An educational kernel for the Raspberry Pi

CS310 - Presentation

Thomas Archbold

March 4, 2019

Introduction

Operating systems are some of the most pervasive pieces of software around, but also some of the most complex

The introduction of the Raspberry Pi has made computers much more accessible - allows and encourages experimenting at all levels

There are several official operating systems for the Pi - NOOBS, Raspbian, Windows IoT core, PiNET - but none provide a resource to learn about the operating system itself

Objectives

Main goal: to write a configurable operating system for the Raspberry Pi capable on booting on real hardware for educational and hobbyist use.

Specifically, this involves:

- providing different approaches for tasks such as scheduling, interprocess communication, and permanent storage
- allowing the user to configure the system at compile time to use different combinations of these approaches, and
- exposing a simple and easily extensible interface for additional features

Background material

- Pintos - Stanford's instructional x86 operating system, used to teach their CS140 course
- Baking Pi - Cambridge's tutorial on writing an operating system for the Raspberry Pi in assembly
- MINIX - Tanenbaum and Woohull's illustrative operating system for "Operating Systems: Design and Implementation"
- The little book about OS development - Helin and Renberg's "practical guide to writing your own x86 operating system"
- `osdev.org/` - community of hobbyist operating system developers, containing information, tutorials, advice, etc.

Why is this project worthwhile?

It is more difficult to get into low-level/systems programming
$\Rightarrow$ focus is on clear code to aid understanding

The project attempts to demystify aspects of operating system development - able to see the theory in practise

It will provide an accessible platform to further tinker and experiment with operating system development, with little to lose

Useful concepts - compilation

Operating System - program that manages a computer's hardware

Freestanding environment - little access to C standard library, and program entry point not necessarily at `main()`
⇒ must implement most of standard library ourselves

Cross-compiler - allows us to compile code that will run on the target architecture from our own machine

Linker - responsible for linking all compiled `.o` files into one executable
Defines the following sections:

- `.text` - executable code
- `.rodata` read-only data (i.e. global constants)
- `.data` - global variables that are itialised at compile-time
- `.bss` - uninitialised global variables

Useful concepts - general

Kernel - the core of the operating system; the one program which is running at all times throughout execution

Exception - an event triggered when something exceptional happens during normal execution (hardware giving CPU data, privileged action, bad instruction)

Process - a program that has been loaded into memory and is executing

Concurrency - the ability for multiple processes to make progress seemingly simultaneously, as a result of some scheduling algorithm

Context Switch - switching execution to another process. Involves:

1. saving the state of the currently executing process (**state save**)
2. loading the saved state of a different process (**state restore**)

Useful concepts - contd.

Synchronisation - the prevention of **race conditions**: when the outcome of two concurrently executing processes depends on the order in which data access took place

Interprocess Communication - mechanism by which cooperating processes may exchange data and information

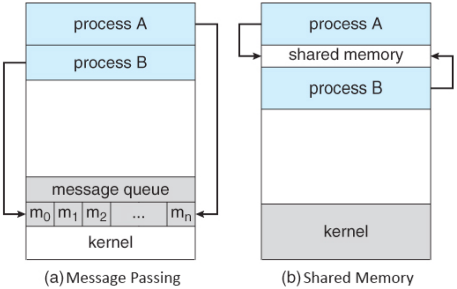

Figure: The two fundamental models of interprocess communication

Tools used

Languages: C and ARM assembly

Cross-compiler toolchain: `arm-none-eabi`

Build automation: Make

Version control: git

Emulation: QEMU

Model: Raspberry Pi 1 Model B+

Raspberry Pi 1 Model B+

Hardware

- System-on-Chip (SoC): BCM2835
- CPU: 700MHz ARM1176JZF-6
- GPU: 250MHz Broadcom VideoCore IV
- Memory: 512MiB
- USB: 4x USB 2.0 ports
- Video output: HDMI
- Peripherals: 40 GPIO pins, UART
- Storage: MicroSD card

Raspberry Pi 1 Model B+

Hardware

- System-on-Chip (SoC): BCM2835
- CPU: 700MHz ARM1176JZF-6
- GPU: 250MHz Broadcom VideoCore IV
- Memory: 512MiB
- USB: 4x USB 2.0 ports
- Video output: HDMI
- Peripherals: 40 GPIO pins, UART
- Storage: MicroSD card

Why the Pi in particular?

Raspberry Pi 1 Model B+

Hardware

- System-on-Chip (SoC): BCM2835
- CPU: 700MHz ARM1176JZF-6
- GPU: 250MHz Broadcom VideoCore IV
- Memory: 512MiB
- USB: 4x USB 2.0 ports
- Video output: HDMI
- Peripherals: 40 GPIO pins, UART
- Storage: MicroSD card

Why the Pi in particular?
Simple boot process - handled entirely by SoC

Raspberry Pi 1 Model B+

Hardware

- System-on-Chip (SoC): BCM2835
- CPU: 700MHz ARM1176JZF-6
- GPU: 250MHz Broadcom VideoCore IV
- Memory: 512MiB
- USB: 4x USB 2.0 ports
- Video output: HDMI
- Peripherals: 40 GPIO pins, UART
- Storage: MicroSD card

Why the Pi in particular?
Simple boot process - handled entirely by SoC
Underlying architecture of BCM2835 chip is identical to BCM2836/7

Raspberry Pi 1 Model B+

Hardware

- System-on-Chip (SoC): BCM2835
- CPU: 700MHz ARM1176JZF-6
- GPU: 250MHz Broadcom VideoCore IV
- Memory: 512MiB
- USB: 4x USB 2.0 ports
- Video output: HDMI
- Peripherals: 40 GPIO pins, UART
- Storage: MicroSD card

Why the Pi in particular?

Simple boot process - handled entirely by SoC
Underlying architecture of BCM2835 chip is identical to BCM2836/7
Standard set of hardware ⇒ more widely accessible

Project overview

Main milestones reached:

- Capable of booting in emulated environment
- Capable of booting on real hardware
- Display on real screen through HDMI
- Interrupts
- Processes and Threads
- Concurrency
- Synchronisation

Current goal: interprocess communication

Design overview - boot

Booting

- Bootloading handled by SoC, so just need to set up system and initialise C runtime
- `boot.S` initialises stack pointer at `0x8000`, zeroes `bss` segment, then loads C kernel entry point `kernel_main()` to begin execution

Memory management

- Bootloader creates list of information about the hardware called **atags**
- Each tag consists of a header and tag-specific data
- To find the amount of memory available to the system, simply iterate over list of tags until the `tag == ATAG_MEM`, at which point return `atag_mem.size`

```
enum atag_tag {
    ATAG_NONE = 0x00000000,
    ATAG_CORE = 0x54410001,
    ATAG_MEM  = 0x54410002,
    ...
};
```

```
struct atag_mem {
    uint32_t size;
    uint32_t start;
};
```

Design overview

Printing to screen Interrupts and Exceptions System Timer Processes and Threads Concurrency Synchronisation Keyboard input

Project management

Next steps

Evaluation