PB173/B Kernel Development

P. Ročkai

October 15, 2020

Organisation

- you write a tiny operating system kernel
- use this document as a ‘todo list’ and a springboard
- use OSDev wiki, architecture manuals, specs, …
- use the chat (lounge) to ask questions

Grading

- there are 6 suggested checkpoints
- some have dependencies, others don't
- meet any 4 to pass the subject
- feel free to negotiate different goals
- set up any schedule you like

Your Own OS (in 6 easy steps)

1. Booting
2. Memory
3. `libc` &c.
4. System Calls
5. Userland
6. Interrupts

Resources

- the OSDev wiki
- OSKit
- the m.br. book
- open-source kernels
    - Linux, *BSD
    - MINIX 3
    - IncludeOS

- `pdclib`, `libc++`, …

Non-Goals

- writing a realistic kernel
- portability
- long-term maintainability
- hardware / drivers
- file systems
- POSIX

Goals

- learn stuff & have fun
- cross an item off your bucket list

Technical goals (stuff to try)

- something that boots
- memory management basics
- C++ in kernel space
- kernel-user separation

Platform

- protected mode, 32 bit x86
- some assembly required (tm)
- let's not muck with cross toolchains
- GRUB2 as the bootloader
- qemu as the system emulator
- serial port for IO

Part 1: Booting

The Boot Sequence

- very platform-specific
- on x86, either legacy or UEFI
- all sorts of stuff elsewhere
- man-years of work

The Easy Way Out

- GRUB with multiboot2
- not actually portable either :(

Multiboot2

- lands you in protected mode
- getting to C in under 10 instructions
- module preloading
- example in study materials

Checkpoint 1: Part 1

- get a copy of GRUB and build it from source
    - also grab `xorriso` to go with it

- read through the multiboot2 spec
- set up version control for your code
- build the example multiboot kernel
    - `multiboot.tgz` in study materials

- ask questions

Multiboot Modules

- GRUB can load extra files for you
- dump it at some location in memory
- give you a list of the modules
- and their load addresses / sizes

Checkpoint 1: Part 2

- print a list of multiboot modules
- load a text file as a module
- copy the text to screen
- we will use this later to load user programs

Checkpoint 1: Part 3

- write a very simple serial port driver
- `https://wiki.osdev.org/Serial_ports`
- you will need `inb` and `outb`
- do not use interrupt mode (for now)
- this lets us get some user input
- more details about this next week

Assembly Syntax

- immediate values get a `$` prefix
- registers get a `%` prefix
- unprefixed numbers are addresses
- `opcode source, destination`
- note well that there are other conventions

The Calling Convention

- specific to C on x86
- scratch registers: eax, ecx, edx
- return value is in eax

```
mov 4(%esp), %eax // arg 1
mov 8(%esp), %edx // arg 2
// do stuff
ret
```

Sidenote: Calling C Functions

```
pushl   %edx
pushl   %eax
pushl   $fmt
call    printf
addl    $12, %esp // clean up arguments
```

Wrapping I/O Instructions

- read with `inb port, register`
- and write with `outb register, port`
- note the argument order
- note the C argument order on the stack
- maybe draw a picture

Serial Port (RS 232)

- `https://wiki.osdev.org/Serial_ports`
- write `inb` and `outb` in assembly
- so that they can be called from C

- defining symbols in asm: `.global foo`
- `foo` is then a standard label
- don't forget to write a C prototype for both

Part 2: Memory

Kernels vs Memory

- physical memory
- MMU and page tables
- memory protection
- dynamic memory in kernel

MMU

- part of the CPU
- Memory Management Unit
- responsible for memory protection
- also virtual memory

Address Types

- physical – what shows up on the memory bus
    - not directly accessible to (normal) software
    - shows up as frame addresses in page tables

- virtual
    - normal pointers in C
    - user-mode software only sees this
    - managed by the OS

Paging

- physical memory is split into 4K frames
- virtual memory is split into 4K pages
- i.e. page is the content, frame is a place
- pages can be moved in and out of frames

Properties of Pages

- each page is of a fixed and uniform size
- pages have permission bits (read, write, execute)
- page table decides which pages 'exist'
- the page table can be changed by the OS
  - useful for context switching

Aside: Segmentation

- different memory protection scheme
- variable-sized segments
- specific use: code, stack & data segments
- not used in modern systems
- we will not use segmentation either

Page Directory

- first level of paging metadata
- lives at a 4K-aligned physical address
- the address of the PD lives in `CR3`
- lists 1024 pointers to 4K page tables

Page Table

- second level of paging metadata
- also lives at 4K-aligned physical addresses
- lists 1024 physical (frame) addresses
  - the page may or may not be present in the frame
  - the P bit decides this
  - accessing a P-less pages traps

Enabling Paging

- paging must be explicitly enabled
- you need to set up a page directory first
- and the page tables to go with it
- then load the physical address of PD into `CR3`
- and flip the `PG` and `PE` bits in `CR0`

Identity Mapping

- portions of memory can be mapped 1:1
- those virtual addresses will be the same as physical
- this is called identity mapping
- makes your life easier, but limits your flexibility

Reserved Physical Memory

- there are areas you cannot touch
- this includes BIOS data structures
- the PCI address space
- data on this is available from multiboot

Memory Allocation

- there are two levels of allocation in kernels
- one deals with obtaining physical pages
- another deals with fine-grained memory
  - it is hard to live without `malloc()`
  - linked lists, dynamic arrays &c. &c.

Page Allocator

- the page allocator can be quite simple
- page size is uniform
- the memory chunks are fairly big
  - which makes metadata small in comparison
  - there aren’t that many pages to be had

Implementing `malloc()`

- malloc works by subdividing bigger chunks of memory
- userspace `malloc()` typically gets memory from `mmap()`
- you can use the page allocator as a backend for malloc
- alternative: fixed size memory area for kernel data
  - simpler, but also less flexible

How does `malloc` work?

- many different approaches
- often size-bucketed storage for small allocations
  - per-bucket bump allocator
  - per-bucket, inline free lists

- alternative: pre-filled free lists
- passthrough of big allocations (page-sized)

Aside: Optimising `malloc`

- consider cache interaction
- free list used in FIFO or LIFO order?
- separate per-thread arenas/pools
- `free` still has to work cross-thread

Checkpoint 2: Part 1

- set up page tables
- identity-map your kernel
- make those pages supervisor-only
- write code to map/unmap user pages

Checkpoint 2: Hints

- you can implement most of page management in C
- like with `inb`/`outb`, you need asm to flip `cr3`
  - and to change bits in `cr0`

- identity-mapping the kernel will save you a lot of trouble
  - but you can do a bootstrap with physical/virtual split
  - no bonus points for doing this

Checkpoint 2: Part 2

- pick a range of addresses for kernel data
- obtain physical memory reservations at boot time
- write `malloc` for in-kernel use
- also write `free` and `realloc`
- if you feel adventurous, try a threadsafe implementation

Checkpoint 2 Resources

- https://wiki.osdev.org/Paging
- https://wiki.osdev.org/Setting_Up_Paging
- https://wiki.osdev.org/Page_Frame_Allocation
- https://wiki.osdev.org/Memory_Allocation
- x86 reference manual

Part 3: `libc` &c.

What is `libc`

- provides ISO C library functions
  - `printf`, `scanf`, `strcmp`, ...
  - `malloc`, `free`, ...
- and the POSIX syscall interface
  - `open`, `read`, `write`

Using `libc` in a Kernel

- no system call interface
- reduced file abstraction
- `malloc` never fails?
- what about thread support?

Support for `FILE`

- this includes `printf` and friends
- it makes sense to tie this to console
  - in our case, serial port

- `FILE` does not need much
  - only a few callbacks

Kernel Threads

- `libc` may contain `pthread` support
- this is very much user-level
- probably a bad idea to use this API in kernel
- kernels still need mutexes and the like

Porting `libc`

- memory allocation (`malloc`)
  - we did this last time

- file abstraction (`FILE *`)
- random platform glue
  - `exit`, `atexit`, `sleep`, …

Porting `libc++`

- based mostly on `libc`
- and `pthread` support code
- also needs `libc++abi`
  - RTTI, exceptions, ...

Thread Support

- our kernel will be single-threaded
- we still need to provide thread APIs
    - `libc++` needs a rudimentary one

- mutex functions can do nothing
- `pthread_once` (equivalent) has to work though

Dependencies Everywhere

- `std::stringstream` is nice to have
- but it needs a `locale` library
    - we need to provide locale stubs for `libc++`
- normal streams are based on `FILE *`

Checkpoint 3

- take a `libc` of your choosing
    - `pdclib` would be a good candidate

- make it build and run
- adapt it for kernel use
- tie `stdout` and `stdin` to the serial port
- `printf` away

Part 4: System Calls

What is a System Call

- calls from user code into the kernel
- works (almost) like a function call
  - with a special calling convention

- switches the CPU into privileged mode

How?

- software interrupts
  - synchronous
  - saves CPU state

- `sysenter` or `syscall` (on x64)
- return with `iret`, `sysleave` or `sysret`

Software Interrupts

- user side: an `int` instruction
  - you get to pick a number (from 32 up)

- kernel side: IDT
  - interrupt descriptor table
  - address stored in `idtr`
  - load with `lidt`

Loading IDT (and GDT)

- `lidt` and `lgdt` expect both size and address
    - this is given as a pointer to a 2-tuple

- the address is a virtual address

IDT Structure

- another table a bit like the page directory
    - or like GDT and LDT (which we don't use)
    - oops, IDT refers to GDT or LDT

- see also `https://wiki.osdev.org/IDT`
- set all but the system call P (present) bits to 0

IDT Entry

- contains a code reference (segment + offset)
  - segment really means a GDT selector
  - you will want this to be a TSS

- and a few control bits / type info

TSS

- task state segment
- used for hardware-assisted context switching
- also needed for ring 3 → ring 0 transition
- you only need to set `ss0` and `esp0`
  - and set `iopb` to 104 (since we won't use the bitmap)

User Side

- the exact sequence is up to you
- you want to send syscall number somehow
  - eax is customary

- you want to send in arguments too
  - probably mostly via stack

User Side in C

- you will probably want a `syscall` function
- implement it in assembly
- needs to cooperate with the kernel side

Checkpoint 4

- implement a system call interface
- testing will be tricky without userland
- but you can do `int` in kernel
    - you won't be able to check ring transitions
    - all else should work like normal

Part 5: Userland

Checkpoint 5

- build a userland version of `libc`
- build a user program that uses `printf`
  - turn it into a multiboot module and load at boot

- prepare memory (including stack) for the program
- execute the program in ring 3

Userland `libc`

- mostly the same as kernel `libc`
- link it statically into your program
- don't forget the syscall mechanism
- hook up file ops into syscalls

Linking

- write a link script to link the program
- you can use a fixed load address
    - feel free to experiment with PIC/PIE

- the linker will produce an ELF binary

Multiboot Module

- you can use a separate module for each section
  - you'll probably need text and data

- you can use `objdump` to extract the sections
- it's also OK to keep & use ELF metadata instead

Loading

- GRUB will load your modules wherever
- set up page tables for userspace
- map the module data on the right virtual addresses
  - either those agreed ahead of time
  - or those parsed out of the ELF header

Switching to User Mode

- you will need to do an `iret`
  - even though no interrupt happened

- set up a stack as if an interrupt just happened
- then do an `iret` into the user mode
- see also `https://is.muni.cz/go/ki6k82`

A Few Hints

- user mode, stack setup and loading are independent
- you can switch into ring 3 within the kernel
- you can create another stack within the kernel too
- you can load (and execute) program without user mode

Bonus: Cooperative Multitasking

- allow 2 (different) programs to be loaded
- add a 'yield' system call
- let the two tasks alternate in execution
- run them in separate address spaces

Part 6: Interrupts

Hardware Interrupts

- hardware can asynchronously signal events
- typically related to input/output
  - new input available
  - finished processing something

- data is moved some other way
  - DMA, PIO (`inb`, `outb`)

Interrupt Enable

- the CPU can mask/unmask interrupts
- on x86, this is controlled by `eflags`
- instructions:
  - `sti` enables interrupts
  - `cli` masks (disables) interrupts
  - `popf` can change the interrupt flag

Interrupt Service Routine (ISR)

- the bit that runs in response to an IRQ
  - also called the top half

- runs on the interrupt stack
- ends with an `iret`
  - chances are the `iret` lands in user mode

Re-entry

- ISRs are concurrent to the rest of the kernel
- if the ISR calls into the rest of the kernel
  - the same function may already be executing
  - similar to POSIX signal handlers

- mutual exclusion will not help

Prohibiting Nesting

- the easiest way is to `cli`
- this masks all (maskable) interrupts
- do not forget to `sti` before `iret`
- this is the easiest (not best) approach

Nested Interrupts

- an interrupt can arrive while an ISR is running
- those are nested interrupts
- in this case, more reentrancy is required
- also, the interrupt stack is finite

Fully Re-entrant ISR

- worst case if the same ISR runs nested
    - only applies to the ‘top half’
    - bottom halves run from a queue

- for example, this is forbidden in Linux
    - but different ISRs can nest on the same CPU

IRQ: Interrupt ReQuest

- the hardware side of interrupts
- (TBD)

PIC

- Programmable Interrupt Controller
- you need to set this up to get IRQs
- IRQs are mapped to interrupts
- `https://wiki.osdev.org/PIC`

Checkpoint 6

- write an IRQ-driven serial port driver
- IDT principles stay the same as with syscalls