



NOVA introduction

Michal Sojka¹

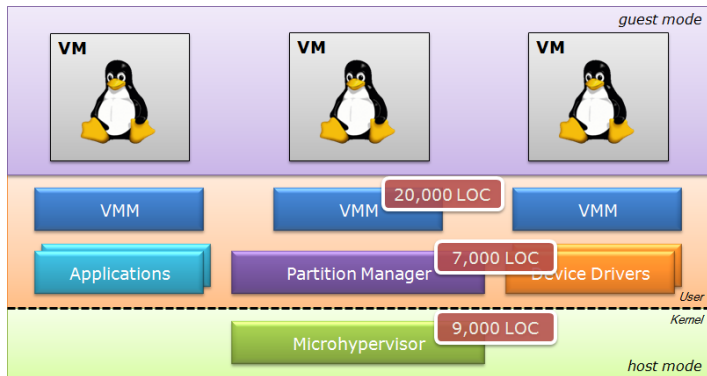
Czech Technical University in Prague,
Email: michal.sojka@cvut.cz

November 12, 2021

¹Based on exercises by Benjamin Engel from TU Dresden.



NOVA microhypervisor



- ▶ Research project of TU Dresden (< 2012) and Intel Labs (≥ 2012).
- ▶ <http://hypervisor.org/>, x86, GPL.
- ▶ We will use a stripped down version (2 kLoC) of the microhypervisor (kernel).



Table of content

Prerequisites

System booting

Program binaries

ELF headers

Virtual memory management

Additional information



What you need to know?

- ▶ NOVA is implemented in C++ (and assembler).
- ▶ Each user “program” is represented by **execution context** data structure (`class Ec`).
- ▶ The first executed program is called **root task** (similar to `init` process in Unix).



Getting started

```
unzip nova.zip
cd nova
make # Compile everything
make run # Run it in Qemu emulator
```

Understanding qemu invocation

```
qemu-system-i386 -serial stdio -kernel kern/build/hypervisor -initrd user/hello
```

- ▶ Serial line of the emulated machine will go to stdout
- ▶ Address of `user/hello` binary will be passed to the kernel via *Multiboot info* data structure

Source code layout

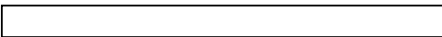
- ▶ `user/` – user space code (hello world + other simple programs)
- ▶ `kern/` – stripped down NOVA kernel
 - ▶ you will need to modify `kern/src/ec.cc`



NOVA booting

1. CPU reset, BIOS executes
2. Bootloader loads the kernel binary and user application into memory
3. Bootloader starts executing the kernel (`kern/src/start.S`)
4. Kernel initializes CPU and paging (virtual memory) (`start.S`, `init.cc`)
5. Kernel allocates and maps one page for application stack (`kern/src/ec.cc`, `Ec::root_invoke()`)
6. Kernel looks at ELF program header to see where the application wants to be loaded (`Ec::root_invoke()`).
7. Kernel creates page table entries according to the ELF header (`Ec::root_invoke()`)
8. Kernel jump to the application entry point (`sysexit` in `Ec::root_invoke()`)

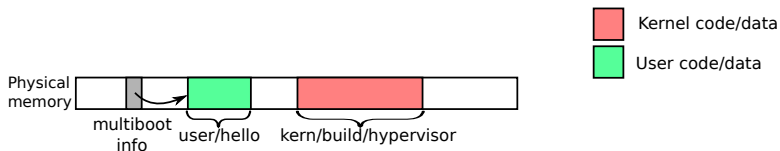
Physical
memory





NOVA booting

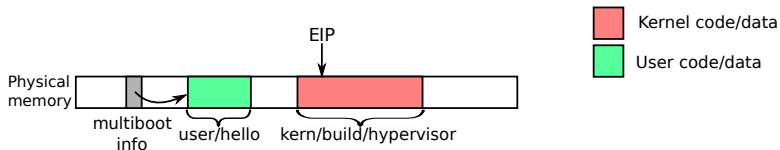
1. CPU reset, BIOS executes
2. Bootloader loads the kernel binary and user application into memory
3. Bootloader starts executing the kernel (`kern/src/start.S`)
4. Kernel initializes CPU and paging (virtual memory) (`start.S`, `init.cc`)
5. Kernel allocates and maps one page for application stack (`kern/src/ec.cc`, `Ec::root_invoke()`)
6. Kernel looks at ELF program header to see where the application wants to be loaded (`Ec::root_invoke()`).
7. Kernel creates page table entries according to the ELF header (`Ec::root_invoke()`)
8. Kernel jump to the application entry point (`sysexit` in `Ec::root_invoke()`)





NOVA booting

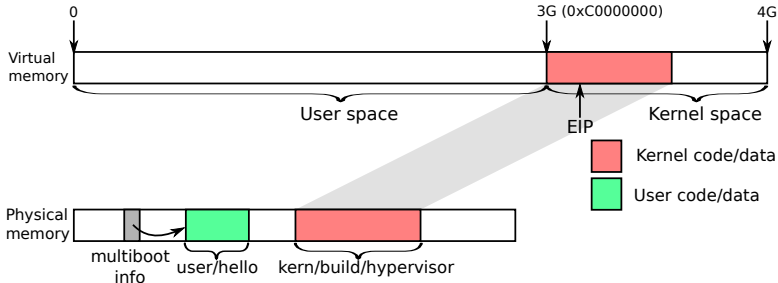
1. CPU reset, BIOS executes
2. Bootloader loads the kernel binary and user application into memory
3. Bootloader starts executing the kernel (`kern/src/start.S`)
4. Kernel initializes CPU and paging (virtual memory) (`start.S`, `init.cc`)
5. Kernel allocates and maps one page for application stack (`kern/src/ec.cc`, `Ec::root_invoke()`)
6. Kernel looks at ELF program header to see where the application wants to be loaded (`Ec::root_invoke()`).
7. Kernel creates page table entries according to the ELF header (`Ec::root_invoke()`)
8. Kernel jump to the application entry point (`sysexit` in `Ec::root_invoke()`)





NOVA booting

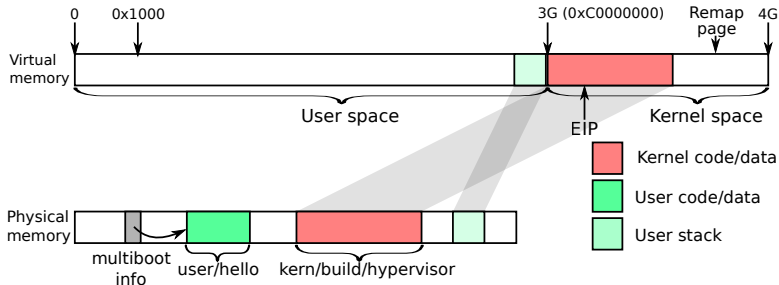
1. CPU reset, BIOS executes
2. Bootloader loads the kernel binary and user application into memory
3. Bootloader starts executing the kernel (`kern/src/start.S`)
4. Kernel initializes CPU and paging (virtual memory) (`start.S`, `init.cc`)
5. Kernel allocates and maps one page for application stack (`kern/src/ec.cc`, `Ec::root_invoke()`)
6. Kernel looks at ELF program header to see where the application wants to be loaded (`Ec::root_invoke()`).
7. Kernel creates page table entries according to the ELF header (`Ec::root_invoke()`)
8. Kernel jump to the application entry point (`sysexit` in `Ec::root_invoke()`)





NOVA booting

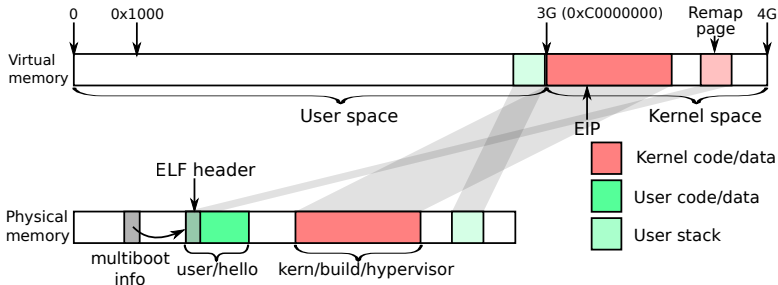
1. CPU reset, BIOS executes
2. Bootloader loads the kernel binary and user application into memory
3. Bootloader starts executing the kernel (`kern/src/start.S`)
4. Kernel initializes CPU and paging (virtual memory) (`start.S`, `init.cc`)
5. Kernel allocates and maps one page for application stack (`kern/src/ec.cc`, `Ec::root_invoke()`)
6. Kernel looks at ELF program header to see where the application wants to be loaded (`Ec::root_invoke()`).
7. Kernel creates page table entries according to the ELF header (`Ec::root_invoke()`)
8. Kernel jump to the application entry point (`sysexit` in `Ec::root_invoke()`)





NOVA booting

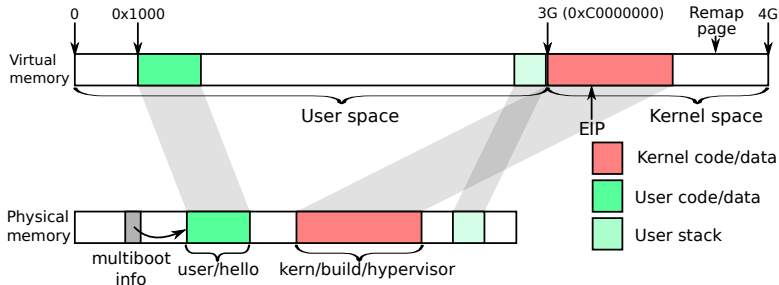
1. CPU reset, BIOS executes
2. Bootloader loads the kernel binary and user application into memory
3. Bootloader starts executing the kernel (`kern/src/start.S`)
4. Kernel initializes CPU and paging (virtual memory) (`start.S`, `init.cc`)
5. Kernel allocates and maps one page for application stack (`kern/src/ec.cc`, `Ec::root_invoke()`)
6. Kernel looks at ELF program header to see where the application wants to be loaded (`Ec::root_invoke()`).
7. Kernel creates page table entries according to the ELF header (`Ec::root_invoke()`)
8. Kernel jump to the application entry point (`sysexit` in `Ec::root_invoke()`)





NOVA booting

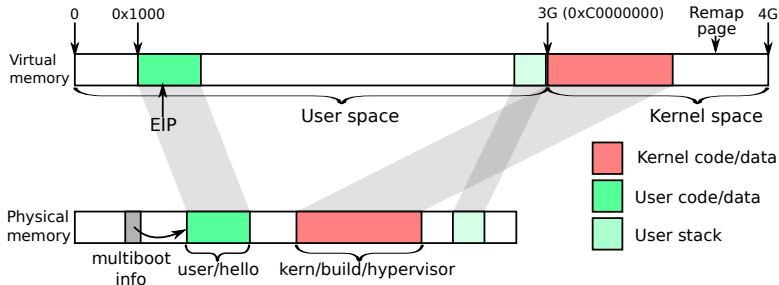
1. CPU reset, BIOS executes
2. Bootloader loads the kernel binary and user application into memory
3. Bootloader starts executing the kernel (`kern/src/start.S`)
4. Kernel initializes CPU and paging (virtual memory) (`start.S`, `init.cc`)
5. Kernel allocates and maps one page for application stack (`kern/src/ec.cc`, `Ec::root_invoke()`)
6. Kernel looks at ELF program header to see where the application wants to be loaded (`Ec::root_invoke()`).
7. Kernel creates page table entries according to the ELF header (`Ec::root_invoke()`)
8. Kernel jump to the application entry point (`sysexit` in `Ec::root_invoke()`)





NOVA booting

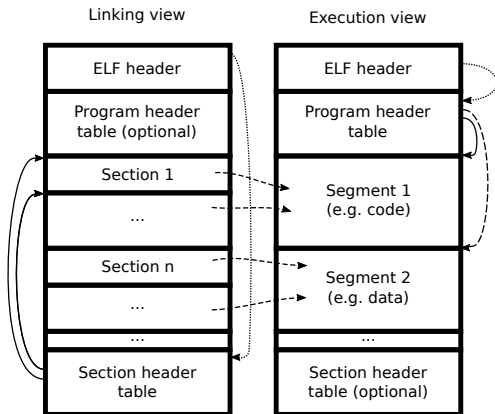
1. CPU reset, BIOS executes
2. Bootloader loads the kernel binary and user application into memory
3. Bootloader starts executing the kernel (`kern/src/start.S`)
4. Kernel initializes CPU and paging (virtual memory) (`start.S`, `init.cc`)
5. Kernel allocates and maps one page for application stack (`kern/src/ec.cc`, `Ec::root_invoke()`)
6. Kernel looks at ELF program header to see where the application wants to be loaded (`Ec::root_invoke()`).
7. Kernel creates page table entries according to the ELF header (`Ec::root_invoke()`)
8. Kernel jump to the application entry point (`sysexit` in `Ec::root_invoke()`)



Program binaries

Executable and Linkable Format (ELF)

<http://www.sco.com/developers/devspecs/gabi41.pdf>, chapter 4



- ▶ Composed of headers, segments and sections
- ▶ One segment contains one or more sections
- ▶ A section may or may not belong to a segment
- ▶ All of this is controlled by “linker scripts” – they tell the linker how to link the program (more info later).



ELF header

elf.h, class Eh

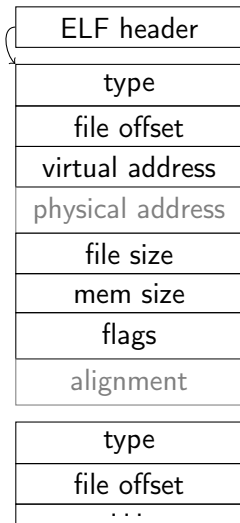
magic: 7f 'E' 'L' 'F'			
class	data	version	padd.
padding			
padding			
type		machine	
version			
entry			
ph_offset			
sh_offset			
flags			
eh_size		ph_size	
ph_count		sh_size	
sh_count		strtab	

- ▶ Each binary starts with this header
- ▶ Can be shown by `readelf -h`
- ▶ The code in `Ec::root_invoke`:
 - ▶ Checks magic, `data == 1` and `type == 2`
 - ▶ Reads entry point, i.e. user EIP
 - ▶ Reads information about program headers
 - ▶ `ph_count`: number of program headers
 - ▶ `ph_offset`: where within the file the program header table starts



Program header table

elf.h, class Ph



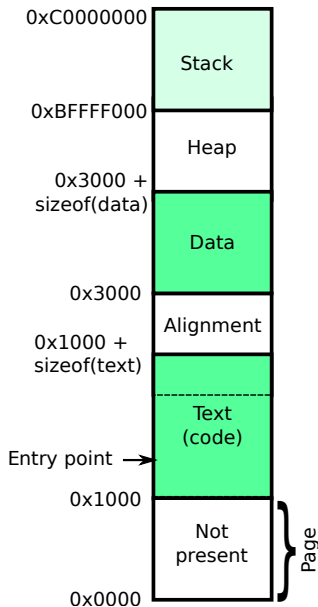
- ▶ Describes segments of the binary
- ▶ `Ec::root_invoke` does:
 - ▶ If `type == PT_LOAD (1)` \Rightarrow map this segment to memory
 - ▶ If flags has `PF_W (2)` set \Rightarrow make the page(s) writable
 - ▶ Read offset to know where this segment starts relative to the beginning of the file
 - ▶ Read virtual address to know where to map this segment to
 - ▶ Read file/mem size to know the segment size (in file and memory)



User space memory map

As defined by a so called “linker script” (user/linker.ld)

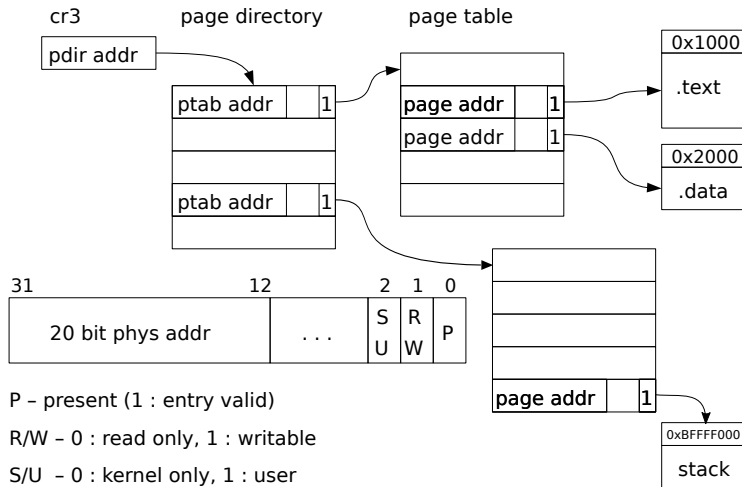
- ▶ Stack is expected to go from 0xC000 0000 downwards.
- ▶ The first page is left “not present” to catch NULL pointer dereference errors.
- ▶ Entry point and sizes of text/data sections is stored in various headers in the program binary.





Understanding Ptab::insert_mapping – x86 page tables

See `kern/src/ptab.cc`



See also Intel System Programming Guide, sect. 4.3 “32-bit paging” ([link](#))



Additional information



Linker script

Linker scripts tell the linker how to link the program, i.e.

- ▶ which sections go to which segment,
- ▶ at which address the segments should be loaded, etc.
- ▶ Documentation: run “info ld Scripts”
 - ▶ <https://sourceware.org/binutils/docs-2.37/ld/Scripts.html>

user/linker.ld

- ▶ Program entry point at symbol **`__start`**
- ▶ Two segments: **text** (5 \Rightarrow RX) and **data** (6 \Rightarrow RW)
- ▶ Put sections **.text** into segment **text** and sections **.data**, **.rodata** and **.bss** into segment **data**²
- ▶ **ALIGN** end of data (and start of text) to a page boundary (0x1000)

².text sections contain executable code, .data initialized (global) variables, .rodata read-only data (e.g. declared with const qualifier) and .bss contains uninitialized (zeroed) data.



Program startup – user/src/start.S

Code that runs before `main()`

```
.text
.global _start
_start:
    mov $stack_top, %esp
    call main
    ud2
```

- ▶ Put this into the `.text` section
- ▶ Define global symbol `_start`
- ▶ Setup a stack by loading the address of `stack_top` into `esp` (`stack_top` is defined in `linker.ld`)
- ▶ Call function `main()`
- ▶ If `main` returns, execute “undefined” instruction. When this is executed, CPU generates an exception and the kernel tells us about that.



Building and inspecting the user program

- ▶ Goto the user directory and run `make` there
- ▶ Inspect the binary by `nm user/hello`

```
00003000 T main
00002000 D stack_top
00003029 T _start
```

- ▶ There are three symbols in the text section (T) and three in data section (D)
- ▶ Decode headers: `readelf -h -l user/hello` or `objdump -x user/hello`



Understanding kernel exceptions

- ▶

```
void main() {  
    *((int*)0x234) = 0x12; /* Write 0x12 to address 0x234 */  
}
```
- ▶ Address 0x234 is in the zeroth page (0x0 – 0x3ff), which is not present (i.e. the present flag in the page table entry is 0).
- ▶ Accessing this page generates a “Page fault” exception.
- ▶ The kernel “handles” the exception by printing useful information about it.
- ▶ You can try the above program (stored in user/pagefault.c) by running:

```
make -C user pagefault  
qemu-system-i386 -serial stdio -kernel kern/build/hypervisor \  
-initrd user/pagefault
```

It produces this output:

```
NOVA Microhypervisor 0.3 (Cleetwood Cove)  
  
Ec::handle_exc Page Fault (eip=0x3000 cr2=0x234)  
eax=0xcfffffff ebx=0x1803000 ecx=0x5 edx==0xc0009000  
esi=0xdf001074 edi=0x5 ebp=0x1801000 esp==0x1ffc  
unhandled kernel exception
```

- ▶ eip – the instruction that caused the fault, cr2 – the faulty address
- ▶ Find the address 0x3000 (eip) in `objdump -S user/pagefault`