barkhausen institut

Microkernel Construction

Exercise 2: Multiboot

Nils Asmussen

05/07/2026

Roadmap

- User `start.S` and linker script
- Multiboot header
- Map physical memory
- ELF

- Hands-on
  - Parse Multiboot Info and ELF header
  - Load and execute user binary

Get the Code

```shell
$ git clone https://github.com/Nils-TUD/MKC
$ git checkout exercise2
```

Shell

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

Shell

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

Shell

Linker and Assembly

- Open `user/src/start.S`
  - In the `.text` segment
  - Global symbol `__start:`
  - Setup a stack: load address of `STACK_TOP` into `RSP`
  - Call `main_func()`

- Open `user/src/linker.ld`
  - Program entry point at symbol `__start`
  - Two segments: `data` (rw) and `text` (rx)
  - Put `.text` into segment `text` and `.data` / `.bss` into `data`
  - `ALIGN` stack and text to page boundary (`4K`)

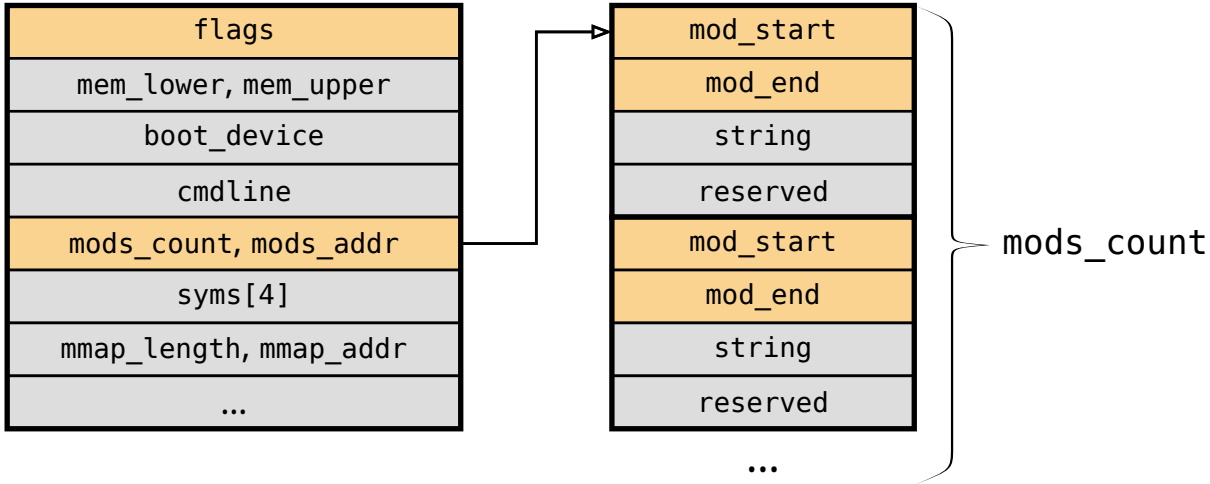
Building and Loading the User Program

- `make` user binary and go to `user/build`
- Inspect binary via `nm user.nova.debug`

```
000000000000200e T main_func                    Text
0000000000002000 D STACK_TOP
0000000000002000 T __start
```

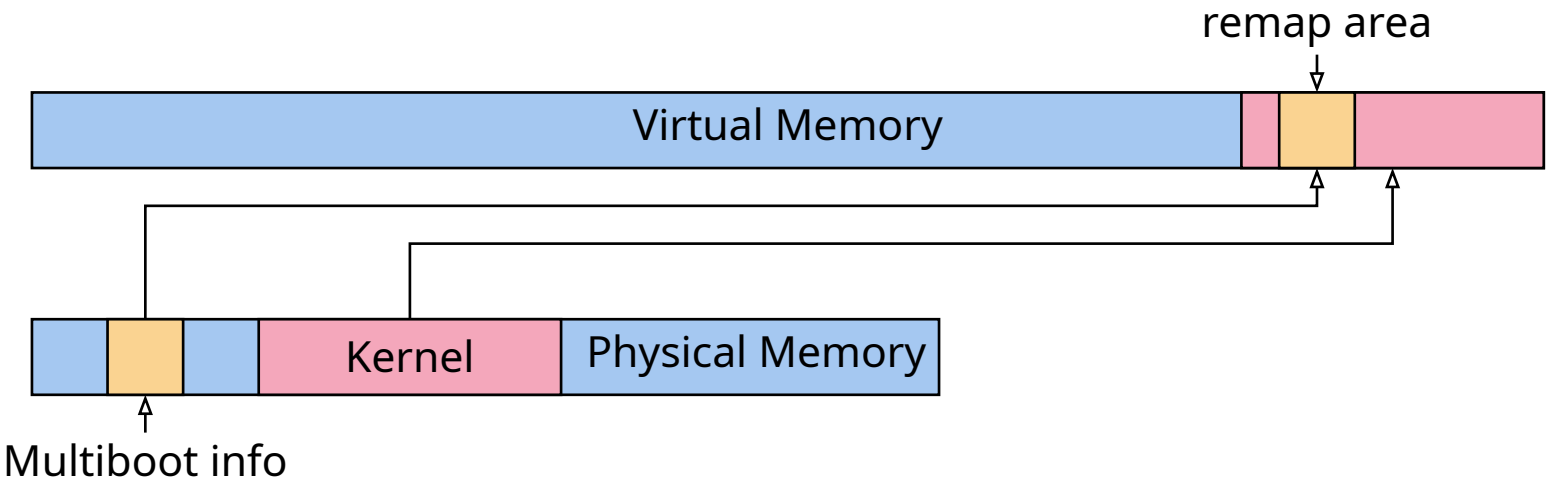
- There are two symbols in the text segment and one in data
- Next: pass binary to the boot loader as a boot module after the kernel
  - `cat Makefile`

Multiboot Information `multiboot.h`



- `flags` is required, all other fields are optional
- If `flags[3]` is set, `mods_count` and `mods_addr` are valid
- `mods_addr` is physical address module array with length `mods_count`
- The boot loader passes a Multiboot info pointer to the kernel

Remap Multiboot Structures



- Multiboot info address and `mods_addr` are **physical** addresses
- Map them temporarily into the kernel remap area:
  - Use `void * Ptab::remap(phys_addr)`
  - `remap()` replaces the previous mapping, so the old pointer becomes invalid

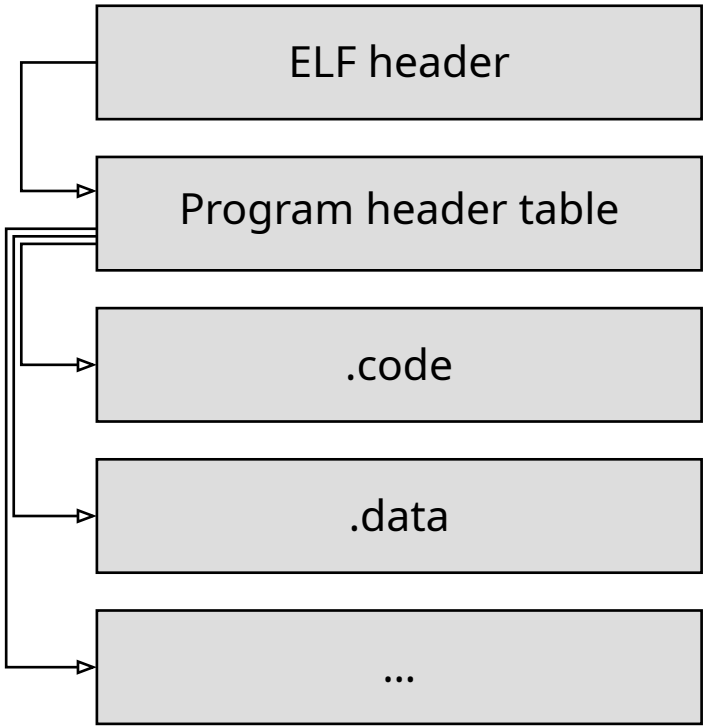
Task 1: Find and Map Binary

- Open `kern/src/ec.cc`: `root_invoke()`
- `Ec::current->regs.rdi` contains the MBI pointer

- Remap Multiboot info, check `flags:3`, get `mods_addr` and module count
- Remap Multiboot module structure, print start/end address of user binary
- Remap the user binary itself; it is an ELF object

- See `kern/include/multiboot.h` and `elf.h`

Executable and Linkable Format (ELF)



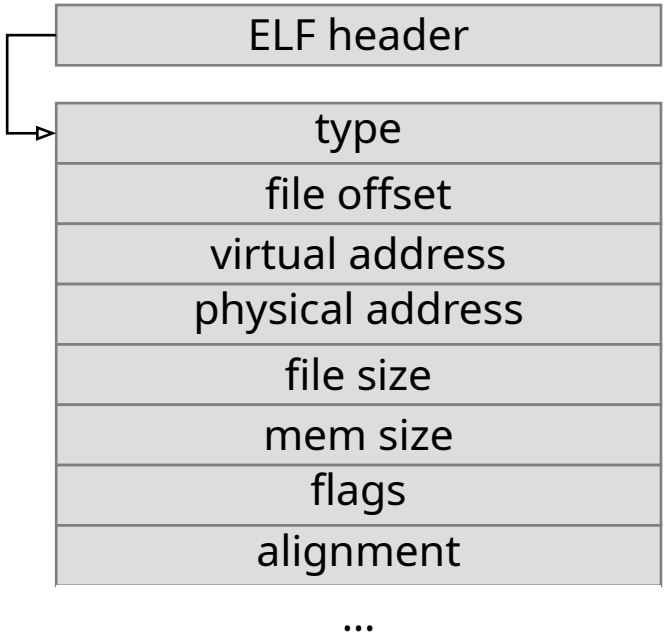
- ELF header contains the offset of the program header table (`ph_offset`)
- The program header table describes the segments used at runtime

ELF Header Format `elf.h`

magic: 7f ‘E’ ‘L’ ‘F’			
class	data	version	osabi
abi version			
padding			
type		machine	
version			
entry			
ph_offset			
sh_offset			
flags			
eh_size		ph_size	
ph_count		sh_size	
sh_count		strtab	

- Check `magic`, `data` (1) and `type` (2)
- `entry`: user `RIP`
- `ph_count`: number of program headers
- `ph_offset`: where the program header table starts within the file

Program Header Table



- Load segments with `type = PT_LOAD (1)`
- `flags:2`: segment is writable?
- `offset`: segment start within file
- `virtual address`: where to map it
- `file size` / `mem size`: segment size in file and memory

Adding Page Table Entries

- Sanity checks:
  - File size and memory size should be equal
  - Virtual address and file offset should be equal modulo page size
- `Ptab::insert_mapping(virt, phys, attr)`
  - Inserts a mapping from `virt` to `phys` with attributes `attr`
- See class `Ph` in `kernel/include/elf.h`
  - If `flags & Ph::PF_W`, the page should be writable
    - writable: `attr = 7`
    - otherwise `attr = 5`
- Add mappings for all pages in all segments
- Use `ret_user_iret()` to start the user program

Task 2: Decode ELF and Start Program

- Continue in `root_invoke()`
  - The user binary is still mapped in
- Set `current->regs.rip` to the correct entry point

- Remap the program header table and iterate over both program headers
- If `type != PT_LOAD`, ignore that segment

- Align everything to 4 KiB page boundaries
  - Physical and virtual addresses: align down
  - Memory size: align up

- Print all virtual and physical addresses and memory sizes