Systèmes et Réseaux: Projet

24 Mai 2018

POIVRE OS

AZIZIAN Waïss, ROBIN David

1 Introduction

This document describes the project POIVRE OS, presented as the final project for course "Systèmes et Réseaux" by Waïss Azizian and David Robin.

The full source code of the project is available at https://gitlab.com/ar-os/kernel/

All source code was written by W.Azizian and D.Robin for this project, unless explicitely stated with the proper copyright notices.

2 Setup

2.1 Development Platforms

The development process was initiated with Qemu, and the code was later ported to Hardware on the Raspberry Pi 3 (ARM 64 bits). From then on, two branches where maintained through Git, one for the hardware-compatible code, and one ahead of it, with the cutting-edge changes that were only tested in Qemu.

Most of the codebase is common to the two versions. However, some details vary, mainly the interface with the special registers mapped to hardware pins. Both were maintained through use of define instructions, that were triggered directly by the makefile, able to generate two versions (for virtual of physical support).

2.2 Uart communication

We chose not to support any graphical interface, because the documentation was not available, which would have made this part a lot more difficult, although not as interesting as other aspects of the operating system.

Communication with the user is thus achieved with UART pins, connected with a serial cable to a USB port on a Linux system that is able to send keyboard input to the pins in ascii encoding and retrieve information from output pins to print it on the screen.

This emulates the use of a keyboard and screen (with only text support) directly with the Raspberry Pi board, without requiring us to support any USB or hardware-specific protocols.

2.3 Forcing GDB and QEMU to cooperate

Debugging a user application is already a tedious process, but debugging an operating system is an even greater challenge. To allow a minimum of non python-style print-based debugging, we tested our software on Qemu with specific build flags to create a local socket on which to listen for debugging instructions. We were then able to point GDB to that socket and thus get a line-by-line execution and access to register's content in real time.

The result was far from perfect and there were several bugs with the communication between Qemu and GDB that we were not able to fix, for instance GDB was not able to correctly guess the line of the C code it was actually executing. Nonetheless, this was a lot easier than dumping all register's content every now and then.

2.4 Makefile

The use of proper make targets helped us speed up the development. We used a target for each build type (Qemu or hardware), one to start GDB, one to deploy the kernel to an SD card, one to connect the pins to the USB port, etc.

Everything was then put together with a configure script, that allowed the use of a fixed log level and custom binaries for compiling and linking, preserved across builds.

2.5 Logging levels

The end user seldom cares about what happens in the background when his machine is booting up, but as developers, we wanted to have a lot more details on what was happening and what had gone wrong.

In order to allow both modes, we chose to split our debugging output into 6 categories (highest to lowest) : VERBOSE, DEBUG, INFO, WARNING, ERROR, and WTF (stands for "What a terrible failure").

Choosing a log level (at compile time) shows all messages reported with this log level and all lower levels. For instance, choosing the log level DEBUG would print all messages relative to log levels DEBUG, INFO, WARNING, ERROR and WTF, but not VERBOSE

2.6 Specific logging

Some very specific parts of the OS needed a log so heavy that it rendered the whole log unreadable. Heavy dumps of the memory allocation structure or the process's states during scheduling fall into this category. This was far beyond what was acceptable for the VERBOSE level, and so they were moved to specific logging levels. Those levels require the use of the VERBOSE main level, and extra build arguments (MALLOC_VERBOSE, PROC_VERBOSE, FILESYSTEM_VERBOSE, etc.) .

2.7 ARMv8 Vocabulary

In ARMv8 there are four execution levels (EL), from EL0 to EL3.
EL3 and EL2 are respectively hypervisor and secure mode that we did not use. EL1 is Kernel mode and EL0 corresponds to applications.

2.8 Official documentation

The Raspberry Pi 3 implements an ARMv8-A instruction set. We were lucky to have the correspondingArchitecture Reference Manual which was our primary source of technical documentation. We tried to refer to it as much as possible in the code. Thus a comment `ARM ARM xxxx` refers you to this manual at page `xxxx`. We also occasionally refer to the ARMv8-A Programmer Guide, ARMv8-A Address Translation and AArch64 Exception Handling. One of our issue was that the Broadcom documentation for our board (BCM2837) was not available. We had to make do with the one of the BCM2835 because as the rumor goes it is quite the same (except the addresses...).

3 Ecosystem

3.1 A micro-kernel

[diagram] Our aim was to realize a micr-kernel. This has driven the conception of the process structure. Some functionnalities are thus being provided by user processes instead of the kernel.

- In ARMv8, by default the execution which powers down the machine cannot be executed at EL0. Only the `init` process can in our system. Thus to shutdown (`libk/sys.c`) a process kindly asks the `init` process.
- The majority of the memory allocator is a library that do not need special rights so it is executed at EL0 along with the same process. But to free, it needs to access the translation tables. It is thus done through a communication with the memory manager process.
- We also created a process handling interactions with the Uart, so that ultimately a process do not need to access peripherals.
- Finally, interactions with the file system can entirely be done by communicating with a dedicated process.

  See the corresponding sections for more details.

3.2 Project structure

The code is in the directory `src` which contains the files `linker.ld`, our linker script and `boot.s` the assembly overviewing the boot process and the following directories:

- `memory`: all code related to the MMU and memory allocation.
- `proc`: implementation of processes.
- `interrupt`: the interrupt vector tables for the processor, handlers and a small timer library.
- `libk`: this should actually be called `lib`, it contains all the libraries, both kernel and user.
- `usr`: the `main` functions of our dedicated processes, i.e. of `init`, the memory manager process, the I/O manager process and the file system manager process (and also the process launching tests).
- `test`: our test code.

The `doc` contains a few pages of documentation we made (`mmu.md` and `proc.md`). Note that the `issues.txt` and `bugs.txt` are highly incomplete.

4 Boot

The boot process is as follows :

- Halt all cores except core 0
- Zero-out .bss segment
- Initialize interrupt tables
- Initialize uart variables (to allow logging)
- Initialize MMU tables (identity mapping)
- Initialize caches
- Invalidate remaining unused entries in the MMU tables
- Check the identity paging for errors
- Enable the MMU if nothing went wrong
- Initialize IRQ and timers for the scheduler
- Initialize process's structures
- Start process "init" with PID 0

5 MMU

The configuration of the MMU was one of our main hurdle as it eventually took weeks.

5.1 linker

The linker script plays a foremost role in the configuration of the MMU, as the variables defined there are used to delimit zones with different access rights and cache policies.

Some space is allocated for the stack at position 0x8000. The kernel code is mapped right after that offset. The filesystem is then inserted in memory (see corresponding section), followed by the interrupt tables and enough space for the memory allocation tables.

5.2 Memory map

We chose a very simple approach. We use an identity mapping for everything that appears in the linker script: the code, the data segments, the MMU tables...

5.3 Page allocation

All unused pages are not mapped at first, but when the kernel tries to access it, a translation fault is trigger. The handler checks where the fault happened, and if no segfault or heap overflow is triggered, then the page is automatically allocated to the kernel by extending the identity mapping, and the kernel resumes execution seamlessly as if the page had already been his to use.

6 Process

The implementatation of processes lies in `src/proc`. We started from the specifications of the micro-kernel given during the class but we ended up doing several things differently.

6.1 System state

The system state is conserved in a structure defined in `proc.h`, `sys_state`. It contains all the necessarry information about processes, including the MMU and memory allocation configuration, inter-process configuration...

6.2 Scheduling

All processes are preempted by default (though it can be specified otherwise in the `proc_descriptor`. For this timers are used. They are implemented in `interrupt/timer.c` and use the memory mapped interface for peripherals (thsu covered by Broadcom documentation and not ARM). When a process is run, a countdown is started. The timer will then generates an IRQ interrupt when time has ran out and the scheduler will be called again.
The scheduler is implemented in `proc.c`. The scheduling strategy is based on priorities, from 0 (lowest) to 15 (highest). The scheduler first chooses a priority as follows:

- If there is a runnable process of priority 15, choose this priority.
- If all the runnable processes are of priority zero, choose priority 0.
- Otherwise pick a random priority between those with a runnable process with probability proportional to $p^2 + n$ where p is the priority and n the number of runnable processes in this priority.

At each priority is associated a cyclical doubly-linked list containing the runnable processes of this category with a pointer to process in this list. When the priority is chosen, the first one, i.e. the one designated by this pointer is run and the pointer is moved to the next one. Thus for a given priority the processes are run in a round-robin manner.
The quantum of time given to a process when it is run is presently one second as it made debugging easier. When the execution of a process is stopped for another reason than a timer interrupt, the amount of time left is backed up and restored when it is run again.

6.3 Context switching

The context switch is quite a tricky operation. The context swicth after an interrupt, as described below, is done in `interrupt/idt_el1.S`, the inetrrupt tables of EL1, in the functions `lower_el_el1_sync_handler` and `el1_irq_handler`. After an interruption coming from a process at EL0, we are at EL1, still with the process' address space but with the stack pointer of EL1. So the first thing to do is switch to the stack pointer from EL0 to avoid messing with the process' stack. Then a C function (`save_context`) is called to back up everything in the `sys_state` structure. And only now we can switch back to the stack pointer from EL1 and to kernel's address space. Moreover some system calls require to read a few bytes into an arbitrary location in the process' memory. This is done during the saving of the context.
The functions to restore context are in `proc_mmu.c` and `proc_asm.s` and quite the smae is done in reverse.

6.4 Process states and syscalls

A process can be in eight different states, defined in `proc.h`: Kernel, Free, Waiting, Zombie, Runnable, Wait_Listener, Sending_ch, Listening_ch. The last three will be discussed in the next section. Kernel a

special state for the kernel (which has PID zero though not actually being a process). The other ones are the same as in the specification of the micro-kernel.
There are 6 system calls, implemented in `proc.c` and defined in `libk/sys.c`. `fork, wait` and `exit` do what we expect them to do, except that `exit` returns an `err_t` structure instead of an `int` (they are also documented in `doc/proc.md`). Note the additional presence of a seventh, the system call 101, which is used in tests: upon such a syscall the scheduler just resumes the execution of the process.

6.5 Forking and address space

When a process is forked, the child has the same address space as its parent, with the same content in the allocated space. To avoid a costly operation, pages allocated by the parent are not immediately copied during the `fork`. The same physical pages are mapped into the child's address space but these shared pages are marked as Read and execute only in both the child's and the parent's MMU tables (done in `process_init_copy_and_write` in `memory/mmu.c`). Only the stack is copied directly (`copy_forked_page`).
Each physical page has an associated counter which keeps track of the number of virtual pages pointing to it. Thus when a process tries to write in a shared page, a new physical page is allocated and the content is copied if it is not the only owner (see `permission_fault_handler` in`memory/mmu.c`).

6.6 Inter-process communication

We designed our own inter-process communication system, which enables the user to small amount of data (less than 1kB) seamlessly.
The first system call is `receive` (see `libk/sys.h`). After a call to this, the process passes in Listening_ch state.
Then a sender calls `send` which sends some data to the target process. (The last argument enables the user to choose whether to wait or not if the target is not in listening state). This data is copied to the place in memory specified by the receiver. The sender now passes in Sending_ch state, awaiting an acknowledge signal. The receiver now becomes Runnable, and can start preparing a response for the sender. When it is done, it sends it back to the sender with `acknowledge`, releasing the sender.

6.7 Services

This communication is sucessfully used in some services. They are started in `usr/init.c`. The client-side code is in `libk` while the "service-side" is in `usr`.

- The IO manager interacts with the Uart on behalf of other processes. Its code is in `usr/io.c` and the library in `usr/io_lib.c`.
- The file system manager handles file system operations. Its code is in `usr/fs_manager.c` and the library in `usr/user_filesystem.c`.
- The memory manager only intervenes for the freeing of memory, implemented in `usr/mem_manager.c`.

7 Malloc

7.1 General

Dynamic memory allocation is achieved through a set of functions located in `src/memory`. The algorithm used to allocate memory is pretty simple : every time a process makes a request for memory, malloc finds a segment of unused memory, stores a header in the first few bytes, and returns a pointer to the first usable byte inside the block. When the process frees the memory, the block can be marked as unused, and can be reallocated later on.

7.2 Free block management

In order to efficiently retrieve an unused block when a process needs memory, the use of a proper data structure is required. We chose to implement a doubly linked list of blocks. Each block points to the previous and next block in the list, and a boolean value indicating if it is free or in use by some process. Finding a free block can thus be achieved in linear time.

When no more blocks are available, malloc calls sbrk to allocate more space on the heap, and then uses that space to create more blocks.

However, this introduces a lot of fragmentation in the memory. To reduce this effect, we implemented a mechanism of merges and splits : when to consecutive blocks are free, they can be merged into one big free block. Then, when a segment of size x is required, and a block of size $y > x$ is available, it can be split into a block of size x that can be returned, and a block of size $y - x$ to be used later.

In order to reduce fragmentation even further, we chose to allocate only blocks of size a power of two. FreeBSD's benchmarks show that this makes it less likely that "unusable" blocks proliferate, by "standardizing" the block's size (i.e. fewer blocks are either too small to be used or split into tiny unusable blocks).

8 Filesystem

8.1 SFS0

We chose to support a rather simple filesystem, yet very similar to the ext filesystem's family, called SFS0. It was presented in one of the classes of the course, and we chose to support it as-is, without any modifications.

It consists of a superblock containing all constants needed to operate the filesystem, and pointers to the last free block and last free inode, which themselves are pointers forming a linked list of free blocks/inodes.

8.2 Hardware

Our goal was at first to implement ext2, to be able to mount the SD card on which the kernel is located. However, documentation for the Raspberry Pi 3 is not available, and the only known operating systems to operate the SD Card driver are based on the linux kernel, which itself relies on ARM for providing the hardware-specific files. This files are thus completely undocumented, and we were not able to get enough information on the interface between the processor and the SD Card to get it to work in time. That is why we use a filesystem that only resides in RAM. Nonetheless, we tried to emulate the way an OS should interact with a regular hardware storage (that is, only reading and writing in blocks, modifying in memory and only then writing back, etc.).