Virtual memory, user program execution

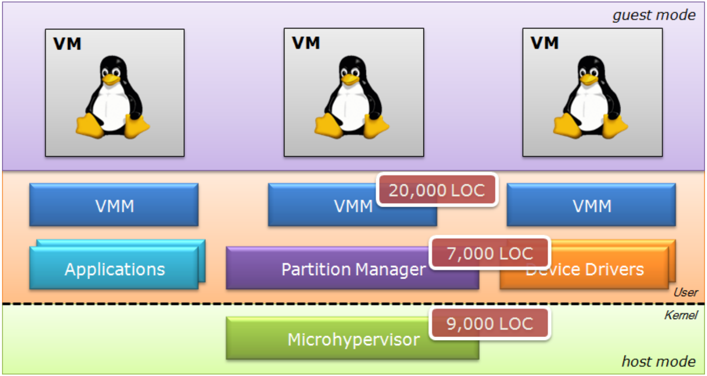
Michal Sojka[1]

Czech Technical University in Prague,
Faculty of Electrical Engineering
Email: sojkam1@fel.cvut.cz

April 27, 2016

[1]Based on exercises by Benjamin Engel from TU Dresden.

NOVA microhypervisor



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

Assignment

Extend the kernel so that it is able to run a user space application. The kernel will be run in hardware emulator *qemu* (i.e. in a virtual machine).

Initial state

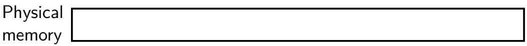
- CPU and kernel initialized
- Application binary loaded in memory

Steps to do

1. Read and parse program header from ELF binary
2. Setup page table entries so that the application can run
3. Jump to the application entry point (and switch the CPU from kernel to user mode)

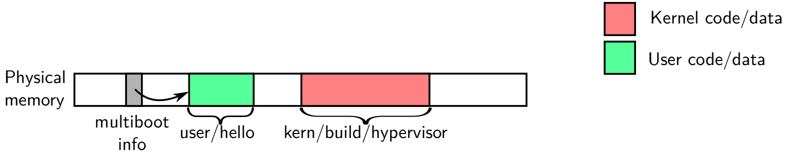
Graphical representation of the assignment

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. You look at ELF program header to see where the application wants to be loaded.
7. You create page table entries according to the ELF header
8. You jump to application entry point (using iret instruction)



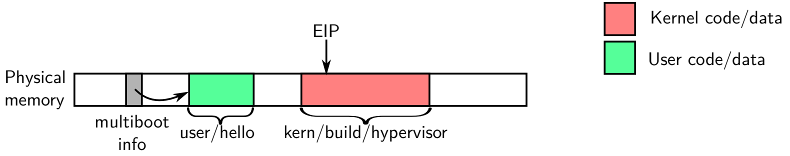
Graphical representation of the assignment

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. You look at ELF program header to see where the application wants to be loaded.
7. You create page table entries according to the ELF header
8. You jump to application entry point (using iret instruction)



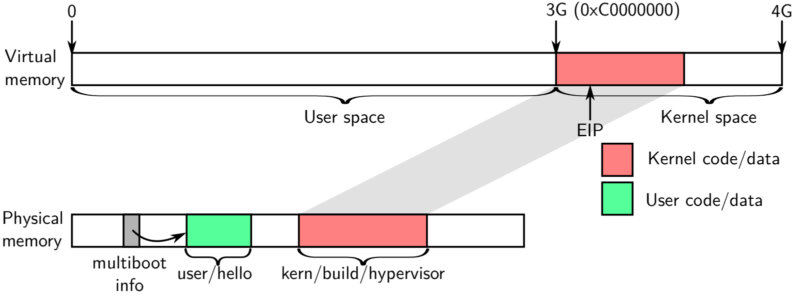
Graphical representation of the assignment

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. You look at ELF program header to see where the application wants to be loaded.
7. You create page table entries according to the ELF header
8. You jump to application entry point (using iret instruction)



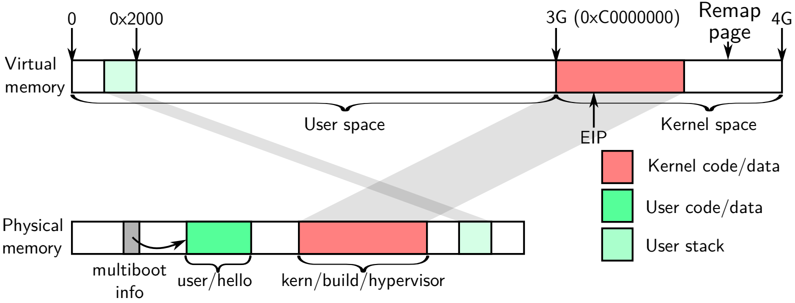
Graphical representation of the assignment

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. You look at ELF program header to see where the application wants to be loaded.
7. You create page table entries according to the ELF header
8. You jump to application entry point (using iret instruction)



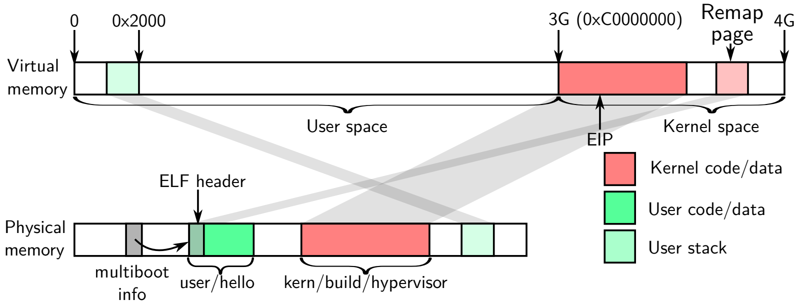
Graphical representation of the assignment

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. You look at ELF program header to see where the application wants to be loaded.
7. You create page table entries according to the ELF header
8. You jump to application entry point (using iret instruction)



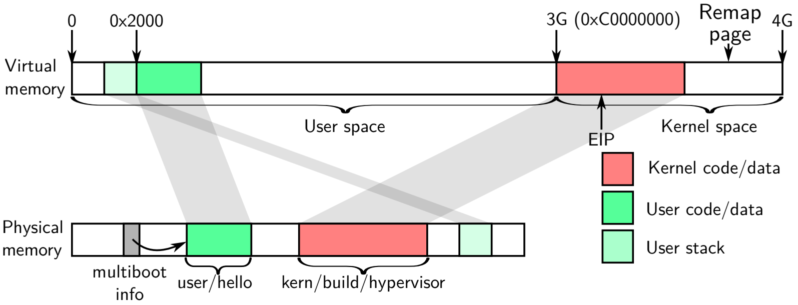
Graphical representation of the assignment

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. You look at ELF program header to see where the application wants to be loaded.
7. You create page table entries according to the ELF header
8. You jump to application entry point (using iret instruction)



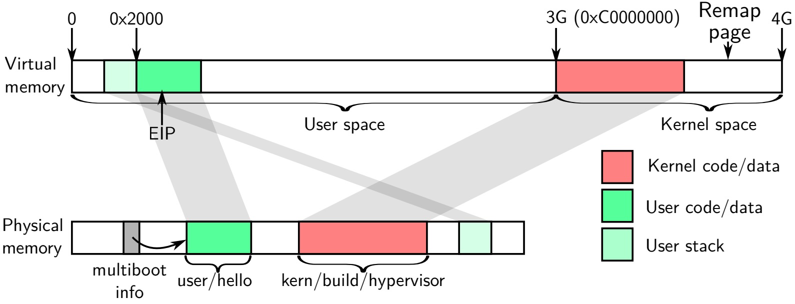
Graphical representation of the assignment

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. You look at ELF program header to see where the application wants to be loaded.
7. You create page table entries according to the ELF header
8. You jump to application entry point (using iret instruction)



Graphical representation of the assignment

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. You look at ELF program header to see where the application wants to be loaded.
7. You create page table entries according to the ELF header
8. You jump to application entry point (using iret instruction)



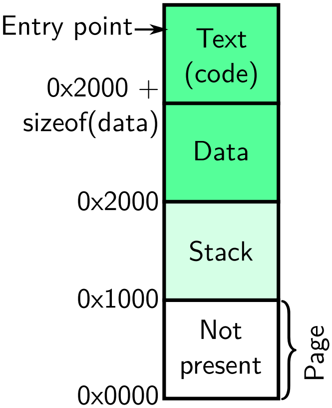
What you need to know?

- 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).
- Where the user program expects to be loaded in memory. Where the program expects to have stack, data, code.
- Structure of the program binary file.

User space memory map

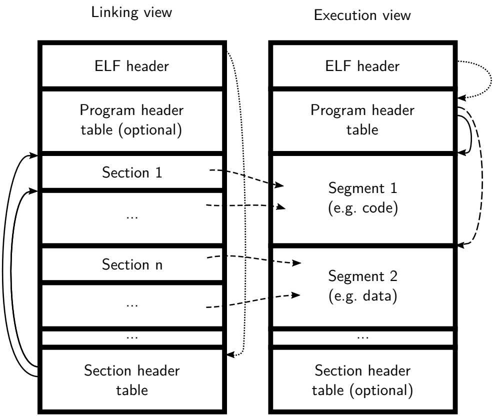
As defined by so called "linker script" (user/linker.ld)

- Stack is expected to go from 0x2000 downwards.
- First page is left "not present" to catch NULL pointer deference errors.
- Entry point and sizes of text/data sections is stored in various headers in the program binary.



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

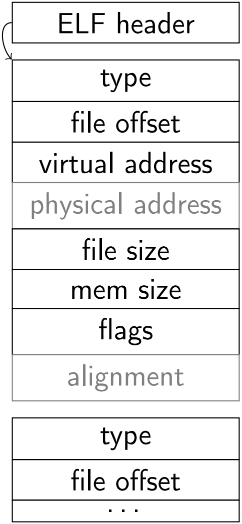
elf.h, class Eh

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`
- Your code should:
  - Check magic, data == 1 and type == 2
  - Read entry, i.e. user EIP
  - Read 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
- Your program should:
  - If type == PT_LOAD (1) ⇒ map this segment to memory
  - If flags has PF_W (2) set ⇒ the memory must be 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)

Getting started

```
tar xf osd-e2.tar.gz
cd osd-e2
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`

Step 1 – Decode ELF headers

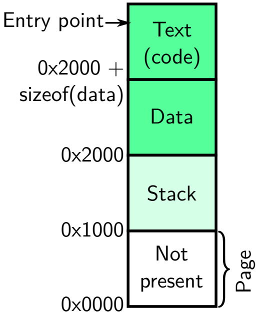
1. Find TODO in `Ec::root_invoke()`
2. `mod.mod_start` is the physical address of the user binary
3. Remap and read the ELF header
4. Remap program header table and iterate over all (two) program segments
   4.1 If type != PT_LOAD, ignore this segment
   4.2 Print all virt/phys addresses and mem sizes
      - Align them properly to 4k page boundaries!
      - phys/virt addresses: align down
      - mem size: align up

When anything goes wrong here, call `panic ("ELF error\n");`

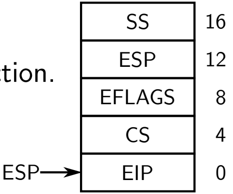
Step 2 – Setup page table entries for the application

1. Some sanity checks:
   - File size and mem size should be equal
   - Virtual address and file offset should be equal (modulo page size)
2. Add mapping for all pages in all segments:
   `Ptab::insert_mapping (virt, phys, attr)`
   - Inserts a mapping from virtual address `virt` to physical address `phys` with attributes `attr`
   - If `flags & Ph::PF_W` ⇒ should be mapped writable, thus `attr = Ptab::PRESENT | Ptab::RW | Ptab::USER`, otherwise `attr = Ptab::PRESENT | Ptab::USER`
   - See the *Ptab::insert_mapping* slide later and class Ph in kernel/include/elf.h

Step 3 - First switch to user space



- After mapping the memory to the right place, we can start executing application code.
- Use `iret` instruction to switch the CPU from kernel to user mode and jump to the user code.
- `iret` takes the operands from the stack!
- Prepare an array with 5 elements:
  - Entry point: user instruction pointer to return to
  - `SEL_USER_CODE`: new CS (`include/selectors.h`)
  - `0x200`: EFLAGS – just set interrupt enabled flag
  - `0x2000`: user stack pointer
  - `SEL_USER_DATA`: new SS stack segment
- Point ESP to the array and execute `iret` instruction.
- If you are successful, the application prints "Hello world!"



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`"

user/linker.ld

- Program entry point at symbol **_start**
- Two segments: **data** (6 ⇒ RW) and **text** (5 ⇒ RX)
- Put section **.text** into segment **text** and sections **.data**, **.rodata** and **.bss** into segment **data**
- **ALIGN** end of data (and start of text) to page boundary (**0x1000**)

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 **main()**
- If main retuns, execute undefined unstruction. This generates exception and the kernel tells us about that.

Building and inspecting the user program

- Goto user and `make` user binary
- Inspect 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 page zero, which is not present (i.e. present flag in page table entry is 0).
- Access to this page generates "Page fault" exception.
- The kernel "handles" the exception by printing useful information about it.
- After your kernel is capable of running user binaries, running:

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

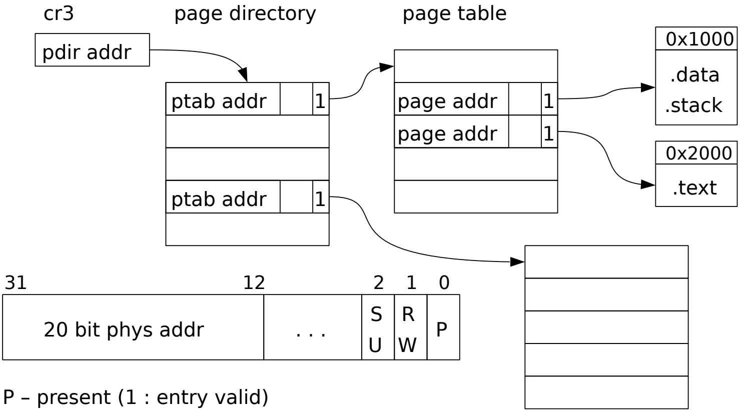
  produces this output:

  ```
  NOVA Microhypervisor 0.3 (Cleetwood Cove)

  Ec::handle_exc Page Fault (eip=0x3000 cr2=0x234)
  eax=0xcffffdc 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`
  ```

Understanding Ptab::insert_mapping – x86 page tables

See `kern/src/ptab.cc`



P – present (1 : entry valid)

R/W – 0 : read only, 1 : writable

S/U – 0 : kernel only, 1 : user

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