Documentation – `f7e26f1`

blog_os

27.11.2025

ThePerkinrex

Contents

1. Introduction

This is an OS that initially was based on Philipp Oppermann's `blog_os`[1] and was later expanded through different sources, mainly from OSDev Wiki[2].

1.1. Functionality

The kernel currently has the following features:

- VGA framebuffer printing
- Serial printing
- Memory paging
- Cooperative kernel multitasking with stack switching
- Userspace process execution in ring3, with ELF loading
- Userspace calls into the kernel, with syscalls (`int 0x80`), and process switching on those syscalls.
- Process & task exiting.

The current WIP features are:

- VFS and user FS API
- Device buses & PCI
- Driver API

Future expected features are:

- StdIO for processes, that could be redirected to different outputs (serial terminals, files…)
- Preemptive task switching
- Advanced task scheduler
- RAM disk support
- Devices on FS tree
- Simple shell & utilities

1.2. Building

To build this OS, the `cargo-make` system is used, so to get a complete OS image, just one command is needed: `cargo make build` at the root of the project. The runner executable will print where the image is located, which will depend on the build profile.

- For debug builds, use `cargo make build`.
- For release builds, use `cargo make -p release build`

Other dev utilities are provided by the `cargo-make` system:

- `cargo make format`: Apply `cargo fmt` to the whole project

[1] https://os.phil-opp.com/
[2] https://wiki.osdev.org/Expanded_Main_Page

- `cargo make docs`: Apply `cargo doc` to the whole project. Each index for each crate/workspace is printed.
- `cargo make pdf`: (*`typst` executable is needed*) builds this pdf.

1.3. Running

To run this OS, `cargo-make` can also be used. QEMU for x86-64 needs to be installed.

- `cargo make run` will start the OS, with a VGA display, and serial output in the terminal.
- `cargo make -p no_display run` will start the OS, without a VGA display, and serial output in the terminal.
- `cargo make -p gdb run` will start the OS, without a VGA display, and serial output in the terminal, attaching the `gdb` debugger to it and stopping immediately, *before the bootloader runs and load the kernel in memory.*

2. Packages and crates

2.1. `kernel`: The Kernel

This crate contains all the main code related with the kernel, and the custom test harness for running it. It is split in many modules:

- `lib.rs`: The base, this contains the initial function for the normal and test execution paths, as well as the panic handlers.
- `main.rs`: It contains the entry point for the OS from the bootloader.
- `allocator.rs`: The heap allocator implementation, based on the *talc* allocator, with heap growth
- `config.rs`: The static config to be provided to the bootloader for memory mappings.
- `dwarf.rs`: DWARF debug format loading utilities for ELF executables.
- `elf.rs`: ELF loading utilities.
- `fs.rs`: root module for any OS filesystems:
  - `fs/ramfs.rs`: The initial *RamFS* implementation
- `gdt.rs`: TSS and GDT setup code, and reloading for custom allocated stacks setup with guard pages.
- `interrupts.rs`: Interrupt handlers and setup (PICS, GPF, PF, `int 0x80`…)
  - `interrupts/info.rs`: Static info about the pointers to Interrupt Handlers, for backtracing.
  - `interrupts/syscalls.rs`: *Syscall* handlers and syscall tail code.
- `io.rs`: All the code related to printing to the screen and serial port. Hopefully it will be replaced somewhat by terminals in the VFS (char devices).

    - `io/framebuffer.rs`: Printing to the VGA framebuffer
    - `io/logging.rs`: Logging infrastructure of the kernel
    - `io/qemu.rs`: Utility for using the QEMU provided io-port for existing with a code.
    - `io/serial.rs`: Printing to the UART serial port
- `memory.rs`: Paging setup and frame allocator/deallocator
    - `memory/multi_l4_paging.rs`: A page table manager that handles multiple L4 page tables with shared kernel tables, and different userspace tables.
    - `memory/pages.rs`: Virtual memory region allocator, to allocate regions of virtual pages, without mapping them, for different usages.
- `multitask.rs`: Kernel task (with multiple kernel stacks) switching infrastructure.
    - `multitask/lock.rs`: Kernel reentrant locking for tasks (without task switching or waiting)
- `priviledge.rs`: Ring 3 jump code from ring 0
- `process.rs`: Data structures and management for userspace processes linked to kernel tasks.
- `progs/`: Copied userspace programs for loading.
- `rand.rs`: PRNG utilities.
- `setup.rs`: Kernel startup code.
- `stack.rs`: Stack creation utility and stack allocator for kernel stacks.
- `unwind.rs`: Backtracing and unwinding infrastructure.
    - `unwind/eh.rs`: Exception handling info extraction from ELFs.
    - `unwind/elf_debug.rs`: Abstractions and helpers for unwinding, in relation with ELF/DWARF info.
    - `unwind/register.rs`: Helper for dealing with register info.

2.2. `kernel-libs`: Libraries used by the kernel

This workspace contains crates that will be used by the kernel, but also some that can also be used by external drivers. This split is useful for code that doesn't necessarily depend on other kernel code, allowing the use of the standard testing framework, and better compile times.

Here there are the following crates:

- VFS:
    - `blog_os_vfs`: Contains kernel-specific VFS code.
    - `blog_os_vfs_api`: VFS API, for FS drivers.
- Device:
- `blog_os-device`: Kernel specific device code, for providing support for the API.
- `blog_os-device-api`: Device API, for drivers (bus, bus device…)
- `blog_os-pci`: The PCI bus driver

- `kernel_utils`: Common utilities used by the kernel, and that can be reused by drivers and other code.
- `api-utils`: Common types used by APIs (common `cglue` code for FFI).

2.3. `qemu-common`: Utilities for interfacing with QEMU and the runner

This is a common crate shared by the kernel testing framework and the runner, allowing for some communication between them through QEMU-specific APIs.

2.4. `runner`: Cargo runner

This is a utility that is used as a cargo runner and a separate binary. It builds the OS image, bundling together the ELF and the bootloader (either for BIOS or UEFI boot). It also supports starting up `gdb`, with config setup for the kernel, and detecting when testing is going on, for better exit codes.

2.5. `userspace`: Anything userspace

Here there lies the `blog_std` crate, for making syscalls and interfacing with the OS.

It also includes a simple test program `test_prog` for testing userspace processes.

3. Kernel startup

This section explains the kernel setup process, formatted in Typst syntax.

3.1. Overview of the Setup Process

The `setup` function performs all early kernel initialization before multitasking begins. It configures CPU tables, interrupt handling, memory management, stack allocation, debugging facilities, and the global kernel state.

3.2. Discovering Layout and Initializing CPU Tables

The kernel begins by discovering available virtual regions using `discover_layout`, which identifies unmapped virtual-address ranges for facilities such as the heap, stack allocator, and drivers.

Then, CPU structures are initialized:
- `gdt::init()` loads the GDT and TSS.
- `interrupts::init_idt()` loads exception handlers.
- `interrupts::init_pics()` initializes hardware interrupt controllers.

If present, the framebuffer is initialized, as well as serial output and logging.

3.3. Logging Kernel Metadata

The kernel logs:

- Virtual and physical kernel addresses
- Kernel image size
- Ramdisk address and length

This provides early diagnostics.

3.4. Paging Setup: bootloader-provided Page Tables

The bootloader provides identity- and higher-half mappings. Paging is initialized via:

```
let page_table = unsafe { init_page_tables(physical_memory_offset) };
```

`init_page_tables`:

- Retrieves the active level-4 page table via `Cr3`.
- Converts its physical address to a virtual address using the physical-memory offset.
- Builds an `OffsetPageTable` over the bootloader-created mappings.

3.5. Physical Frame Allocator

A `BootInfoFrameAllocator` is created using the memory regions from the bootloader. Only regions marked `Usable` are iterated, producing an iterator of 4 KiB frames. After the heap is initialized, the allocator also supports frame deallocation using an internal queue.

3.6. PageTables Structure

A `PageTables` object wraps the bootloader mappings and kernel base virtual offset. It provides:

- Walking and modifying page tables
- Mapping/unmapping pages
- Enumerating mapped ranges

3.7. Virtual Region Allocator

The virtual region allocator identifies large contiguous unmapped virtual memory regions. These are used by:

- Heap initialization
- Kernel stack allocator
- Drivers and subsystems requiring dynamic virtual memory

The allocator is created using:

```
init_region_allocator(&page_table, &layout, &page_table)
```

3.8. AllocKernelInfo

`AllocKernelInfo` stores:

- Page table structure
- Frame allocator
- Virtual region allocator

This is wrapped in a `ReentrantMutex` and globally stored via `Once`.

3.9. Heap Initialization

The heap is implemented using **Talc**. The region allocator assigns contiguous pages for the initial heap:

- A range of virtual pages is allocated.
- Each page is mapped using newly allocated frames.
- Talc claims this region as usable heap memory.

Heap growth (OOM handling) allocates additional virtual regions via the same allocator, maps new pages, and expands Talc's span.

3.10. Unmapping Bootloader Userspace Pages

All mapped pages below the kernel's virtual base are enumerated and unmapped. These are typically leftover mappings (e.g., old GDT state). Each page:

- Is logged
- Unmapped from the page table
- TLB-flushed

3.11. Stack Allocator Initialization

Using the region allocator, the kernel initializes `StackAlloc`. It allocates:

- `esp0`: the privileged ring-0 stack
- `ist_df`: the IST double-fault stack

These stacks are registered in the TSS via `set_tss_guarded_stacks`.

3.12. ELF Loading and Debugging Metadata

The kernel ELF image is reconstructed from the physical-to-virtual offset. From this:

- The ELF is parsed (`SystemElf`)
- Exception-unwind information (`EhInfo`) is extracted
- DWARF debug information is loaded
- An `addr2line` context is constructed when possible

3.13. Constructing KernelInfo

A global static `KernelInfo` is created holding:

- Bootloader metadata (API version, ACPI RSDP, TLS template, ramdisk info)
- Kernel ELF and debug metadata
- AllocKernelInfo
- Stack allocator
- Optional addr2line context

It becomes the central state reference for the kernel.

3.14. Enabling Multitasking

Finally:

- `multitask::init()` initializes the scheduler.
- A task entry is created for the current running context.
- Kernel stack switching becomes possible.

At this point, memory management, interrupts, debug info, and tasking are fully active.

4. Simple I/O

5. Memory

6. Multitasking

7. Processes

8. Interrupts & Syscalls

9. Backtrace, unwinding, & DWARF

10. The VFS & FS API

11. Userspace API & programs