An educational kernel for the Raspberry Pi

CS310 - Presentation

Thomas Archbold

March 5, 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 - compilation

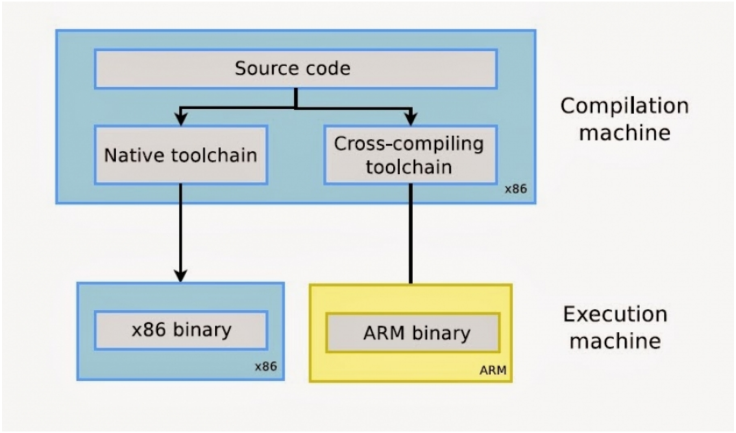


Figure: Cross-compilation

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 - general 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

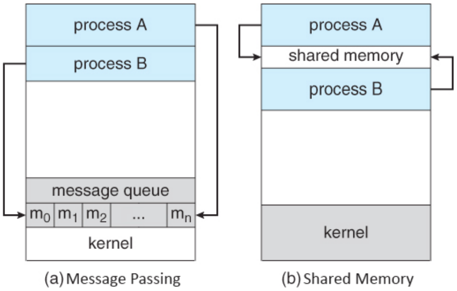


Figure: The two fundamental models of interprocess communication

Useful concepts - Pi specific

Peripheral - a device with a specific address that it may read and write data to and from

- All peripherals may be described by an offset from the Peripheral Base Address (`0x20000000` on the Raspberry Pi 1 Model B+)
- Peripherals include: timers, interrupt controller, GPIO, USB, UART

Memory Mapped I/O - uses same address space to address both memory and I/O devices ⇒ the memory and registers of the I/O devices are mapped to address values

- e.g. to turn the ACT LED on, we simply write a bit at the correct offset: `mmio_write(ACT_GPSET, 1 << ACT_GPBIT)`
- where `mmio_write()` is simply `*(volatile uint32_t *) reg = data;`

Tools used

Languages: C and ARM assembly

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

Build automation: Make

Version control: git

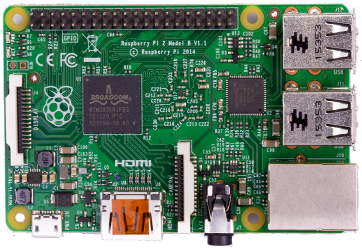
Emulation: QEMU

Model: Raspberry Pi 1 Model B+

Raspberry Pi 1 Model B+

Hardware

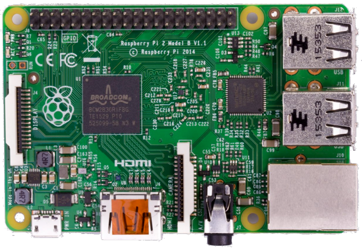
- System-on-Chip (SoC): BCM2835
- CPU: 700MHz ARM1176JZF-6
- GPU: 250MHz Broadcom VideoCore IV
- Memory: 512MiB
- USB: 4x USB 2.0 ports
- Video output: HDMI
- Peripherals: 40 GPIO pins, UART
- Storage: MicroSD card



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

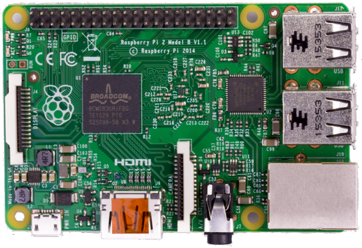


Why the Pi in particular?

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

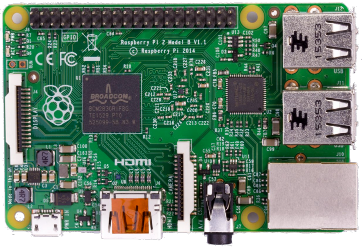


Why the Pi in particular?
Simple boot process - handled entirely by SoC

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?

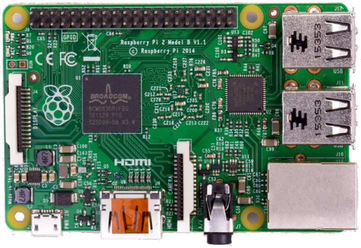
Simple boot process - handled entirely by SoC

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

Project management - organisation

Project has been developed more-or-less with a waterfall-style approach

- rigidity of (early) system lends itself well to this

Project only now beginning to open up more

Term 1:

- reading technical documentation (Technical Reference Manuals, etc.)
- barebones kernel followed by extension
- ended term having written dynamic memory allocator (`kmalloc, kfree`

Term 2:

- HDMI output up to concurrency and synchronisation achieved

Despite progress, somewhat behind schedule

- Certain amount of underestimation - lack of experience

Project management - testing

Fortnightly supervisor meetings held in Term 1 - higher course load

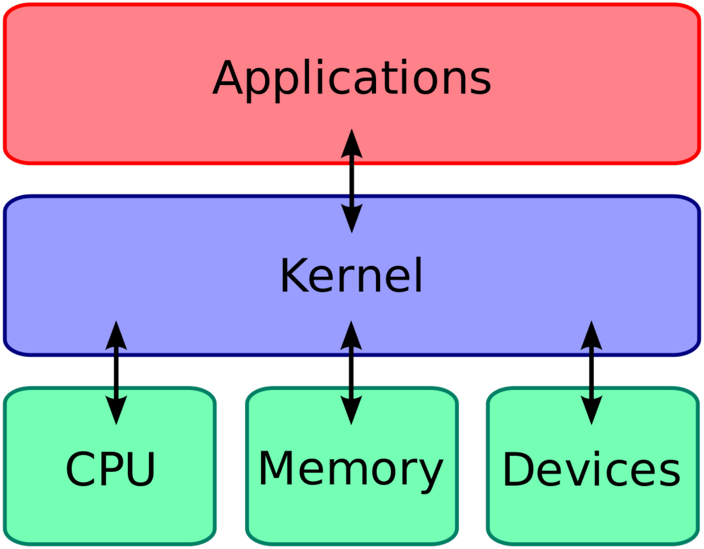
Increased to weekly meetings - focus shifted more towards project

Testing
Almost entirely manual - exists little infrastructure to debug
In most early cases this was done through blinking the ACT LED
$\Rightarrow$ vital to implement `printf()` soon after printing to HDMI

Incremental/waterfall approach has helped in this regard

- once a feature is complete, can be left fairly untouched

Design overview



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 - atags

- 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 - memory

Memory - paging

- Memory split into 4kiB pages
- Each page contains metadata (allocated, kernel page, kernel heap page, shared), metadata for `all_pages` stored just after `__end`
- Maintain two lists: `all_pages` and `free_pages`
- Allocating and freeing pages - simply dequeue/append page to `free_pages` and alter flags

Design overview - memory

Memory - paging

- Memory split into 4kiB pages
- Each page contains metadata (allocated, kernel page, kernel heap page, shared), metadata for `all_pages` stored just after `__end`
- Maintain two lists: `all_pages` and `free_pages`
- Allocating and freeing pages - simply dequeue/append page to `free_pages` and alter flags

Memory - heap

- 1MiB reserved after page metadata for the heap
- Associate each allocation with header and store in linked list
  - allocation size
  - "in-use" flag
- `kmalloc(bytes)` - traverse list to find best-fitting unused segment
- `kfree(ptr)` - unset `allocated` bit and coalesce adjacent segments

Design overview - HDMI output

Printing to screen

Must ask GPU for **framebuffer** - piece of memory shared between CPU and GPU
Process differs slightly between Pi 1 and Pi 2 - both use the mailbox peripheral

- Pi 1 - uses framebuffer mailbox channel
- Pi 2 - uses property mailbox channel (more abstract)

```
struct fb_init {
    uint32_t width;
    uint32_t height;
    uint32_t v_width;
    uint32_t v_height;
    uint32_t bytes;
    uint32_t depth;
    uint32_t ignore_x;
    uint32_t ignore_y;
    void     *pointer;
    uint32_t size;
};}
```

Asking for a framebuffer:

- Set desired framebuffer attributes by initialising `fb_init`
- Send 16-byte aligned structure as message through framebuffer mailbox to GPU
- On success, initialise `fb_info` struct using values from `fb_init`

Design overview - framebuffer

- Pitch - number of bytes on each row of the screen
- Depth - number of bits per pixel

```
struct framebuffer_info {
    uint32_t width;
    uint32_t height;
    uint32_t pitch;
    void     *buffer;
    uint32_t bufsize;
    uint32_t max_col;
    uint32_t max_row;
    uint32_t col;
    uint32_t row;
};
```

Design overview - interrupts

Interrupts and Exceptions
When an exception occurs, a specific address is loaded into the Program Counter
Branch instructions must be written at these locations to branch to correct exception-handling routines

Design overview - interrupts

Interrupts and Exceptions
When an exception occurs, a specific address is loaded into the Program Counter
Branch instructions must be written at these locations to branch to correct exception-handling routines

Address	Exception	Source
0x00	Reset	Hardware reset
0x04	Undefined instruction	Executing garbage instruction
0x08	Software Interrupt (SWI)	Software wants to execute privileged operation
0x0c	Prefetch Abort	Bad memory access of instruction
0x10	Data Abort	Bad memory access of data
0x14	Reserved	Reserved
0x18	Interrupt Request (IRQ)	Hardware telling CPU something
0x1c	Fast Interrupt Request (FIQ)	Piece of hardware can do this faster than all others

Design overview - IRQs

Exception handlers are functions, but not normal ones

IRQ peripheral used to determine device that triggered IRQ

- located at offset `0xb000` from peripheral base address

IRQ peripheral registers:

- Pending - indicate whether a given interrupt has triggered
- Enable - enable certain interrupts by setting appropriate bit
- Disable - disable certain interrupts by setting appropriate bit

Pi has 72 possible IRQs

- 0-63 are shared by CPU and GPU
- 64-71 are specific to CPU

Timer is IRQ 1, USB controller is IRQ 9

Design overview - IRQs

Exception handlers are functions, but not normal ones
⇒ `void __attribute__((interrupt("ABORT"))) reset_handler(void);`

IRQ peripheral used to determine device that triggered IRQ

- located at offset `0xb000` from peripheral base address

IRQ peripheral registers:

- Pending - indicate whether a given interrupt has triggered
- Enable - enable certain interrupts by setting appropriate bit
- Disable - disable certain interrupts by setting appropriate bit

Pi has 72 possible IRQs

- 0-63 are shared by CPU and GPU
- 64-71 are specific to CPU

Timer is IRQ 1, USB controller is IRQ 9

Design overview - System Timer

QEMU does not simulate a system timer, at least for the Raspberry Pi
$\Rightarrow$ at this point it is vital to switch to real hardware

The system timer is a hardware clock which can keep time and generate interrupts after a certain time

- starts when Pi boots and increments 64-bit counter every microsecond
- it is a peripheral that is located at offset `0x3000` from peripheral base

Four registers which the timer compares the low 32 bits with each counter tick

- if any compare register matches the counter, an IRQ is triggered

Timer is set by setting `compare1` register to the current value plus some number of microseconds

Design overview - Processes

Process Control Block (PCB) - structure which holds all of the information about a process

- State save is done simply by pushing all of the registers onto the stack

We maintain two lists:

- job queue - holds all processes in the system
- ready queue - holds all processes residing in main memory and waiting to execute

Creating a new process as simple as allocating space for PCB and process' stack and adding it to the ready queue

- simple interface:
  `void create_kthread(kthreadfn func, char *name, int name_len);`

Design overview - PCB

```
struct proc_state {
    uint32_t r0;
    uint32_t r1;
    uint32_t r2;
    uint32_t r3;
    uint32_t r4;
    uint32_t r5;
    uint32_t r6;
    uint32_t r7;
    uint32_t r8;
    uint32_t r9;
    uint32_t r10;
    uint32_t r11;
    uint32_t cpsr;
    uint32_t sp;
    uint32_t lr;
};
```

```
struct proc {
    struct proc_state *state
    uint32_t pid;
    char name[32];
    void *stack_page;
    DEFINE_LINK(proc);
};
```

Design overview - scheduling

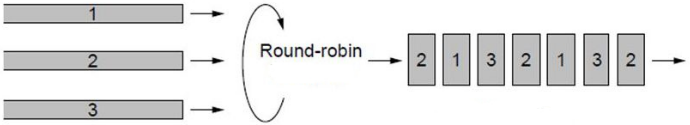
Processes have no consideration for other processes

- if they could they would hog the entire CPU until they are done

⇒ we have to systematically kick them off the CPU

Round Robin

- each process given a set quantum of time to use CPU
- CPU use then given to another waiting process, regardless of process' progress



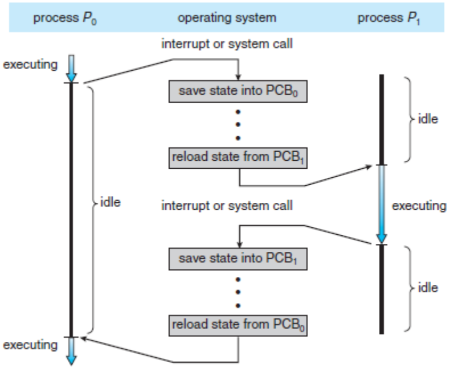
Design overview - context switch

To switch between processes, we must perform a **context switch**

- save process' registers and stack pointer
- load saved stack pointer of next process and pop registers

```
switch_context(old, new):

    str sp, [r0]
    ldr sp, [r1]
```



We also set the timer to go off in another quantum

Design overview - synchronisation

With concurrency, there is the opportunity for race-conditions

We use the concept of an **atomic swap** to implement spinlocks and mutexes - cannot be preempted while trying to take lock

```
if (lock == 1)
    lock = 0;
```

The problem is that it can be preempted while checking `lock` value

- instead, check if we got the lock after taking it

Spinlock - try to acquire lock in loop until successful
Mutex lock - maintain list of processes that want it

Configuration

Configuration done at compile-time

- based on command-line arguments, send different directives
  e.g. `make model=1 sched=roundrobin`
  $\Rightarrow$ `DIRECTIVES = -DRPIBPLUS -DSCHED_ROUNDROBIN`

Once interprocess communication is implemented, this will similarly configured

Next steps

Currently working on:

- keyboard input and shell
- interprocess communication

After this, the final core feature will be permanent storage i.e. filesystems and interacting with microSD card

Evaluation

Overall the project has been success - despite the difficulty in sticking to schedule, much progress has been made

Gained a lot of experience in systems programming

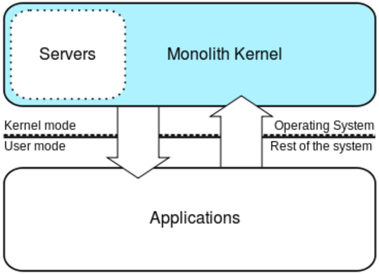
- process (preprocessing, compilation, linking)
- C and ARM competency improved (ability to switch between `.S` and `.c`, more advanced C)
- confidence in Make (conditional compilation, well-structured)
- ability to digest technical documentation (deep knowledge of Pi required)

Evaluation

Focus has somewhat shifted from modularity to educational

- kernel currently monolithic - no user space to speak of
- compile-time configuration emulates modularity to an extent
- truly modular kernel would have been much more complex

Operating system beginning to look like one you might use day-to-day



Questions

Thank you for listening