barkhausen institut

Microkernel Construction

Exercise 1: Kernel Entry / Exit

Nils Asmussen, Viktor Reusch

04/30/2026

Roadmap

- Brief intro/review on kernel bootstrapping

- Start within minimal kernel
- Leave kernel to userland via `iretq`
- Reenter via `syscall`
- Do very basic system calls (`nop`, `add`, ...)

Initial Machine State

- Protected mode, no paging, but segmentation
- All segments: base `0`, limit `0xFFFF_FFFF`
- `CS`: 32bit r+x code segment
- `DS`, `ES`, `FS`, `GS`, `SS`: 32bit r+w data segment
- Exact values are undefined
- See `Multiboot` specification for details

Get the Code

```shell
$ git clone https://github.com/Nils-TUD/MKC.git
$ git checkout exercise1
```

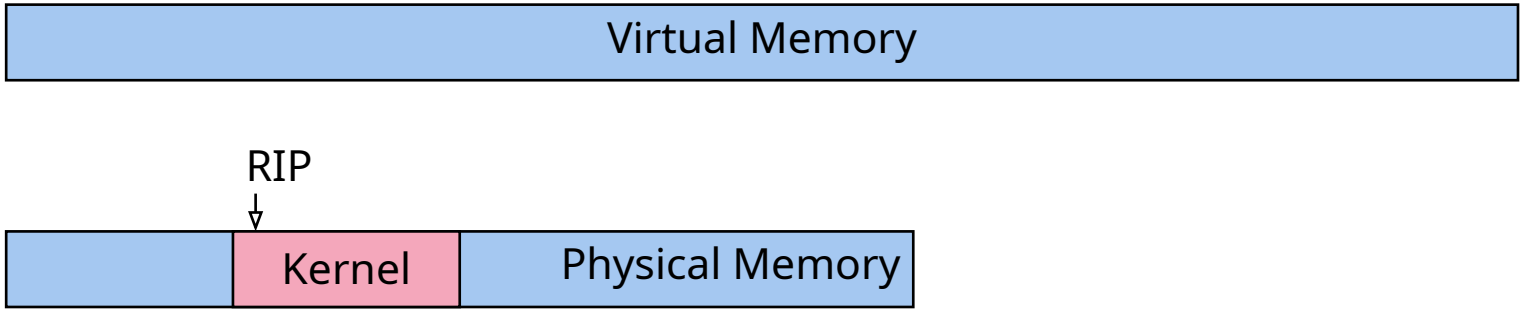
```shell
# build it
$ make
```

```shell
# run it
$ make run
```

The Very First Few Instructions

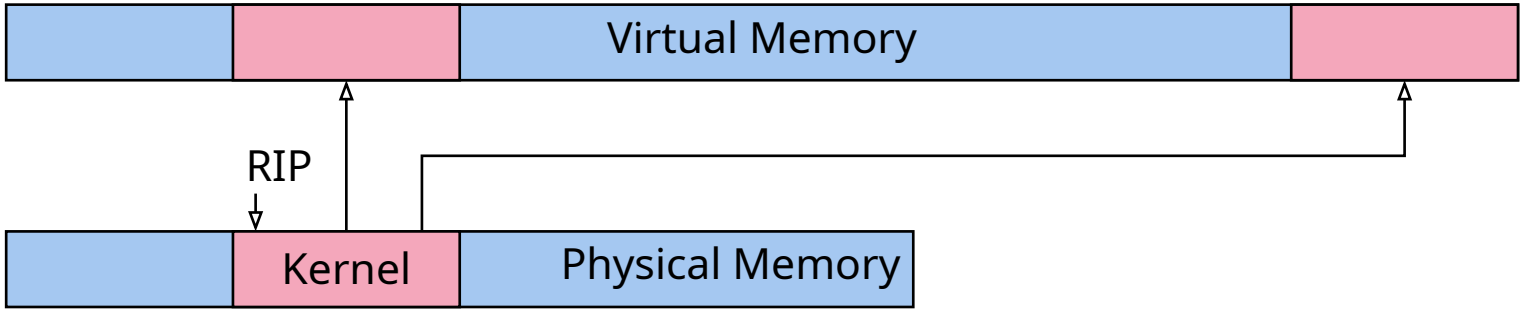
- Open `src/start.S`
- Hard-coded segment descriptor table
- Execution starts at symbol `__start`
- Setup boot page table
- Enable paging
- Load segment selectors
- Call `init()`

Setup Memory



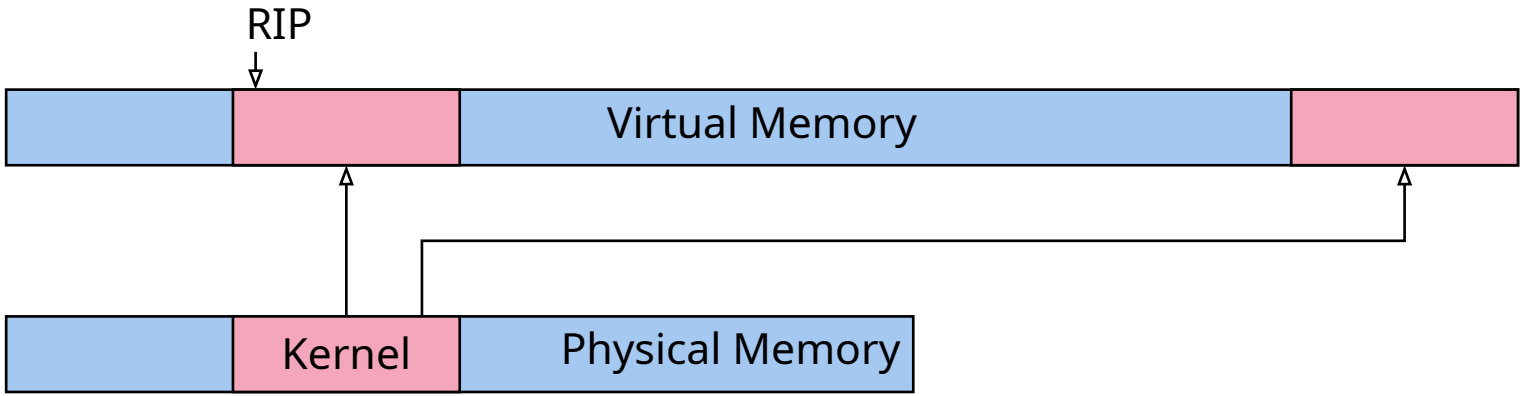
- Setup boot page table

Setup Memory



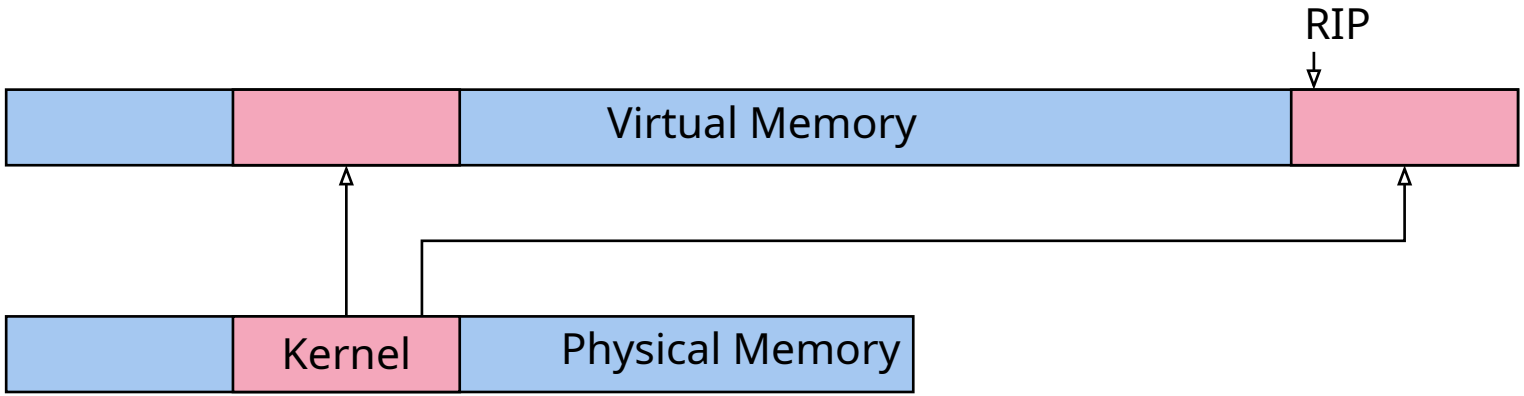
- Setup boot page table
- Establish 1:1 mapping

Setup Memory



- Setup boot page table
- Establish 1:1 mapping
- Enable paging → next instruction fetch from same address in VM

Setup Memory

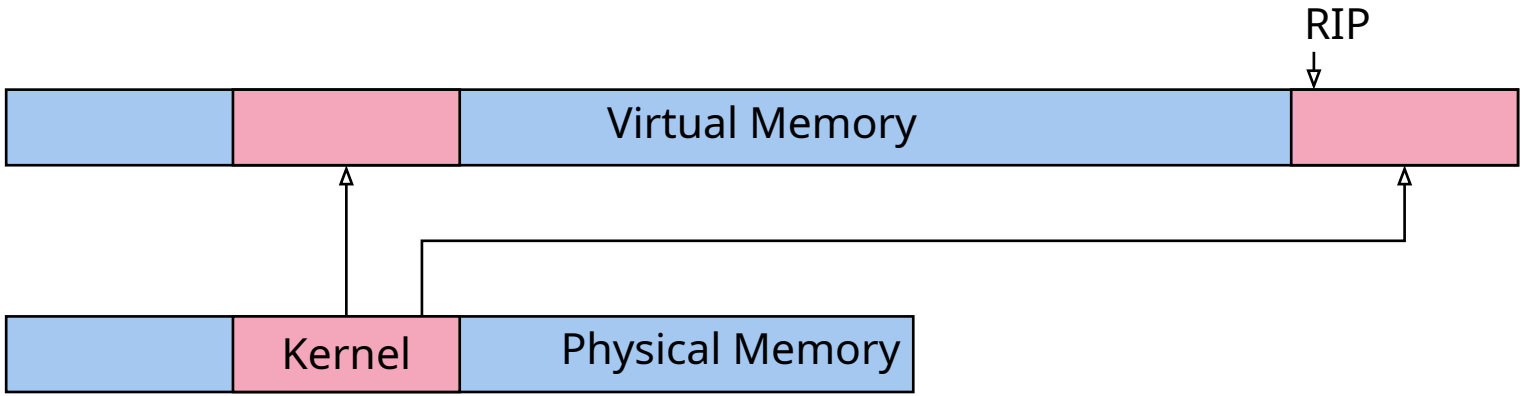


- Setup boot page table
- Establish 1:1 mapping
- Enable paging → next instruction fetch from same address in VM
- Jump to high memory (but still same kernel stack)

Init() ...ialization

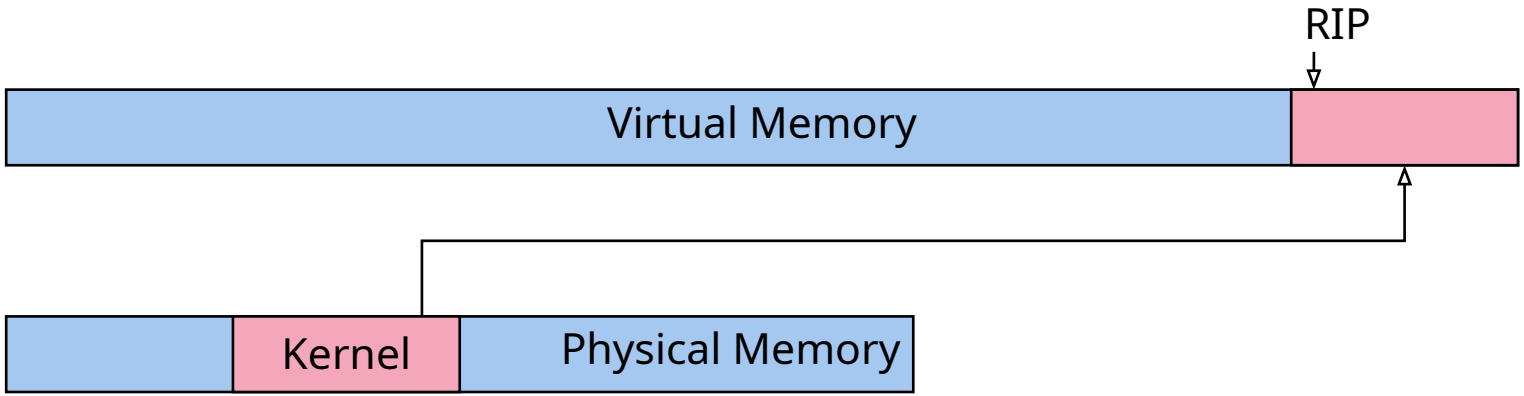
- Open `src/init.cc`
- Setup serial port for early debug output
- Map new kernel stack
- Setup `GDT`, `IDT`, `GSI`, and `TSS`
- Init `PIC`, mask all `IRQs` or install handlers
- Prepare `syscall` (`CS`, `RIP`, disable interrupts on entry)
- Switch kernel stack

Init() ...ialization



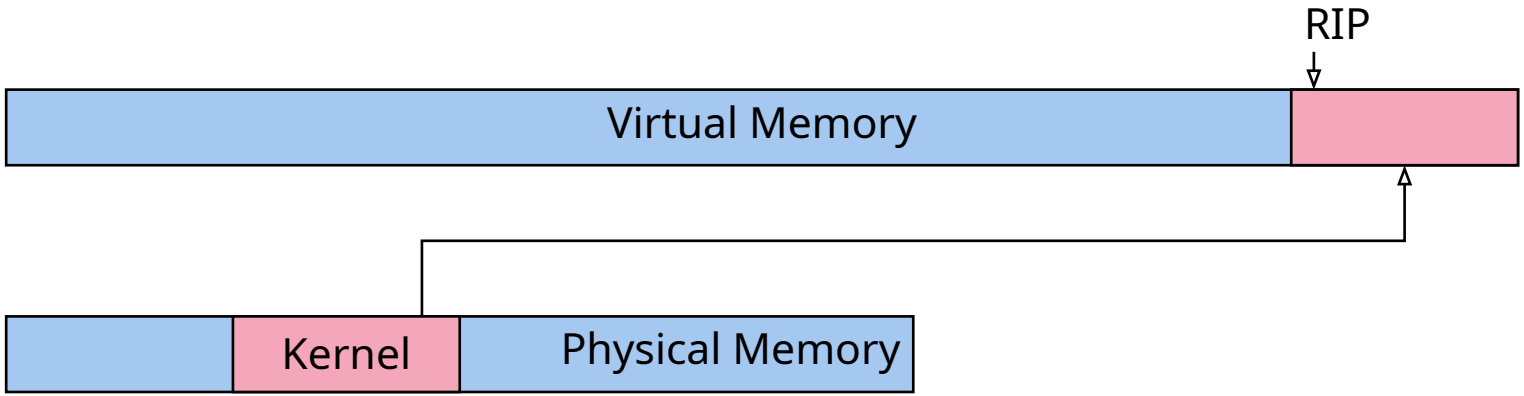
- Again, `src/init.cc`, `bootstrap()`

Init() ...ialization



- Again, `src/init.cc`, `bootstrap()`
- Remove 1:1 mapping and flush `TLB`

Init() ...ialization



- Again, `src/init.cc`, `bootstrap()`
- Remove 1:1 mapping and flush TLB
- Create new EC (thread) for our user-code
- Switch to that EC

Getting out of the Kernel

- Open `src/ec.cc`, `root_invoke()`
- Prepare address space
  - Map 1 page user stack (at address `0x1000`)
  - Map 1 page user code (at address `0x2000`)
- Prepare stack frame to be used with `iretq`
  - User code segment + instruction pointer (`CS`, `RIP`)
  - User stack segment + stack pointer (`SS`, `RSP`)
  - No data segment for now
- `iretq`: loads `CS:RIP`, `SS:RSP` and `RFLAGS`

IRET stack layout

- (kernel) `RSP` points to array with: `CS:RIP`, `RFLAGS`, and `SS:RSP`
- `iretq` atomically loads registers and switches from privilege level 0 to 3
- CPU starts executing from new `RIP`

Inline Assembly in a Nutshell

```cpp
asm volatile(
    "nop;"
    : <out> : <in> : <clobber>
);
```

Example:

```cpp
mword i = 2, j = 3;
asm volatile(
    "add %%rbx, %%rax;"
    : "+a" (i) : "b" (j)
);
printf("%d %d\n", i, j);
```

And... ACTION!

- Prepare array with 5 elements and `iretq`
  - `USER_CODE`: user address to exit to
  - `SEL_USER_CODE`: new `CS` (`include/selectors.h`)
  - `0x202`: `EFLAGS`, interrupt enabled flag set and reserved bit
  - `USER_CODE`: new stack pointer (grows down)
  - `SEL_USER_DATA`: new `SS` stack segment
- Open `src/usercode.cc`: `usercode()`
  - 1st fault immediately
  - 2nd reenter kernel via `syscall`
  - 3rd prepare `sysret` by marking `RCX` (for `RIP`) and `R11` (for `RFLAGS`) as clobbered
  - 4th do simple system calls, like add 2 numbers

Hello user

- Open `src/usercode.cc`, function `usercode()`
- To check if everything is ok, fault immediately
  - `asm ("ud2");`
  - `User EXC 0x6 (RIP=0x2000 CR2=0x0)`
- Force a page fault by accessing an address somewhere below `0x1000`
  - `User EXC 0xE (RIP=0x2000 CR2=0x23)`