



**TECHNISCHE  
UNIVERSITÄT  
DRESDEN**

**Department of Computer Science** Institute of System Architecture, Operating Systems Group

# **EXERCISE 1: GETTING STARTED**

**MICHAEL ROITZSCH**

- first contact with a microkernel OS
- getting to know QEMU
- compile Fiasco
- compile minimal system environment
- talk about system booting
- the usual „Hello World“
- review some stuff and play with the system

- developing your own kernel usually requires a dedicated machine
- we will use a virtual machine
- QEMU is open-source software providing a virtual machine by binary translation
- it emulates a complete x86 PC
- available for other architectures as well
- our QEMU will boot from an ISO image

# Setup

- download the source tarball from `http://os.inf.tu-dresden.de/Studium/KMB/WS2009/Exercise1.tar.bz2`
- unpack the tarball
  - it comes with a working directory
  - `cd` in there and have a look around
- initialize the environment with `make setup` in the toplevel directory you unpacked

# Test-Driving QEMU

- create a bootable ISO image
  - create an `iso` subdirectory for the ISO's content
  - run `isocreator` from `src/14/tool/bin` on this directory
- your ISO will contain a minimal grub installation
- launch QEMU with the resulting ISO:  
`qemu -cdrom boot.iso`

# Compiling the System

- run `make` within the toplevel directory

# BOOTING

- Basic Input Output System
- fixed entry point after „power on“ and „reset“
- initializes the CPU in 16-bit real-mode
- detects, checks and initializes some platform hardware (like RAM, PCI, ATA)
- finds the boot device

**BIOS**

- Extensible Firmware Interface
- no legacy PC-AT boot (no A20 gate)
- no text-mode boot any more
- extensible for new hardware
- built-in boot manager
- more than four partitions

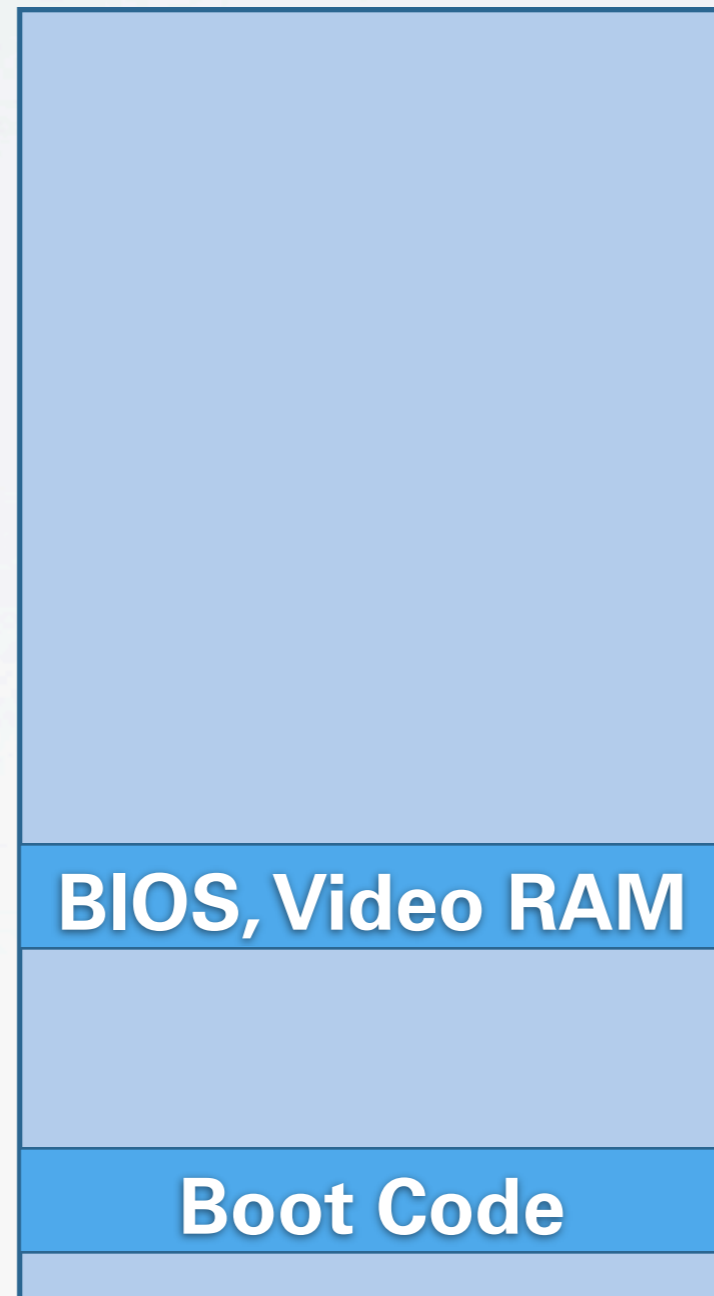


**BIOS**

- first sector on boot disk
- 512 bytes
- contains first boot loader stage and partition table
- BIOS loads code into RAM and executes it
- problem: How to find and boot an OS in 512 bytes?



**BIOS**



Physical Memory

**BIOS**

- popular boot loader
- used by most (all?) Linux distributions
- uses a two-stage-approach
  - first stage fits in one sector
  - has hard-wired sectors of second stage files
  - second stage can read common file systems

**Boot Loader**

**BIOS**

- second stage loads a menu.lst config file to present a boot menu
- from there, you can load your kernel
- supports loading multiple modules
- files can also be retrieved from network

**Boot Loader**

**BIOS**

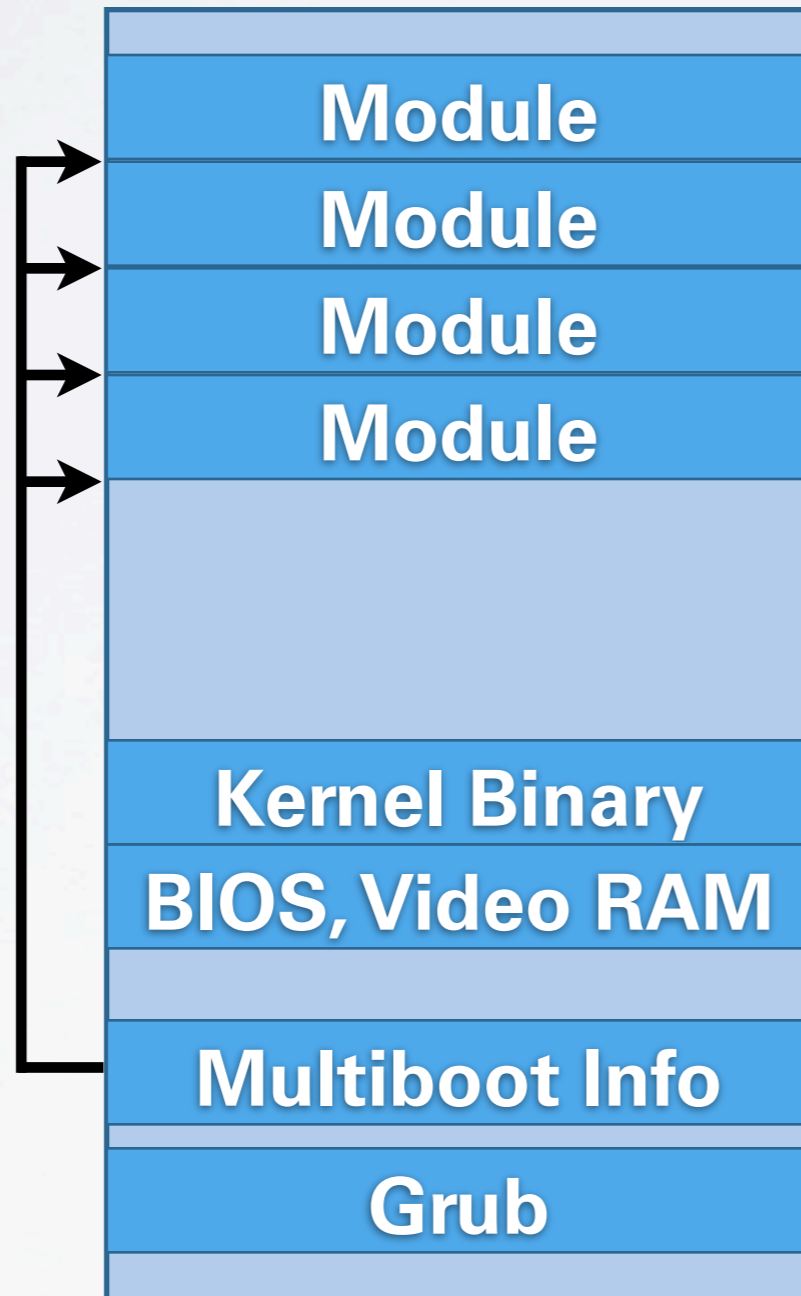
- switches CPU to 32-bit protected mode
- loads and interprets the „kernel“ binary
- loads additional modules into memory
- sets up multiboot info structure
- starts the kernel

**Boot Loader**

**BIOS**



# MEMORY LAYOUT



Physical Memory

**Boot Loader**

**BIOS**

- our modules are ELF files:  
executable and linkable  
format
- contain multiple sections
  - code, data, BSS
- bootstrap interprets the ELF  
modules
- copies sections to final  
location in physical memory

**Bootstrap**

**Boot Loader**

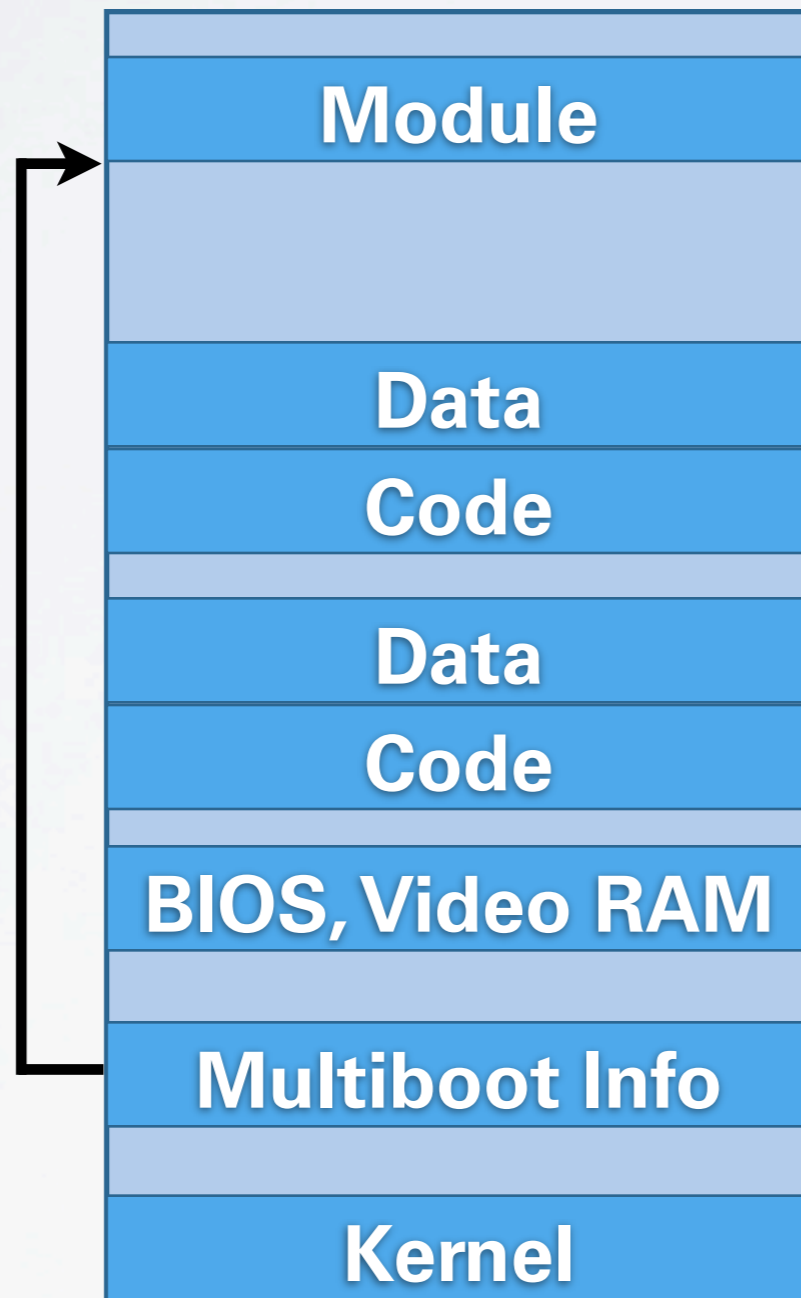
**BIOS**

- actual kernel is the first of the modules
- must know about the other modules
- bootstrap sets up a kernel info page
  - contains entry point and stack pointer of `sigma0` and `moe`
- passes control to the kernel

**Bootstrap**

**Boot Loader**

**BIOS**



Physical Memory



- initial kernel code
- basic CPU setup
  - detecting CPU features
  - setup various CPU-tables
- sets up basic page table
- enables virtual memory mode
- runs the actual kernel code

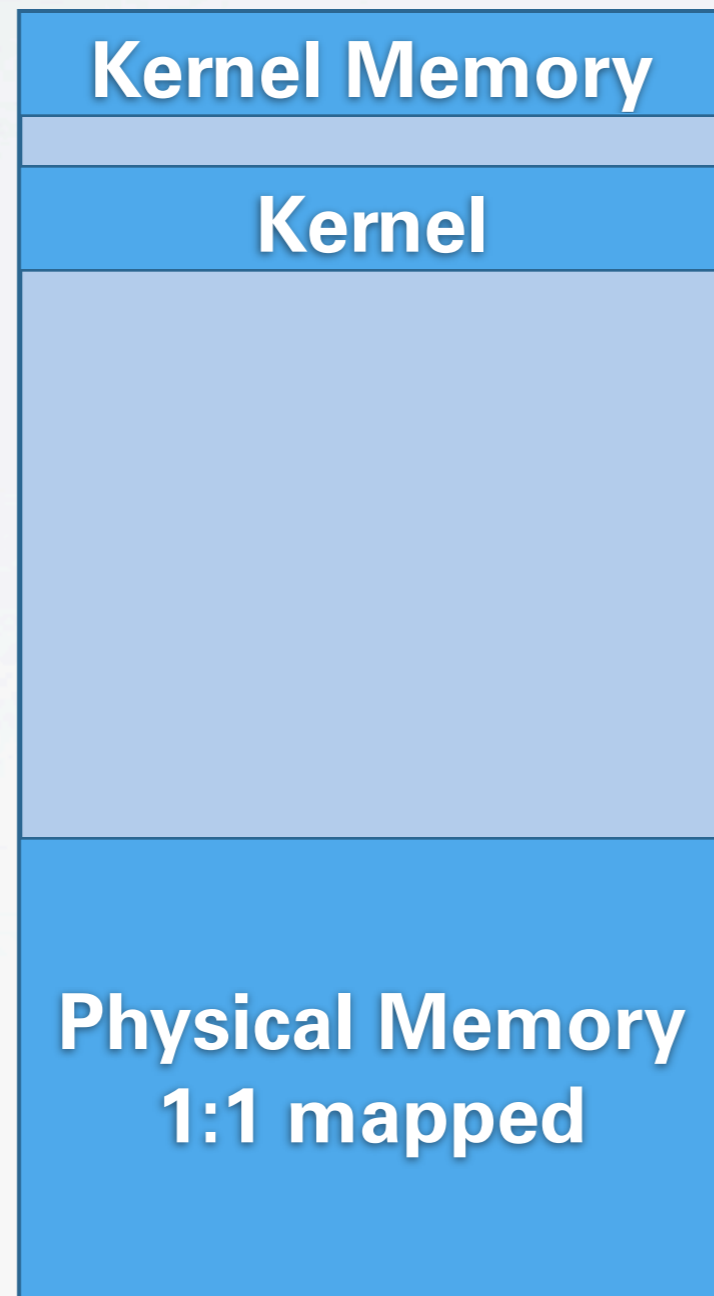


**Kernel Loader**

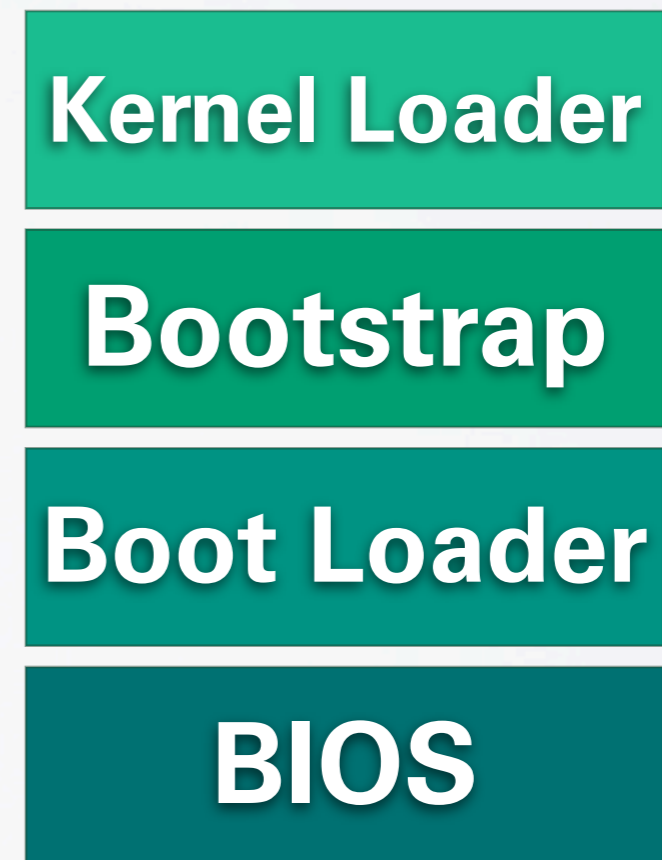
**Bootstrap**

**Boot Loader**

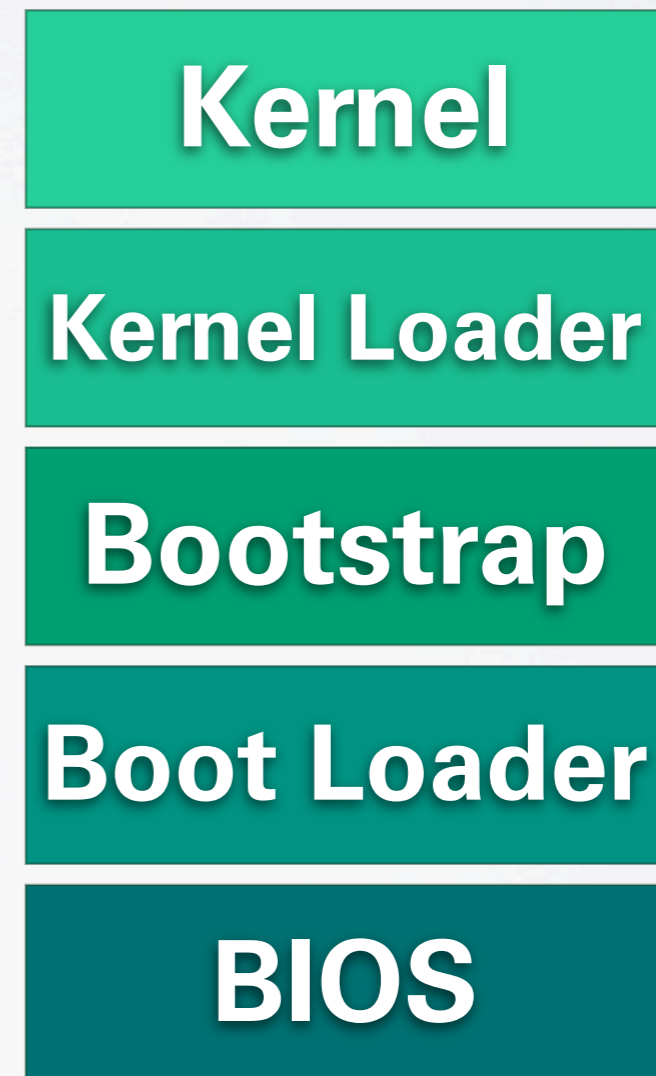
**BIOS**



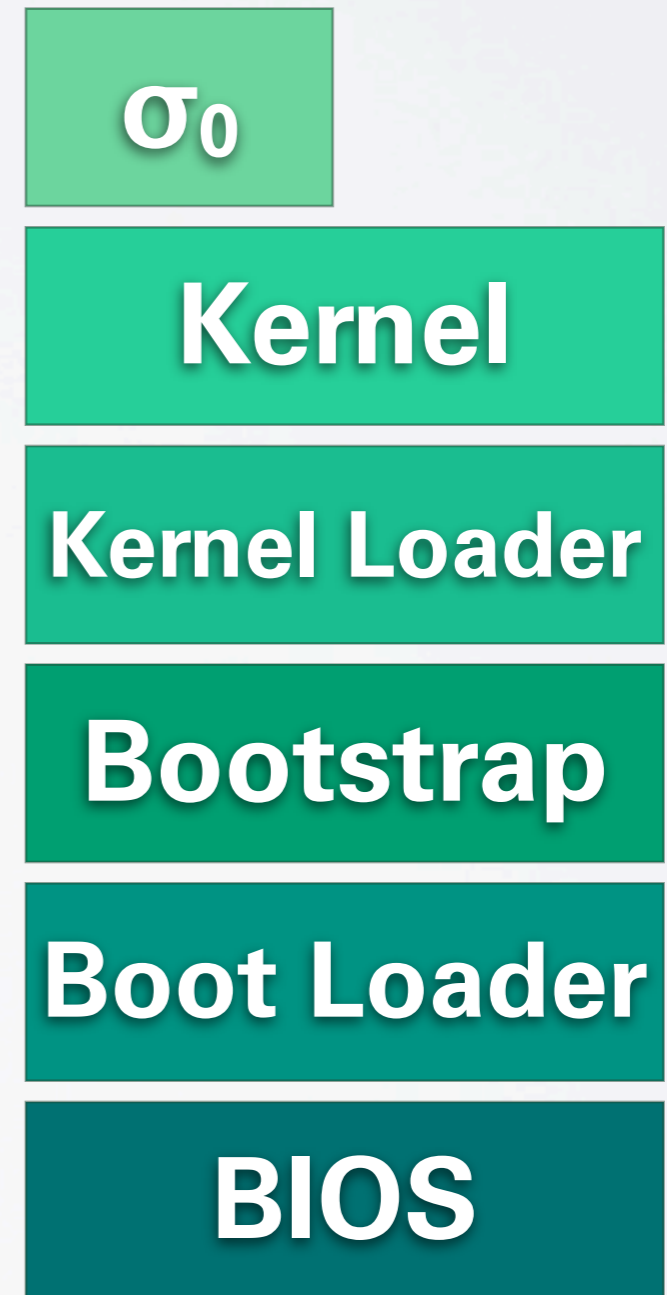
Virtual Memory



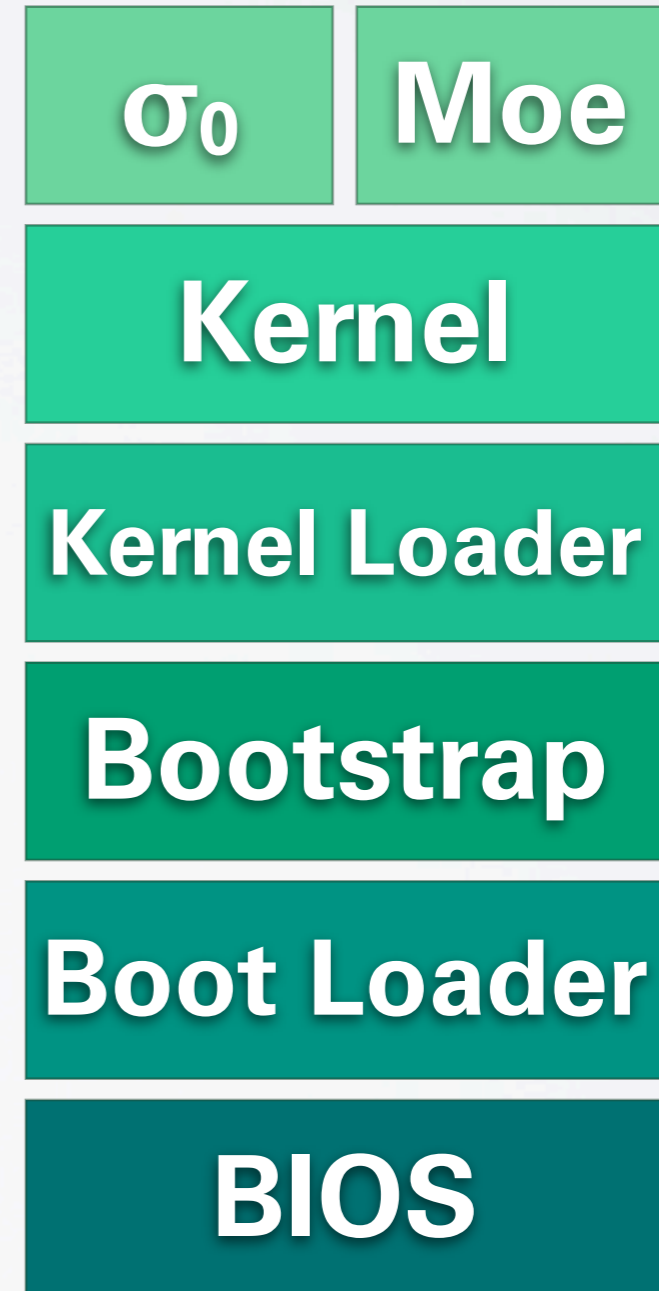
- sets up kernel structures
- sets up scheduling timer
- starts first pager
- starts first task
- starts scheduling
- scheduler hands control to userland for the first time



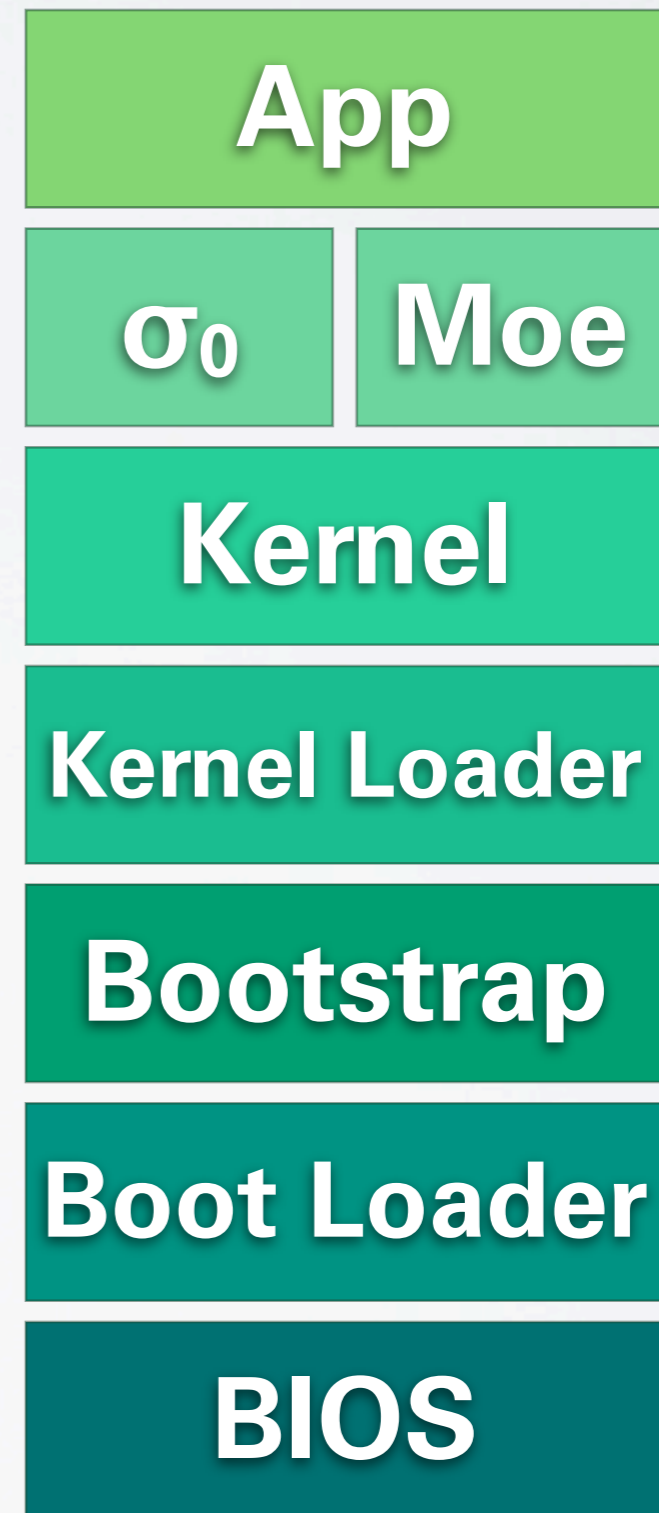
- is the first pager in the system
- initially receives a 1:1 mapping of physical memory
- sigma0 is the root of the pager hierarchy
- pager for moe



- manages initial resources
  - namespace
  - memory
- provides logging facility
- mini-filesystem for read-only access to boot-modules
- script-driven loader for further programs



- your killer-app is running now
  - moe injects a common service kernel into every task
- additional software can be loaded by
  - retrieving binaries via disk or network drivers
  - providing a loader service



# Booting Fiasco

- copy some files to the ISO directory
  - `fiasco` from the Fiasco build directory  
`obj/fiasco/ia32/`
  - `bootstrap` from  
`obj/14/x86/bin/x86_586/`
  - `sigma0, moe` and `14re` from  
`obj/14/x86/bin/x86_586/14f/`

# Booting Fiasco

- edit `iso/boot/grub/menu.lst`:  
title Getting Started  
kernel /bootstrap -modaddr 0x02000000  
module /fiasco  
module /sigma0  
module /moe  
module /l4re
- rebuild the ISO and run `qemu`

# Preparing for Hello

- provide `moe` with a loader script
  - pass `rom/hello.moe` as parameter to `moe`
  - add this line to `menu.lst`:  
`module /hello.moe`
- create the file `hello.moe` in the `iso` directory:  
`rom/hello`

# Exercise 1: Hello World

- create a directory for your hello-project
- create a Makefile with the following content:

```
PKGDIR          = .
L4DIR           = path to L4 source tree
OBJ_BASE       = absolute path to L4 build tree
TARGET         = hello
SRC_C           = hello.c
include $(L4DIR)/mk/prog.mk
```

- fill in `hello.c` and compile with `make`
- run in `qemu`

# Exercise 2: Ackermann Function

- write a program that spawns six threads
  - you can use pthreads in our system
  - add the line  
`L4_MULTITHREADED = y`  
to your `Makefile`
- each thread should calculate one value  $a(3,0..5)$  of the Ackermann function:
  - $a(0,m) = m+1$
  - $a(n,0) = a(n-1,1)$
  - $a(n,m) = a(n-1,a(n,m-1))$