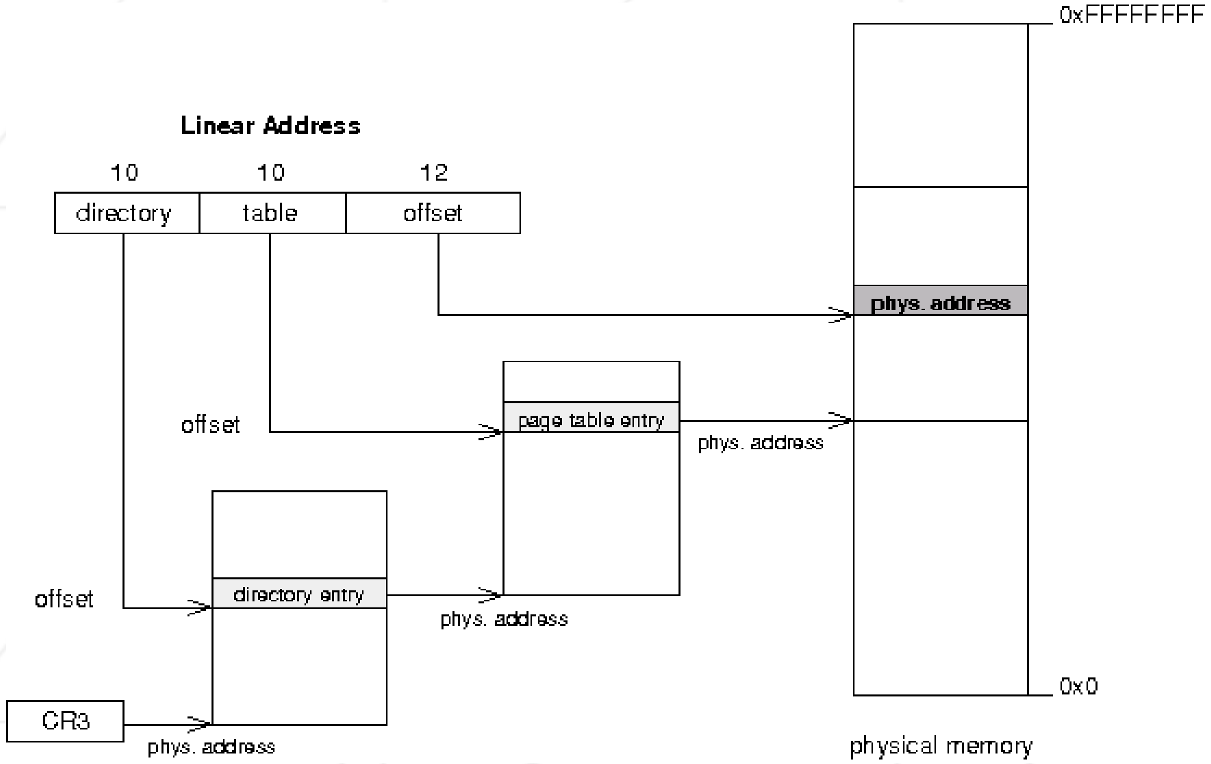
Virtual Memory

| | |
|---|---|
| 00x | H A V E |
| 01x | L O T |
| 02x | S   O F |
| 03x | F U N |
| 04x | ######## |
| 05x | ; - ) |

### II.2.2 How does it work ?

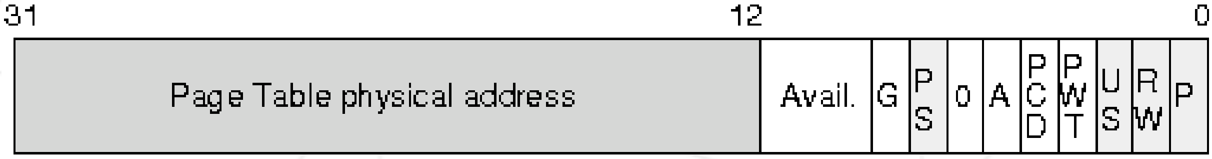
The translation of a linear address to a physical address is done in multiple steps:

- The processor use the registry CR3 to know the physical address of the pages directory.
- The first 10 bits of the linear address represent an offset (between 0 and 1023), pointing to an entry in the pages directory. This entry contains the physical address of a pages table.
- The next 10 bits of the linear address represent an offset, pointing to an entry in the pages table. This entry is pointing to a 4ko page.
- The last 12 bits of the linear address represent an offset (between 0 and 4095), which indicates the position in the 4ko page.


Linear Address
10
10
12
directory
table
offset
offset
offset
CR3
phys. address
directory entry
phys. address
page table entry
phys. address
phys. address
0xFFFFFFFF
0x0
physical memory

### II.2.3 Format for pages table and directory

The two types of entries (table and directory) look like the same. Only the field in gray will be used in the `kernel`.



- `P`: indicate if the page or table is in physical memory
- `R/W`: indicate if the page or table is accessible in writting (equals 1)
- `U/S`: equals 1 to allow access to non-preferred tasks
- `A`: indicate if the page or table was accessed
- `D`: (only for pages table) indicate if the page was written
- `PS`: (only for pages directory) indicate the size of pages :
    - 0 = 4kb
    - 1 = 4mb

Notes: Physical addresses in the pages directory or pages table are written using 20 bits because these addresses are aligned on 4kb, so the last 12bits should be equal to 0.

- A pages directory or pages table used 1024*4 = 4096 bytes = 4k
- A pages table can address 1024 * 4k = 4 Mb
- A pages directory can address 1024 (1024 4k) = 4 Gb

# Chapter III

# Goals

At the end of this subject, you will have:

- A complete memory code structure, with pagination handling
- Read and Write rights on memory
- User space memory and Kernel space memory
- Physical and Virtual memory
- Code helpers for physical memory, like `kmalloc`, `kfree`, `ksize`, `kbrk`
- Code helpers for virtual memory, like `vmalloc`, `vfree`, `vsize`, `vbrk`
- Kernel Panic handling

Lot of work on this one !

# 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 of 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 turn in to.

### 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 a `C++` example):

- `-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 you `CANNOT` 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 don't. And rewrite all from scratch. Your call !

# Chapter V

# Mandatory part

You must implement a `complete`, `stable` and `functionnal` memory system in your kernel.
Let's follow this task, point by point:

- You must enable memory paging in your `Kernel`
- You must code a memory structure that handle paging and memory rights (Careful, you don't have the tools yet to know who's accessing the memory, so all of this is theoric at the moment)
- You must define `Kernel` and `User space`
- You must implement a function to `create` / `get` memory pages
- You must implement functions to `allocate`, `free` and `get` size of a variable.
- You must implement those functions for `virtual` and `physical` memory
- You must handle "`kernel panics`" (Print, stop the kernel)
- Your work should not exceed 10 MB.

Some notes:
First, about this implementation. If you remember correctly, in the first subject, i was speaking about language (Other than `C`) limitation, and memory integration. Now's the time to implement it !
Secondly, about the panics. There are some times when the kernel must stop, and some times when the kernel can continue. So all panics are not fatal, make the difference.

# Chapter VI

# Bonus part

Since this subject is really, really hard, the bonuses are not really important.
Try to focus on the code itself, because the memory is most important part of your kernel, by far. But if you are looking for some things to do after that, try to implement `memory dumping` and `debug` in the last "mini-shell" subject. Keep in mind that will be not graded.

# 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.