Build a Minimal Operating System in ARM Cortex-M3
Mini-arm-os

- A minimal multi-tasking OS kernel for ARM from scratch
- From simple to deep, mini-arm-os is a good tutorial to get involved in self-build operating system.
- Including Hello World, Context Switch, Multi-Tasking, Timer Interrupt, Preemptive and Thread.

Let's see how easy an OS could be...
```c
#include <stdint.h>
#include "reg.h"

#define USART_FLAG_TXE ((uint16_t) 0x0080)

int puts(const char *str)
{
    while (*str) {
        while (!(USART2_SR & USART_FLAG_TXE));
        *(USART2_DR) = *str++ & 0xFF;
    }
    return 0;
}

__attribute__((section(". isr_vector")))
uint32_t *isr_vectors[] = {
    0,
    (uint32_t *) reset_handler,    /* code entry */
};

void main(void)
{
    *(RCC_APB2ENR) |= (uint32_t) (0x00000001 | 0x00000004);
    *(RCC_APB1ENR) |= (uint32_t) (0x00002000);

    /* USART2 Configuration */
    *(GPIOA_CRL) = 0x000000B0;
    *(GPIOA_CRH) = 0x44444444;

    *(USART2_CR1) = 0x0000000C;
    *(USART2_CR1) |= 0x2000;

    puts("Hello World!\n");

    while (1);
}
```
From Hello World to Multi-thread

<table>
<thead>
<tr>
<th>Language</th>
<th>files</th>
<th>blank</th>
<th>comment</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>2</td>
<td>9</td>
<td>2</td>
<td>33</td>
</tr>
<tr>
<td>make</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td>C/C++ Header</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td><strong>SUM:</strong></td>
<td><strong>4</strong></td>
<td><strong>19</strong></td>
<td><strong>5</strong></td>
<td><strong>66</strong></td>
</tr>
</tbody>
</table>

Less than 70 lines!!

Even a multi-threading:

<table>
<thead>
<tr>
<th>Language</th>
<th>files</th>
<th>blank</th>
<th>comment</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>4</td>
<td>84</td>
<td>59</td>
<td>339</td>
</tr>
<tr>
<td>C/C++ Header</td>
<td>4</td>
<td>15</td>
<td>5</td>
<td>61</td>
</tr>
<tr>
<td>make</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>21</td>
</tr>
<tr>
<td><strong>SUM:</strong></td>
<td><strong>9</strong></td>
<td><strong>102</strong></td>
<td><strong>64</strong></td>
<td><strong>421</strong></td>
</tr>
</tbody>
</table>

Also less than 500 lines
● Describe how the sections in the input files should be mapped into the output file, and to control the memory layout of the output file

● We bind entry point address in hardware specific address

● So boot loader can know where to enter the startup

ENTRY(reset_handler)

MEMORY
{
  FLASH (rx) :
    ORIGIN = 0x00000000,
    LENGTH = 128K
}

SECTIONS
{
  .text :
  {
    KEEP(*(.isr_vector))
    *(.text)
  } >FLASH
}
Interrupt vector table

- A table of interrupt vectors (IVT) that associates an interrupt handler with an interrupt request in a machine specific way.
- Each entry of the IVT is the address of an interrupt service routine (ISR).
- In ARM, It contains the entry point, initial stack pointer and different kinds of exception handlers

```c
__attribute__((section(".isr_vector")))
uint32_t *isr_vectors[] = {
    (uint32_t *) &_estack,   /* stack pointer */
    (uint32_t *) reset_handler,  /* code entry point */
    (uint32_t *) nmi_handler,    /* NMI handler */
    (uint32_t *) hardfault_handler /* hard fault handler */
};
```
Hello world

Now we've set memory mapping and had an entry point

So now you can:

- Initial everything your own system and hardware need.

- And do what you want in your kernel!!
Multi-Tasking

From Kernel to Task: activate
From Task to Kernel: SVC
privilege levels

● Unprivileged:
  - has limited access to some instructions (MRS, etc.)
  - cannot access the system timer, NVIC, or system control block
  - might have restricted access to memory or peripherals.

● Privileged:
  - Full access
Processor mode

- **Thread mode**
  - The processor enters Thread mode when it comes out of reset
  - Used to execute application software
  - Privilege or Unprivileged

- **Handler mode**
  - Used to handle exceptions
  - Only enter when exception happen
  - Privilege
Stack pointer

- **MSP** : Main Stack Pointer
  - When program start, It's the reset value

- **PSP** : Process Stack Pointer
  - Each task can have its own stack pointer, it's useful for multi-threading
The processor core registers are:

- **Low registers**
  - R0
  - R1
  - R2
  - R3
  - R4
  - R5
  - R6
  - R7
  - R8
  - R9

- **High registers**
  - R10
  - R11
  - R12

- **Stack Pointer**
  - SP (R13)

- **Link Register**
  - LR (R14)

- **Program Counter**
  - PC (R15)

- **Special registers**
  - PSR
  - PRIMASK
  - FAULTMASK
  - BASEPRI
  - CONTROL

- **General-purpose registers**
  - PSP†
  - MSP††

†Banked version of SP

Program status register
Exception mask registers
CONTROL register
activate:

/* save kernel state */
mrs ip, psr
push {r4, r5, r6, r7, r8, r9, r10, r11, ip, lr}
/* switch to process stack */
msr psp, r0
mov r0, #3
msr control, r0
isb
/* load user state */
pop {r4, r5, r6, r7, r8, r9, r10, r11, lr}

/* jump to user task */
bx lr
Context-switch by exception

- **SVCall:**
  - A supervisor call (SVC), applications can use SVC instructions to access OS kernel functions and device drivers.

- **SysTick**
  - A SysTick exception is an exception the system timer generates when it reaches zero.
    - SysTick Control and Status
    - SysTick Reload Value
    - SysTick Current Value
    - SysTick Calibration Value
Activate vs Exception

They are both use as context-switch in mini-arm-os

What's different?

svc_handler:

systick_handler:

/* save user state */
mrs r0, psp
stmdb r0!, {r4, r5, r6, r7, r8, r9, r10, r11, lr}

/* load kernel state */
pop {r4, r5, r6, r7, r8, r9, r10, r11, ip, lr}
msr psr_nzcvq, ip
bx lr
activate:
    /* save kernel state */
mrs ip, psr
push {r4, r5, r6, r7, r8, r9, r10, r11, ip, lr}
    /* switch to process stack */
msr psp, r0
mov r0, #3
msr control, r0
isb
    /* load user state */
pop {r4, r5, r6, r7, r8, r9, r10, r11, lr}
    /* jump to user task */
    bx lr

svc_handler:
systick_handler:
    /* save user state */
mrs r0, psp
stmdb r0!, {r4, r5, r6, r7, r8, r9, r10, r11, lr}
    /* load kernel state */
pop {r4, r5, r6, r7, r8, r9, r10, r11, lr}
msr psr_nzcvq, ip
bx lr
Exception entry and return

- In Activate, we save context and change to PSP in ourself
- But in exception, cortex-M3 provide specific exception handling:
  - Exception entry
  - Exception return
Exception entry and return

- **Exception entry**
  - Save xPSR, PC, LR, R12(ip), R3, R2, R1, R0 automatically
  - Change to handler mode and MSP
  - Happen in Thread mode, or exception has higher priority

- **Exception return**
  - Tell system that exception is complete
  - According to which special address (EXC_RETURN) was set to PC, system will do switch to specific mode
<table>
<thead>
<tr>
<th>EXC_RETURN</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0FFFFFFF1</td>
<td>Return to Handler mode. Exception return gets state from the main stack. Execution uses MSP after return.</td>
</tr>
<tr>
<td>0x0FFFFFFF9</td>
<td>Return to Thread mode. Exception Return get state from the main stack. Execution uses MSP after return.</td>
</tr>
<tr>
<td>0x0FFFFFFFD</td>
<td>Return to Thread mode. Exception return gets state from the process stack. Execution uses PSP after return.</td>
</tr>
</tbody>
</table>
Handler mode and MSP now (automatically)

Save previous state (state before exception)

The state we saved in activate is now pop out
Mini-arm-os

Hello world

Context switch

Multi-Thread

?
Rtenv+
rtenv+

- rtenv is a small Real-time operating system (RTOS) based on Cortex-M3, used for education
- All source files are written by NCKU students
- Its context-switch mechanism is similar to mini-arm-os, but make more progress with PendSV
- Able to run on real hardware (STM32F429i-discovery)
- Able to write user own application like FreeRTOS
microkernel

Messages sent from client to server
- Scheduling:
  - Multi-tasking with priority + round-robin

- Task:
  - ready queue, event queue and three basic states: ready - running – blocked

- Communication:
  - Pipe, block, message queue, register file

- File system:
  - Romfs, read/write by block
Better context-switch

PendSV

- Pended System Call (PendSV) is an interrupt-driven request for system-level service.
- In an OS environment, use PendSV for context switching when no other exception is active.
- Lowest priority
Without PendSV, context-switch usually happened in Systick or SVCall:

When one exception preempts another, the exceptions are nested, and something wrong might be happened.
PendSV

IRQ1 → ISR1 → task1
IRQ2 → ISR2
ISR2 → ISR1
ISR1 → PendSV
PendSV → task2
build a new feature

- PSE51:
  - Minimal Real-time System Profile IEEE Std 1003.13 'PSE51'
  - This profile is intended for embedded systems, with a single multi-threaded process, no file system, no user and group support and only selected options from IEEE Std 1003.1b-1993.
Building Pthread

- Understand POSIX standard for each Pthread API
- Understand what system you are building in
- Use posixtestsuit to make sure your Pthread's behavior is correct
- Trial and error
Now rtenev+ finished...

- pthread_create
- pthread_cancel/exit
- pthread_attr_* (not all)
- Signal.h

Still working~~~
GDB helps build an OS

- Trace register by layout reg
- 'x' to see value in address
- Dprintf : combines a breakpoint with formatted printing
- Backtrace (bt) is useless for exception
- Break, commands and end
Conclusion

- From simple to deep, everyone can Build Your Own Operating System
Reference

- IVT: https://en.wikipedia.org/wiki/Interrupt_vector_table
- Linker script: https://sourceware.org/binutils/docs/ld/Scripts.html
- Processor mode and privilege levels for software execution:
- Core Register:
- Exception entry and return:
- Exception type:
● Mini-arm-os
  https://github.com/jserv/mini-arm-os
● Rtenv+
  http://wiki.csie.ncku.edu.tw/embedded/rtenv
● PSE51
  http://www.opengroup.org/austin/papers/wp-apis.txt
● POSIX
  http://pubs.opengroup.org/onlinepubs/9699919799/