# 程式人《十分鐘系列》

本文衍生自維基百科
CC BY SA

## 用十分鐘

## 向 jserv 學習作業系統設計

陳鍾誠

2016 年 9 月 21 日

# 我是金門大學資工系的老師

# 這學期

- 有組專題學生說要寫作業系統！

# 好樣的！

# 那當然好啊！

# 但是學生說

- 我們先各自看《怎麼寫作業系統》的書，看完之後再開始寫！

# 我想想

# 感覺不太妙！

# 等你們看完

# 搞不好這學期就過完了！

# 到時候

# 你們被當了！

# 我也過意不去！

# 所以

# 我建議學生們

# 直接從 jserv 的 700 行系列開始

# 看看高手怎麼寫

# 先瞭解怎麼寫出第一個

# 然後再來寫自己的作業系統

# 這樣感覺比較可行！

# 現在

- 就讓我們來看看！

# Jserv 的 700 行系列吧！

## jserv 700 行系列

| 主題 | 專案 | 資源 |
|---|---|---|
| Rubi JIT compiler | https://github.com/embedded2015/rubi | 文件 |
| Mini ARM OS | https://github.com/embedded2015/mini-arm-os | |
| Ray Tracing | https://github.com/embedded2016/raytracing | |
| 3D renderer | https://github.com/jserv/stm32f429-r3d | |
| Dalvik Virtual Machine | https://github.com/jserv/simple-dvm | |
| AMaCC = Another Mini ARM C Compiler | https://github.com/jserv/amacc | |

# 我曾經研究過

- 這系列中的 Rubi JIT compiler

# 寫得很棒！

# 又清楚

- 又簡短
- 執行速度又超快！

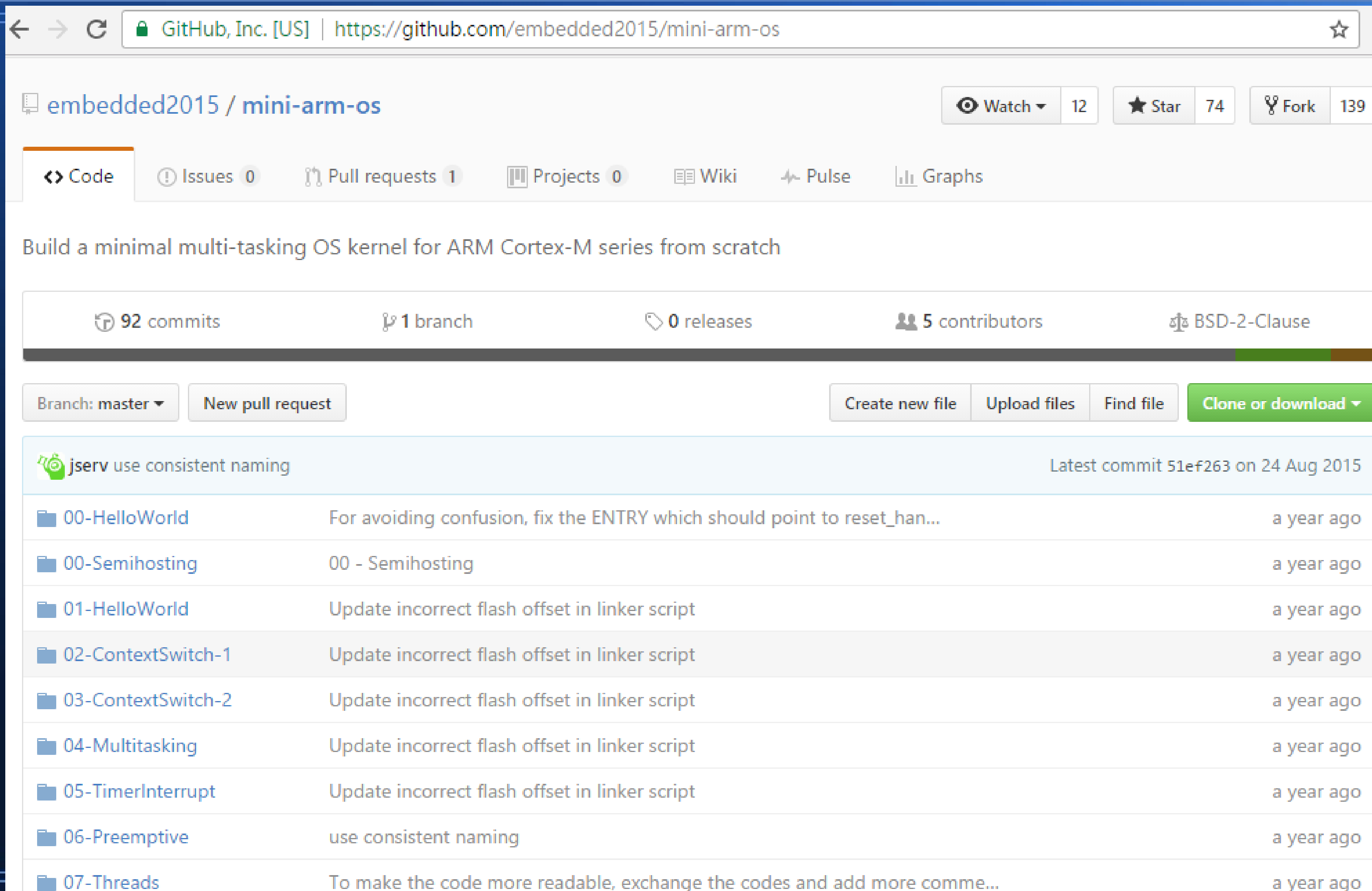
# 現在要研究作業系統

# 我想從 jserv 的 700 行系列開始

- 應該是個不錯的選擇！

# 我們研究的標的物

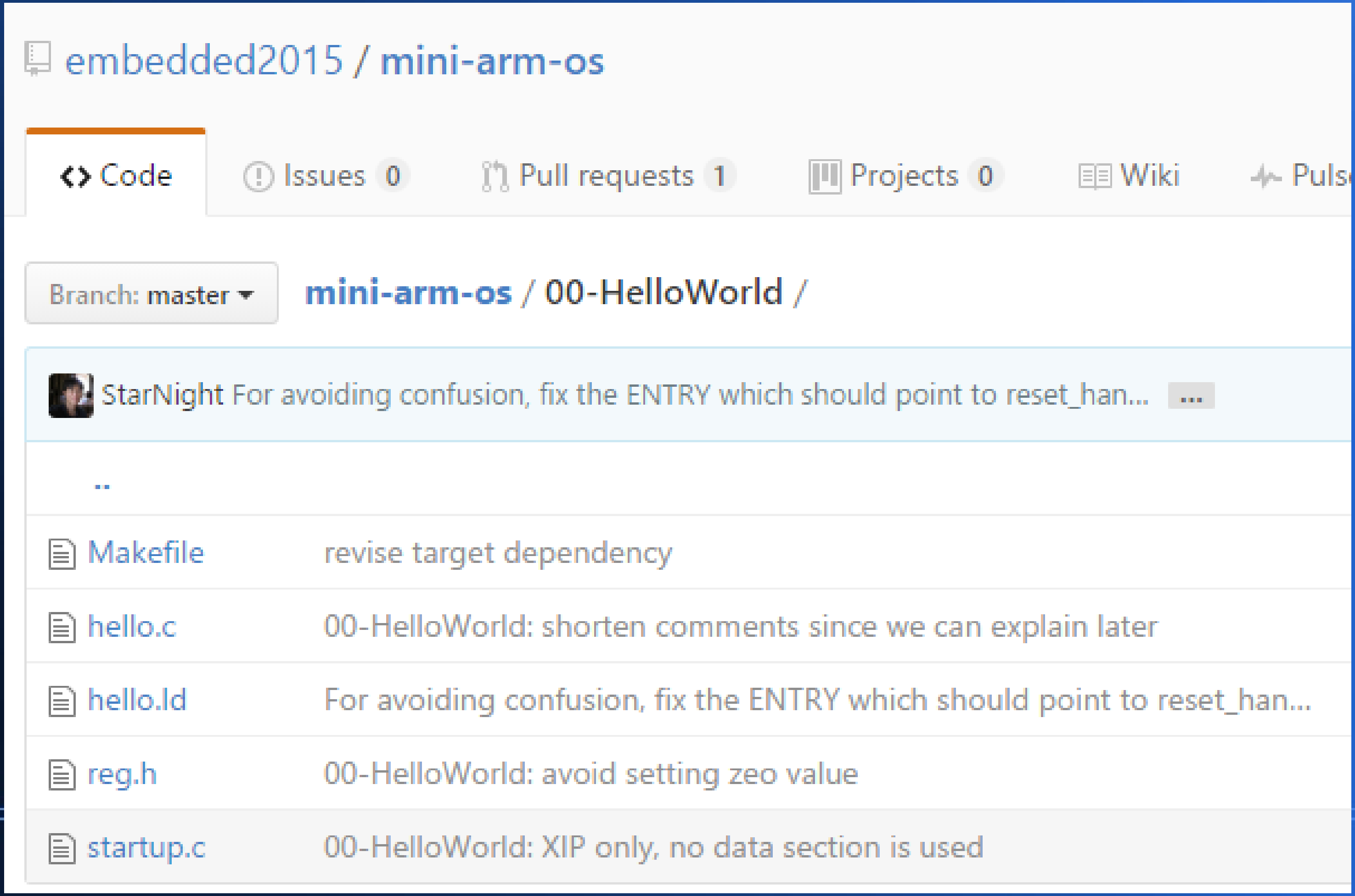
- 是 jserv 的 Mini ARM OS




https://github.com/embedded2015/mini-arm-os

# 讓我們從第一個

- 00-HelloWorld 開始



# 主程式 main.c

- 看來是用 UART 傳回一個 hello World! 字串
- 傳回到 UART 接收電腦上！（這電腦通常就是你用來編譯程式的那一台）

31 lines (23 sloc) | 520 Bytes

```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;
}

void main(void)
{
        *(RCC_APB2ENR) |= (uint32_t) (0x00000001 | 0x00000004);
        *(RCC_APB1ENR) |= (uint32_t) (0x00020000);

        /* USART2 Configuration */
        *(GPIOA_CRL) = 0x00004B00;
        *(GPIOA_CRH) = 0x44444444;

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

        puts("Hello World!\n");

        while (1);
}
```

# main.c 的其他部分

- 基本上就是用記憶體映射的方式，進行設定和輸出操作！

31 lines (23 sloc) | 520 Bytes

```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;
}

void main(void)
{
	*(RCC_APB2ENR) |= (uint32_t) (0x00000001 | 0x00000004);
	*(RCC_APB1ENR) |= (uint32_t) (0x00020000);

	/* USART2 Configuration */
	*(GPIOA_CRL) = 0x00004B00;
	*(GPIOA_CRH) = 0x44444444;

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

	puts("Hello World!\n");

	while (1);
}
```

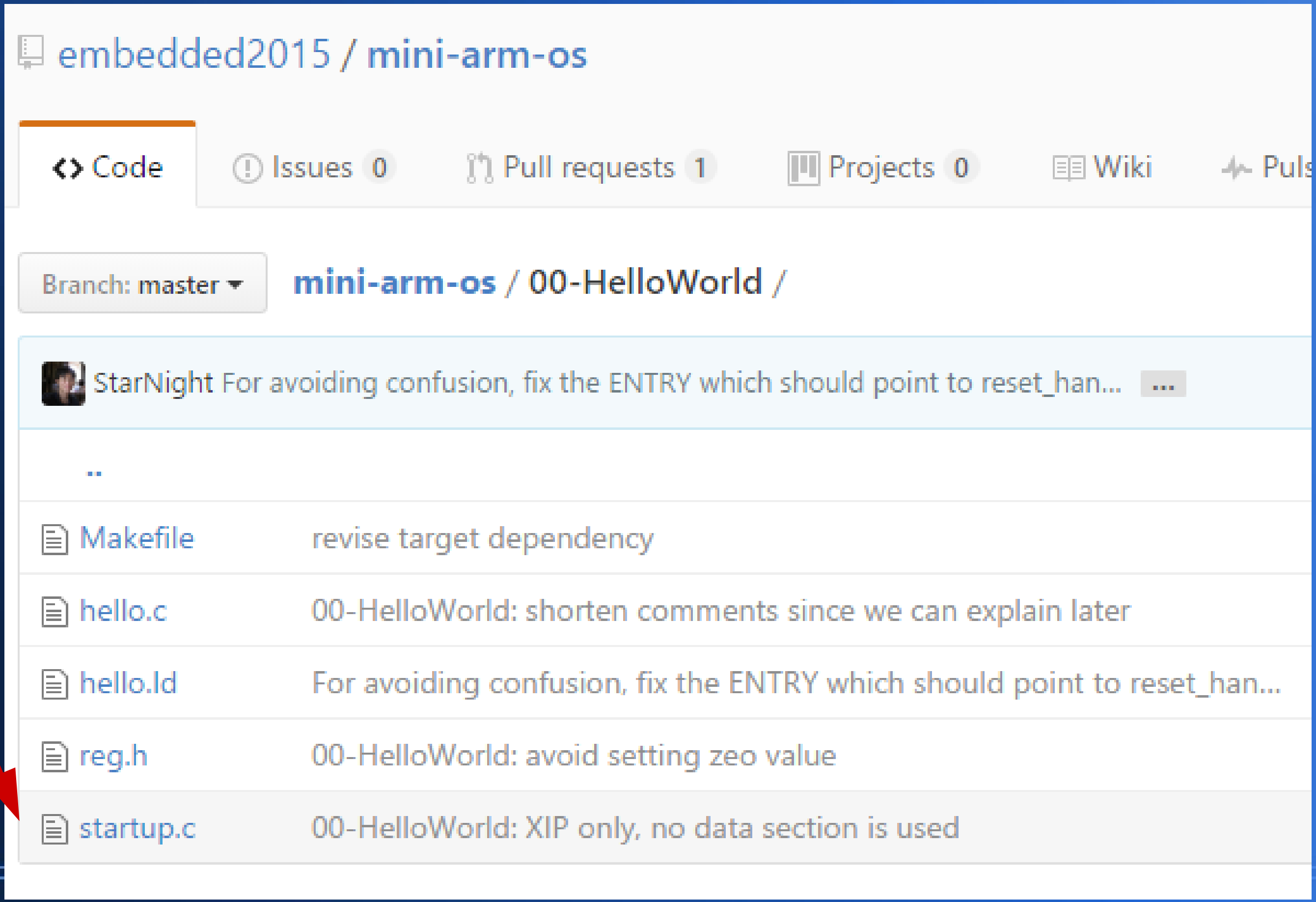
輸出操作

設定

# 但是、這只是一般的嵌入式寫法

- 啟動程式在哪裡呢？

我想應該是這裡吧？



# 點進去看看

- 挖！
- 連啟動程式

  都是用 C 寫的
- 竟然不需要

  半行組合語言

Branch: master ▾ **mini-arm-os** / 00-HelloWorld / **startup.c**

**jserv** 00-HelloWorld: XIP only, no data section is used

1 contributor

15 lines (12 sloc) | 238 Bytes

```c
#include <stdint.h>

extern void main(void);
void reset_handler(void)
{
	/* jump to C entry point */
	main();
}

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

中斷向量表

# 然後用連結的 ld 檔

- 指定中斷向量必須放在程式的前面。


Branch: master ▾ mini-arm-os / 00-HelloWorld / hello.ld

StarNight For avoiding confusion, fix the ENTRY which should point to

1 contributor

16 lines (13 sloc) | 152 Bytes

```
ENTRY(reset_handler)

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

SECTIONS
{
	.text :
	{
		KEEP(*(.isr_vector))
		*(.text)
	} >FLASH
}
```

中斷向量
放前面

# 當然

- 還需要 寫 makefile

然後用 qemu 來試跑

Branch: master ▾ mini-arm-os / 00-HelloWorld / Makefile

jserv revise target dependency

1 contributor

用 arm-none-eabi-gcc 交叉型編譯器，將系統編成 ARM 的映像檔

23 lines (18 sloc) | 574 Bytes

```
CROSS_COMPILE ?= arm-none-eabi-
CC := $(CROSS_COMPILE)gcc
CFLAGS = -fno-common -O0 \
         -mcpu=cortex-m3 -mthumb \
         -T hello.ld -nostartfiles \

TARGET = hello.bin
all: $(TARGET)

$(TARGET): hello.c startup.c
	$(CC) $(CFLAGS) $^ -o hello.elf
	$(CROSS_COMPILE)objcopy -Obinary hello.elf hello.bin
	$(CROSS_COMPILE)objdump -S hello.elf > hello.list

qemu: $(TARGET)
	@qemu-system-arm -M ? | grep stm32-p103 >/dev/null || exit
	@echo "Press Ctrl-A and then X to exit QEMU"
	@echo
	qemu-system-arm -M stm32-p103 -nographic -kernel hello.bin

clean:
	rm -f *.o *.bin *.elf *.list
```

# 而那些

- 和板子有關的記憶體映射位址
- 則是放在 reg.h 當中！

Branch: master ▾ **mini-arm-os** / 00-HelloWorld / **reg.h**

jserv 00-HelloWorld: avoid setting zeo value

1 contributor

24 lines (18 sloc) | 605 Bytes

```c
#ifndef __REG_H_
#define __REG_H_

#define __REG_TYPE      volatile uint32_t
#define __REG           __REG_TYPE *

/* RCC Memory Map */
#define RCC             ((__REG_TYPE) 0x40021000)
#define RCC_APB2ENR     ((__REG) (RCC + 0x18))
#define RCC_APB1ENR     ((__REG) (RCC + 0x1C))

/* GPIO Memory Map */
#define GPIOA           ((__REG_TYPE) 0x40010800)
#define GPIOA_CRL       ((__REG) (GPIOA + 0x00))
#define GPIOA_CRH       ((__REG) (GPIOA + 0x04))

/* USART2 Memory Map */
#define USART2          ((__REG_TYPE) 0x40004400)
#define USART2_SR       ((__REG) (USART2 + 0x00))
#define USART2_DR       ((__REG) (USART2 + 0x04))
#define USART2_CR1      ((__REG) (USART2 + 0x0C))

#endif
```

# 這樣

- 我們就看完了，第一個 Hello World! 的專案了！

# 只是、這還不能算一個作業系統

- 只能算是可以開機印字的程式而已！

# 但是看完後

- 已經學到不少東西了！

# 接著看 01-HelloWorld 吧！

- 奇怪的是，怎麼有兩個 HelloWorld 呢？
  - 00-HelloWorld
  - 01-HelloWorld
  - ???

# 比較一下，會發現

- 01 版的 HelloWorld，包含了 data 段與 BSS 段的內容

```
SECTIONS
{
	.text :
	{
		KEEP(*(.isr_vector))
		*(.text)
		*(.text.*)
		*(.rodata)
		_sromdev = .;
		_eromdev = .;
		_sidata = .;
	} >FLASH

	.data : AT(_sidata)
	{
		_sdata = .;
		*(.data)
		*(.data*)
		_edata = .;
	} >RAM

	.bss :
	{
		_sbss = .;
		*(.bss)
		_ebss = .;
	} >RAM

	_estack = ORIGIN(RAM) + LENGTH(RAM);
}
```

# 其 startup.c 裏

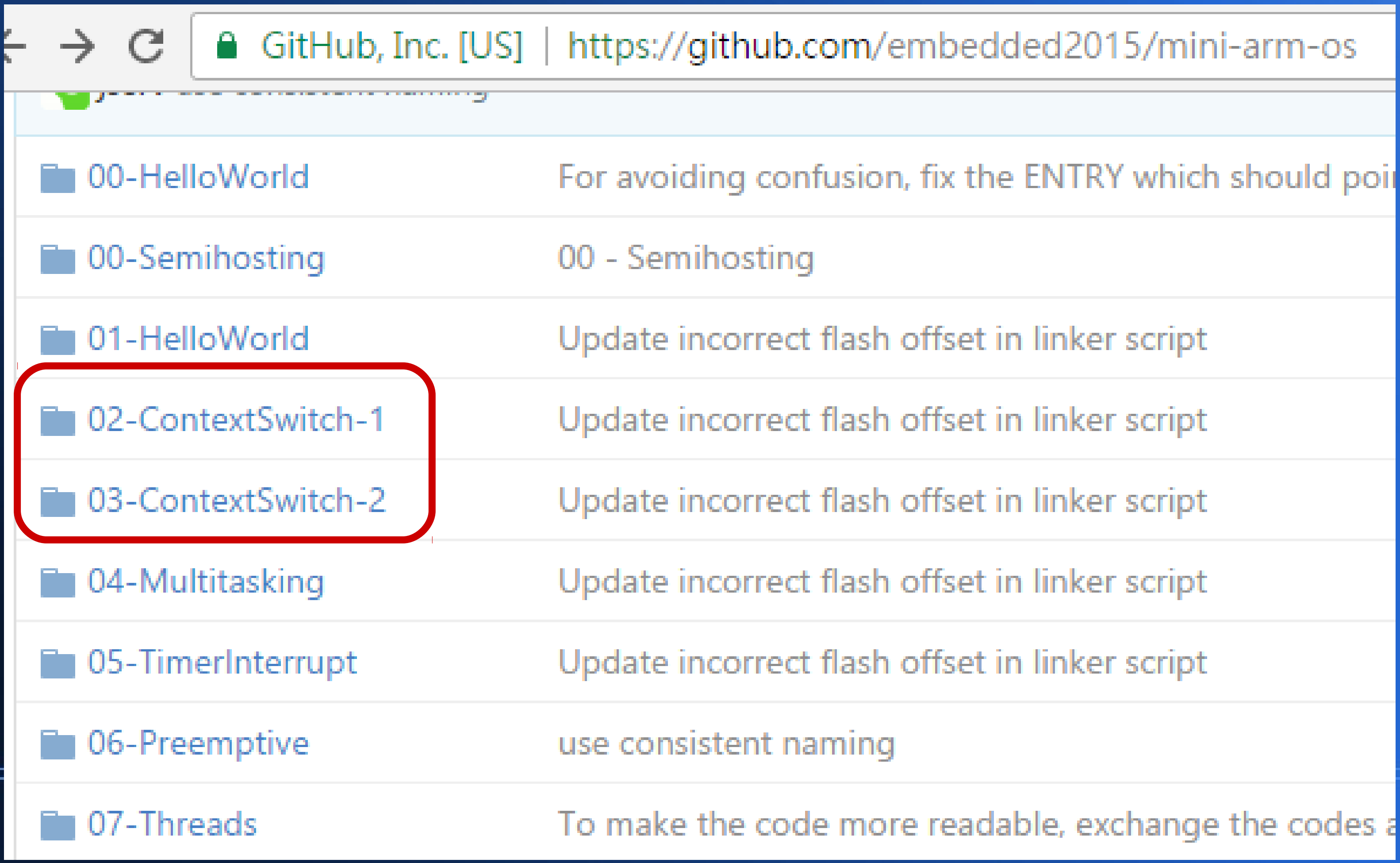
- 也多了這方面的定義與處理

```c
void nmi_handler(void)
{
        while (1);
}

void hardfault_handler(void)
{
        while (1);
}

__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 */
};

void rcc_clock_init(void)
{
        /* Reset the RCC clock configuration to the default reset
        /* Set HSION bit */
        *RCC_CR |= (uint32_t) 0x00000001;
```

```c
/* start address for the initialization values of the .data section.
defined in linker script */
extern uint32_t _sidata;
/* start address for the .data section. defined in linker script */
extern uint32_t _sdata;
/* end address for the .data section. defined in linker script */
extern uint32_t _edata;
/* start address for the .bss section. defined in linker script */
extern uint32_t _sbss;
/* end address for the .bss section. defined in linker script */
extern uint32_t _ebss;
/* end address for the stack. defined in linker script */
extern uint32_t _estack;

void rcc_clock_init(void);

void reset_handler(void)
{
        /* Copy the data segment initializers from flash to SRAM */
        uint32_t *idata_begin = &_sidata;
        uint32_t *data_begin = &_sdata;
        uint32_t *data_end = &_edata;
        while (data_begin < data_end) *data_begin++ = *idata_begin++;

        /* Zero fill the bss segment. */
        uint32_t *bss_begin = &_sbss;
        uint32_t *bss_end = &_ebss;
        while (bss_begin < bss_end) *bss_begin++ = 0;

        /* Clock system intitialization */
        rcc_clock_init();
```

# 接著讓我們開始進入比較令人興奮的主題

- 那就是ContextSwitch（內文切換）



# 所謂的內文切換

- 就是 multi-tesking 的多工作業系統，要切換 task 時，所需要做的動作。
- 這個動作通常需要儲存原 task 的暫存器，然後切換成新 task 的暫存器。

# 先看 02-ContextSwitch-1

- 使用者任務 usertask
- 切換動作

問題是： activate 到底是甚麼呢？

```c
void usertask(void)
{
	print_str("User Task #1\n");
	while (1); /* Never terminate the task */
}

int main(void)
{
	/* Initialization of process stack.
	 * r4, r5, r6, r7, r8, r9, r10, r11, lr */
	unsigned int usertask_stack[256];
	unsigned int *usertask_stack_start = usertask_stack + 256 - 16;
	usertask_stack_start[8] = (unsigned int) &usertask;

	usart_init();

	print_str("OS Starting...\n");
	activate(usertask_stack_start);

	while (1); /* We can't exit, there is nowhere to go */

	return 0;
}
```

https://github.com/embedded2015/mini-arm-os/blob/master/02-ContextSwitch-1/os.c

# 原來定義在

context_switch.S

這個組合語言檔裡面

問題是： activate 到底是甚麼呢？
答案是：這一段組合語言程式

這一段特別重要，是內文切換的重點程式！

但是也特別難懂 ... XD

Branch: master mini-arm-os / 02-ContextSwitch-1 / context_switch.S

rampant1018 Add usertask saved register

2 contributors

20 lines (15 sloc) | 313 Bytes

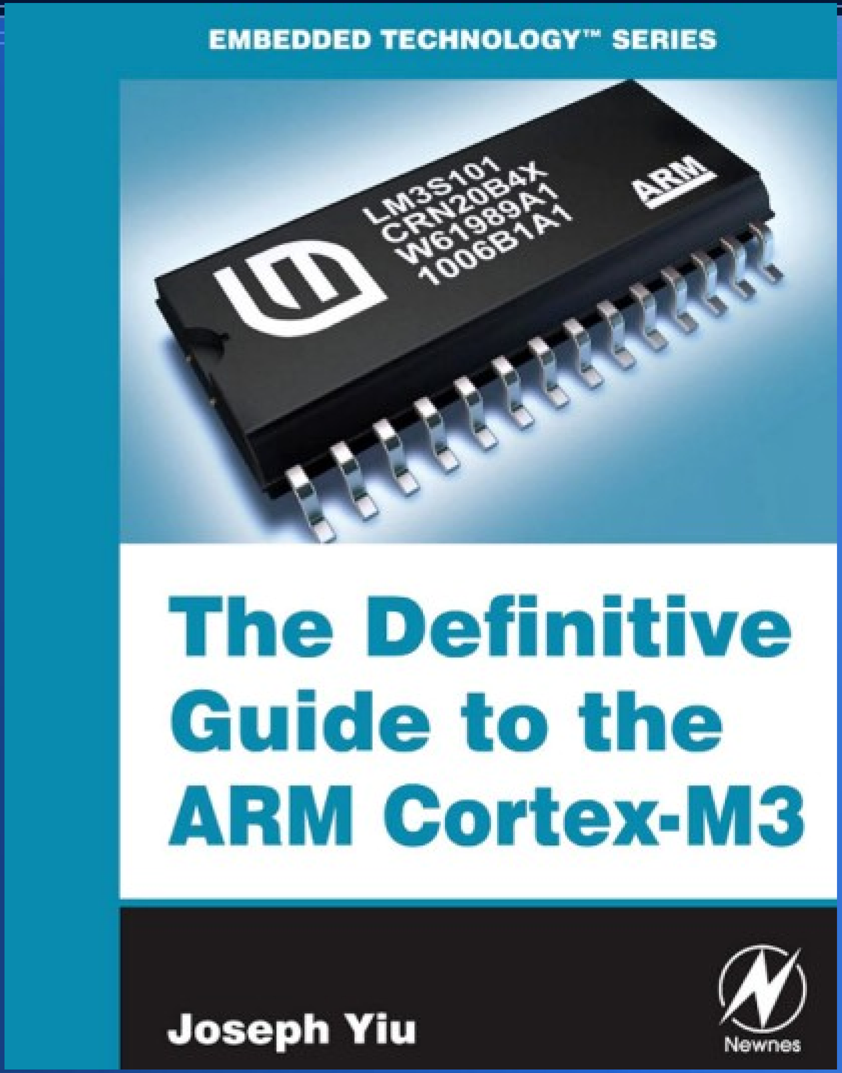
```
.thumb
.syntax unified

.global activate
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

	/* load user state */
	pop {r4, r5, r6, r7, r8, r9, r10, r11, lr}

	/* jump to user task */
	bx lr
```

# 要理解 context_switch.S

- 必須對 STM32 的 ARM Cotex M3 系列有所瞭解才行

# 想瞭解必須參考下列文件

- The Definitive Guide To ARM Cortex M3




http://tinymicros.com/mediawiki/images/7/75/Definitive_Guide_To_The_ARM_Cortex_M3.pdf

# ARM Cortex M3 的通用暫存器如下

| Name | | Functions (and Banked Registers) | |
|---|---|---|---|
| R0 | | General-Purpose Register | Low Registers |
| R1 | | General-Purpose Register | |
| R2 | | General-Purpose Register | |
| R3 | | General-Purpose Register | |
| R4 | | General-Purpose Register | |
| R5 | | General-Purpose Register | |
| R6 | | General-Purpose Register | |
| R7 | | General-Purpose Register | |
| R8 | | General-Purpose Register | High Registers |
| R9 | | General-Purpose Register | |
| R10 | | General-Purpose Register | |
| R11 | | General-Purpose Register | |
| R12 | | General-Purpose Register | |
| R13 (MSP) | R13 (PSP) | Main Stack Pointer (MSP), Process Stack Pointer (PSP) | |
| R14 | | Link Register (LR) | |
| R15 | | Program Counter (PC) | |

**Figure 2.2 Registers in the Cortex-M3**

# 特用暫存器如下

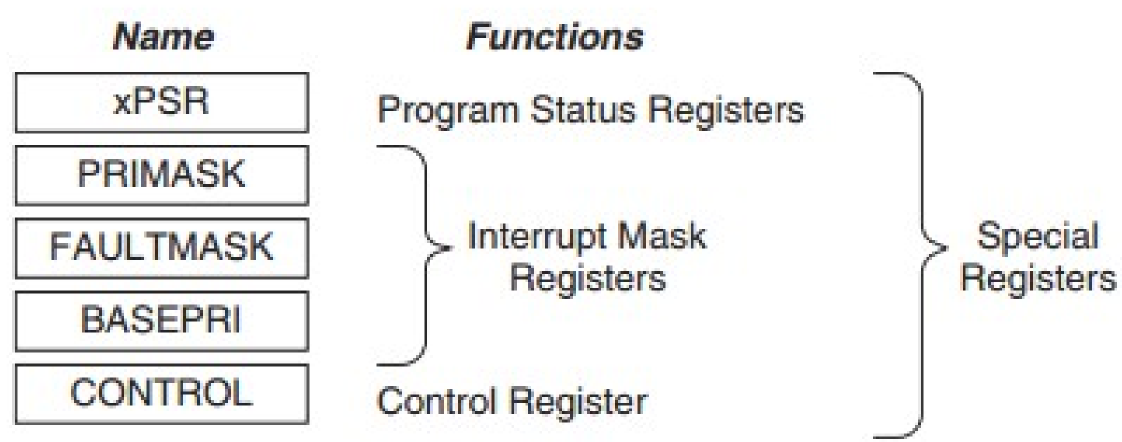



**Figure 2.3 Special Registers in the Cortex-M3**

**Table 2.1 Registers and Their Functions**

| Register | Function |
|---|---|
| xPSR | Provide ALU flags (zero flag, carry flag), execution status, and current executing interrupt number |
| PRIMASK | Disable all interrupts except the nonmaskable interrupt (NMI) and HardFault |
| FAULTMASK | Disable all interrupts except the NMI |
| BASEPRI | Disable all interrupts of specific priority level or lower priority level |
| CONTROL | Define privileged status and stack pointer selection |

# 記憶體映射配置如下

Overall, the 4 GB memory space can be divided into the ranges shown in Figure 2.6.

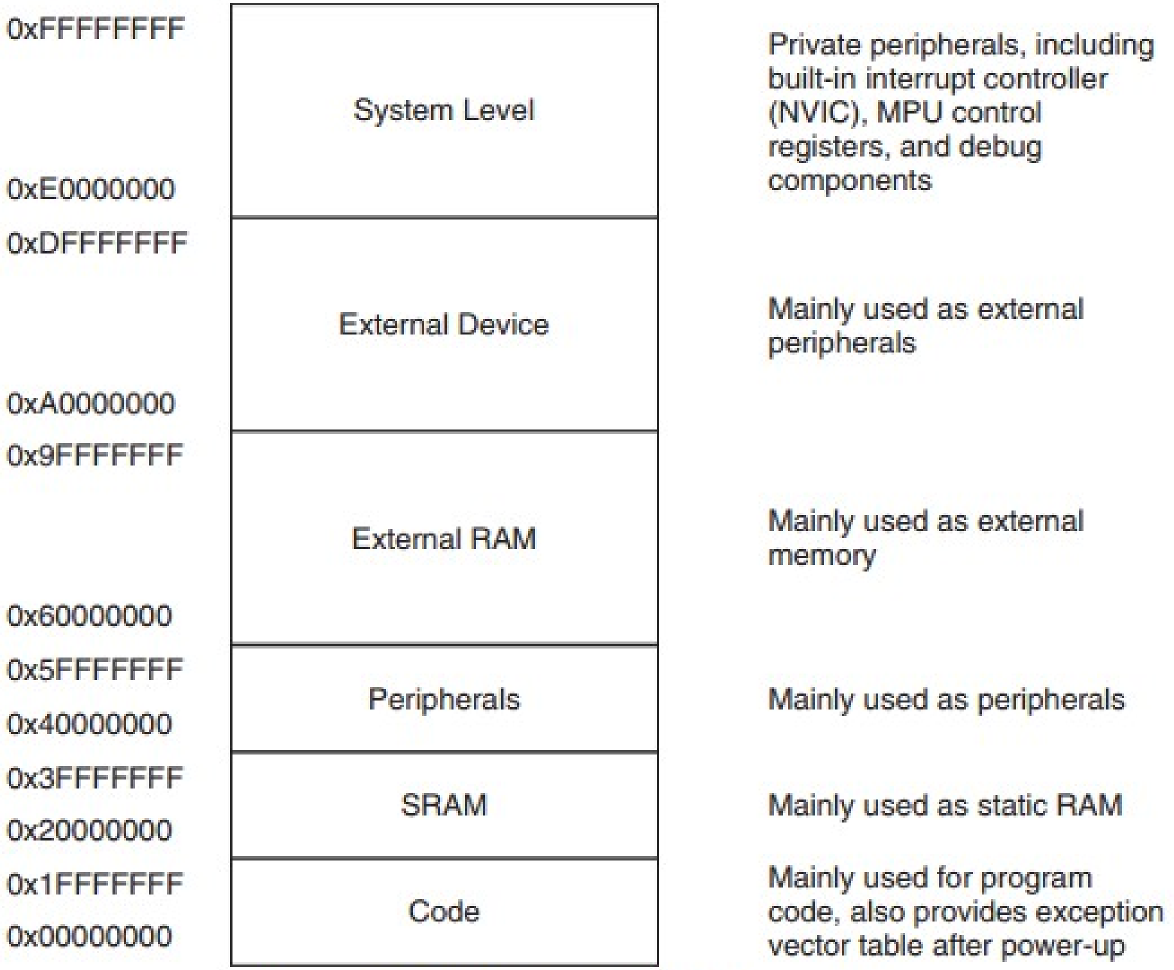



**Figure 2.6 The Cortex-M3 Memory Map**

# 堆疊暫存器 MSP, PSP 的功能

### *Stack Pointer R13*

R13 is the stack pointer. In the Cortex-M3 processor, there are two stack pointers. This duality allows two separate stack memories to be set up. When using the register name R13, you can only access the current stack pointer; the other one is inaccessible unless you use special instructions MSR and MRS. The two stack pointers are:

- Main Stack Pointer (MSP), or *SP_main* in ARM documentation: This is the default stack pointer; it is used by the OS kernel, exception handlers, and all application codes that require privileged access.
- Process Stack Pointer (PSP), or *SP_process* in ARM documentation: Used by the base-level application code (when not running an exception handler).

# 特殊暫存器的用途

## Special Registers

The special registers in the Cortex-M3 processor include these:

- Program Status Registers (PSRs)
- Interrupt Mask Registers (PRIMASK, FAULTMASK, and BASEPRI)
- Control Register (Control)

Special registers can only be accessed via MSR and MRS instructions; they do not have memory addresses:

```
MRS   <reg>, <special_reg>; Read special register
MSR   <special_reg>, <reg>; write to special register
```

***Program Status Registers (PSRs)***

The program status registers are subdivided into three status registers:

- Application PSR (APSR)[1]
- Interrupt PSR (IPSR)
- Execution PSR (EPSR)

The three PSRs can be accessed together or separately using the special register access instructions MSR and MRS. When they are accessed as a collective item, the name *xPSR* is used.

# 狀態暫存器的存取方式

| | 31 | 30 | 29 | 28 | 27 | 26:25 | 24 | 23:20 | 19:16 | 15:10 | 9 | 8 | 7 | 6 | 5 | 4:0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| APSR | N | Z | C | V | Q | | | | | | | | | | | |
| IPSR | | | | | | | | | | | | Exception Number | | | | |
| EPSR | | | | | | ICI/IT | T | | | ICI/IT | | | | | | |

**Figure 3.3 Program Status Registers (PSRs) in the Cortex-M3**

| | 31 | 30 | 29 | 28 | 27 | 26:25 | 24 | 23:20 | 19:16 | 15:10 | 9 | 8 | 7 | 6 | 5 | 4:0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| xPSR | N | Z | C | V | Q | ICI/IT | T | | | ICI/IT | | Exception Number | | | | |

**Figure 3.4 Combined Program Status Registers (xPSR) in the Cortex-M3**

You can read the program status registers using the MRS instruction. You can also change the APSR using the MSR instruction, but EPSR and IPSR are read-only. For example:

```
MRS    r0, APSR    ; Read Flag state into R0
MRS    r0, IPSR    ; Read Exception/Interrupt state
MRS    r0, EPSR    ; Read Execution state
MSR    APSR, r0    ; Write Flag state
```

In ARM assembler, when accessing xPSR (all three program status registers as one), the symbol *PSR* is used:

```
MRS    r0, PSR    ; Read the combined program status word
MSR    PSR, r0    ; Write combined program state word
```

# 控制暫存器的位元與用途

### *The Control Register*

The Control register is used to define the privilege level and the stack pointer selection. This register has two bits, as shown in Table 3.3.

**Table 3.3 Cortex-M3 CONTROL Register**

| Bit | Function |
|---|---|
| CONTROL[1] | Stack status:<br>1 = Alternate stack is used<br>0 = Default stack (MSP) is used<br>If it is in the Thread or base level, the alternate stack is the PSP. There is no alternate stack for handler mode, so this bit must be zero when the processor is in handler mode. |
| CONTROL[0] | 0 = Privileged in Thread mode<br>1 = User state in Thread mode<br>If in handler mode (not Thread mode), the processor operates in privileged mode. |

# 這樣我們就找到了 context_switch.S 裡的關鍵資訊，整理如下：

***The Control Register***

The Control register is used to define the privilege level and the stack pointer selection. This register has two bits, as shown in Table 3.3.

**Table 3.3 Cortex-M3 CONTROL Register**

| Bit | Function |
|---|---|
| CONTROL[1] | Stack status:<br>1 = Alternate stack is used<br>0 = Default stack (MSP) is used<br>If it is in the Thread or base level, the alternate stack is the PSP. There is no alternate stack for handler mode, so this bit must be zero when the processor is in handler mode. |
| CONTROL[0] | 0 = Privileged in Thread mode<br>1 = User state in Thread mode<br>If in handler mode (not Thread mode), the processor operates in privileged mode. |

```
MRS     r0, PSR     ; Read the combined program status word
MSR     PSR, r0     ; Write combined program state word
```

- Process Stack Pointer (PSP), or *SP_process* in ARM documentation: Used by the base-level application code (when not running an exception handler).

# 再重新看一次 context_switch.S

```
.thumb
.syntax unified

.global activate
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

	/* load user state */
	pop {r4, r5, r6, r7, r8, r9, r10, r11, lr}

	/* jump to user task */
	bx lr
```

# 把每一行的意義寫上

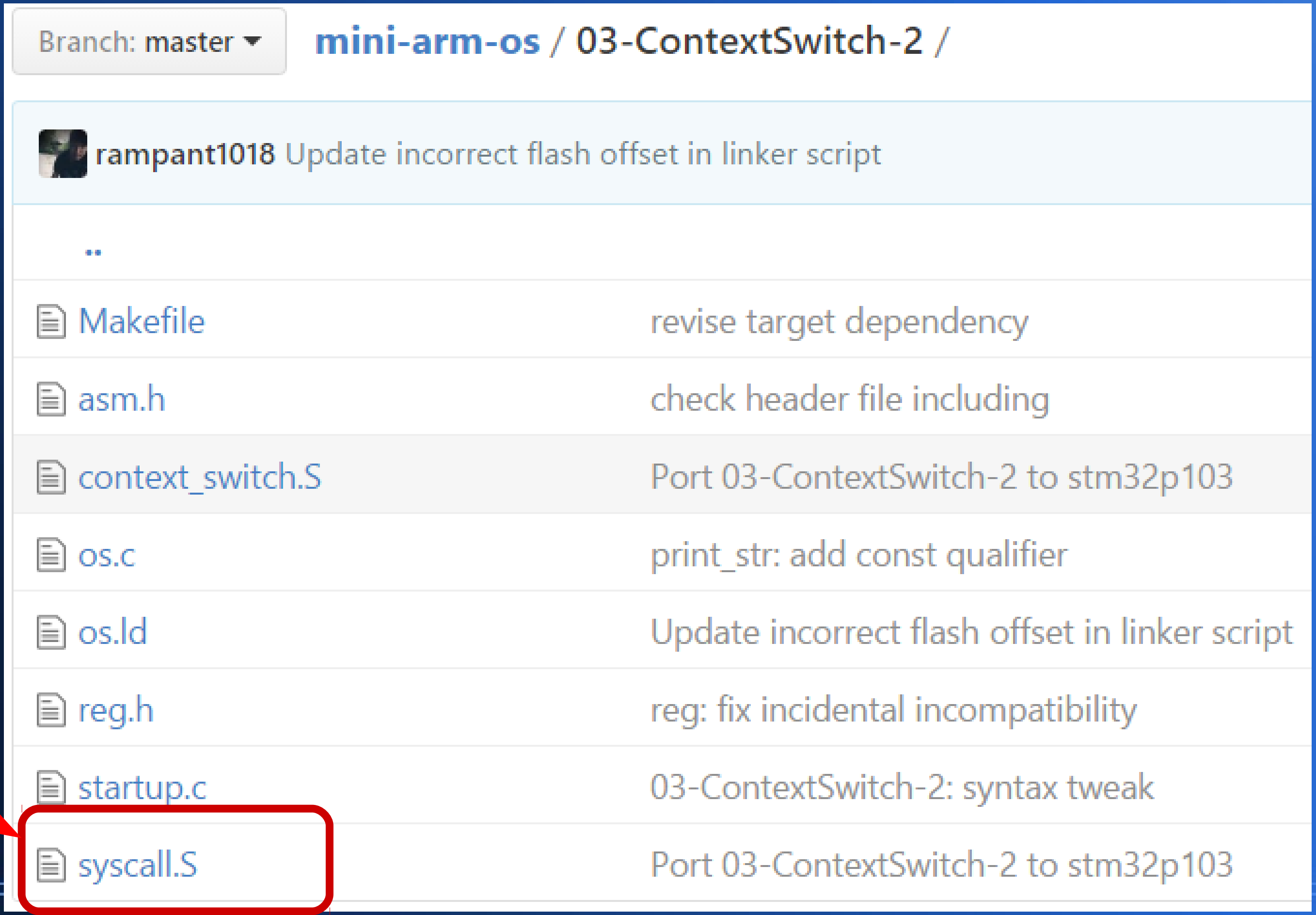
```
.thumb
.syntax unified

.global activate
activate:
        /* save kernel state */
        mrs ip, psr                                  // ip <= psr ( 狀態暫存器 )
        push {r4, r5, r6, r7, r8, r9, r10, r11, ip, lr}   // 儲存核心暫存器
                                                     // 包含在 ip 中的 psr

        /* switch to process stack */
        msr psp, r0        // psp <= r0, r0 裡放的是 task 的堆疊，因為 activate(stack)
        mov r0, #3
        msr control, r0    // control <= #3=0x011, 切換到使用者堆疊 (process stack)

        /* load user state */    // 載入行程的暫存器
        pop {r4, r5, r6, r7, r8, r9, r10, r11, lr}

        /* jump to user task */
        bx lr          // 跳到剛剛取出的 lr 暫存器，也就是行程的指令位址上開始執行
```

# 現在我們已經確定

- context_switch.S 的 activate 函數，確實完成了行程切換的動作了！

# 接著就可以看下一個

- 03-ContextSwitch-2 專案了！




多了 syscall.S
應該是系統呼叫

https://github.com/embedded2015/mini-arm-os/tree/master/03-ContextSwitch-2

# syscall.S 的內容

```
.thumb
.syntax unified

.global syscall
syscall:
        svc 0        // 這軟體中斷是幹嘛的 ???
        nop
        bx lr        // 返回
```

# 我們得查查 svc 0 是做甚麼的？

# 本來我查到的是
# svc 0 會印 R0 所指向的字串

## SVC Example: Use for Output Functions

Previously we developed various subroutines for output functions. Sometimes it is not good enough to use BL to call the subroutines—for example, when the functions are in different object files so that we might not be able to find out the address of the subroutines or when the branch address range is too large. In these cases, we might want to use SVC to act as an entry point for the output functions. For example:

```
LDR     R0,=HELLO_TXT
SVC     0          ; Display string pointed to by R0
MOV     R0,#'A'
SVC     1          ; Display character in R0
LDR     R0,=0xC123456
SVC     2          ; Display hexadecimal value in R0
MOV     R0,#1234
SVC     3          ; Display decimal value in R0
```

To use SVC, we need to set up the SVC handler. We can modify the function that we have done for IRQ. The only difference is that this function takes an exception type as input (SVC is exception type 11). In addition, this time we have further optimized the code to use the Thumb-2 instruction features:

# 但是我誤會了

- 那段只是舉例，不是真的
- Jserv(Jim Huang) 跳出來解說！

**Jim Huang** Page 63 - 64 說 svc 印出 r0 指向的字串，這描述不正確。SVC = Supervisor Calls，在 mini-arm-os 中，用來從 user mode 切換到 kernel mode: http://infocenter.arm.com/help/index.jsp......

ARM Information Center

INFOCENTER.ARM.COM

收回讚 · 回覆 · 移除預覽 · 50 · 昨天 13:11

**陳鍾誠** 原來是切換模式，我誤會了啊！

讚 · 回覆 · 6 · 昨天 13:15

**陳穎平** 所以基本上是 trap 進 kernel mode 這樣囉？

讚 · 回覆 · 23 小時前

**陳彥任** · Roger W. Wu和另外 1 人的朋友

這跟ARM CPU的架構有關.
以我以前用的 ARM9 而言, CPU本身提供兩個 modes (還有其他 modes, 如interrupt, etc...) 給 kernel space 和 user space 用. 在切換時可透過 software interrupt 進入 supervisor mode, 即 kernel space. SVC instruction bit0~23 可以定義多個 SVC routines....... 查看更多

| Exception Type | Priority | Mode | Vector | High Vector |
|---|---|---|---|---|
| Reset | 1 | Supervisor | 0x00000000 | 0xFFFF0000 |
| Undefined Instruction | 6 | Undefined | 0x00000004 | 0xFFFF0004 |
| SWI | 6 | Supervisor | 0x00000008 | 0xFFFF0008 |
| Prefetch Abort | 5 | Abort | 0x0000000C | 0xFFFF000C |
| Data Abort | 2 | Abort | 0x00000010 | 0xFFFF0010 |
| Reserved | | | 0x00000014 | 0xFFFF0014 |
| IRQ | 4 | IRQ | 0x00000018 | 0xFFFF0018 |
| FIQ | 3 | FIQ | 0x0000001C | 0xFFFF001C |

SAURABHSENGARBLOG.FILES.WORDPRESS.COM

# 所以 syscall.S 的內容

- 主要是切換模式，回到 OS 控制下

```
.thumb
.syntax unified

.global syscall
syscall:
        svc 0       // 從 user mode 切換到 kernel mode
        nop
        bx lr       // 返回
```

http://infocenter.arm.com/help/index.jsp?topic=%2Fcom.arm.doc.dai0179b%2Far01s02s07.html

# 於是 os.c 裡也可以進行系統呼叫了

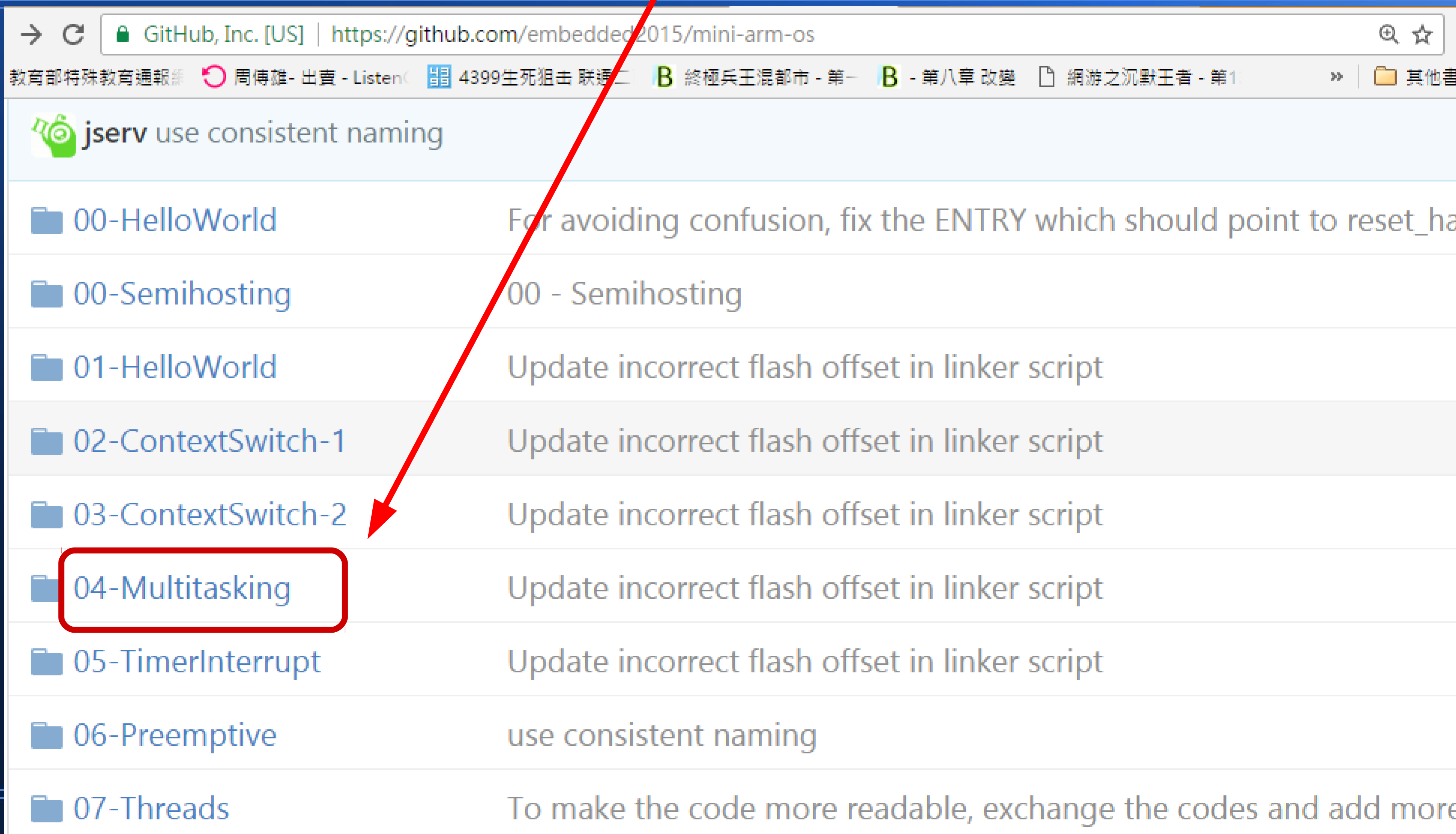
```
void usertask(void)     // 原本只會用 UART 傳回給主電腦印出的
{
        print_str("usertask: 1st call of usertask!\n");
        print_str("usertask: Now, return to kernel mode\n");
        syscall(); // 會從 user mode 切換到 kernel mode 交還控制權給 OS
        print_str("usertask: 2nd call of usertask!\n");
        print_str("usertask: Now, return to kernel mode\n");
        syscall(); // 會從 user mode 切換到 kernel mode 交還控制權給 OS
        while (1)
                /* wait */ ;
}
```

# 為了處理系統呼叫 syscall context_switch.S 也多了中斷處理

```
.type svc_handler, %function
.global svc_handler
svc_handler:
        /* save user state */    // 中斷時先儲存使用者行程狀態（暫存器群）
        mrs r0, psp
        stmdb r0!, {r4, r5, r6, r7, r8, r9, r10, r11, lr}

        /* load kernel state */ // 然後再恢復核心的狀態（暫存器群）
        pop {r4, r5, r6, r7, r8, r9, r10, r11, ip, lr}
        msr psr, ip

        bx lr

.global activate
activate:
        /* save kernel state */
        mrs ip, psr
```

# 以上就是《內文切換》的範例

# 接著要進入《多工》
# multitasking 的世界



# 檔案名稱幾乎沒改變

Branch: **master** ▾ **mini-arm-os** / 04-Multitasking /

**rampant1018** Update incorrect flash offset in linker script

..

| | |
|---|---|
| Makefile | revise target dependency |
| asm.h | check header file including |
| context_switch.S | Fix multitasking, now it works correctly |
| os.c | print_str: add const qualifier |
| os.ld | Update incorrect flash offset in linker script |
| reg.h | reg: fix incidental incompatibility |
| startup.c | 04-Multitasking: syntax tweak |
| syscall.S | Debugging multitasking |

# 但顯然加入了 task 的概念

```c
unsigned int *create_task(unsigned int *stack, void (*start)(void))
{
	static int first = 1;

	stack += STACK_SIZE - 32; /* End of stack, minus what we are about to push */
	if (first) {
		stack[8] = (unsigned int) start;
		first = 0;
	} else {
		stack[8] = (unsigned int) THREAD_PSP;
		stack[15] = (unsigned int) start;
		stack[16] = (unsigned int) 0x01000000; /* PSR Thumb bit */
	}
	stack = activate(stack);

	return stack;
}
```

https://github.com/embedded2015/mini-arm-os/blob/master/04-Multitasking/os.c

# 然後寫了兩個 task

```c
void task1_func(void)
{
	print_str("task1: Created!\n");
	print_str("task1: Now, return to kernel mode\n");
	syscall();
	while (1) {
		print_str("task1: Executed!\n");
		print_str("task1: Now, return to kernel mode\n");
		syscall(); /* return to kernel mode */
	}
}

void task2_func(void)
{
	print_str("task2: Created!\n");
	print_str("task2: Now, return to kernel mode\n");
	syscall();
	while (1) {
		print_str("task2: Executed!\n");
		print_str("task2: Now, return to kernel mode\n");
		syscall(); /* return to kernel mode */
	}
}
```

# 並且啟動了這兩個 task

```
unsigned int user_stacks[TASK_LIMIT][STACK_SIZE];
unsigned int *usertasks[TASK_LIMIT];
size_t task_count = 0;
size_t current_task;

usart_init();

print_str("OS: Starting...\n");
print_str("OS: First create task 1\n");
usertasks[0] = create_task(user_stacks[0], &task1_func);
task_count += 1;
print_str("OS: Back to OS, create task 2\n");
usertasks[1] = create_task(user_stacks[1], &task2_func);
task_count += 1;
```

# 還寫了大輪迴的排程法

```c
        print_str("\nOS: Start multitasking, back to OS till task yield!\n");
        current_task = 0;

        while (1) {
                print_str("OS: Activate next task\n");
                usertasks[current_task] = activate(usertasks[current_task]);
                print_str("OS: Back to OS\n");

                current_task = current_task == (task_count - 1) ? 0 : current_task + 1;
        }

        return 0;
}
```

以大輪迴方式選擇下一個行程

# 但是這個大輪迴排程是有缺陷的

- 因為沒有強制時間中斷，所以使用者 task 如果當掉了，那系統也會因此當掉。
- 只有在使用者 task 都正常地以 syscall 呼叫交還控制權給 OS 時，系統才能持續正常運作

# 這讓我想到很久以前

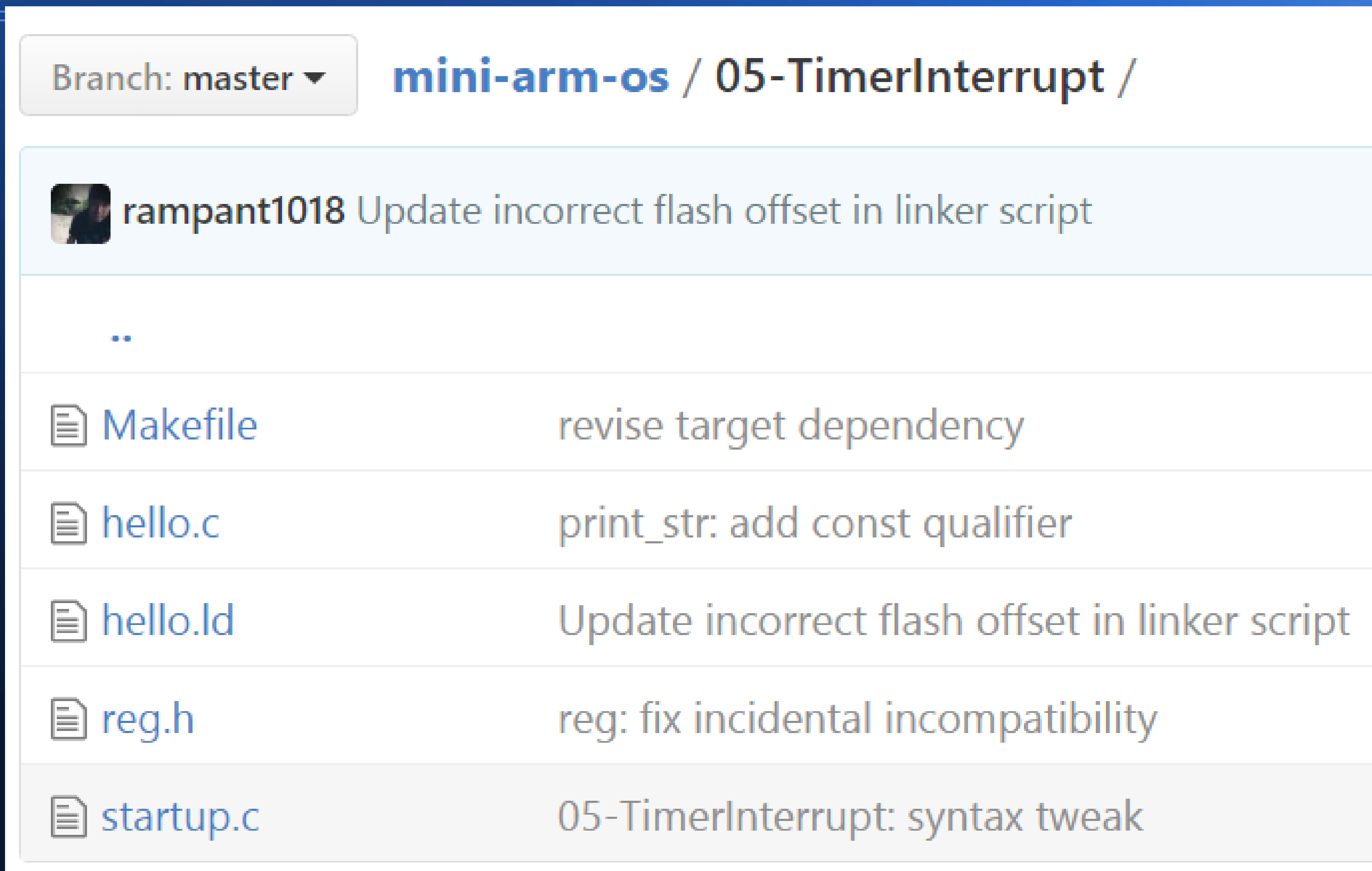
- Windows 3.1 的那個協同式多工

（這個故事應該至少要四十歲以上的人才會知道了）

# 所以我們需要先學習

- 如何加入強制時間中斷！

# 這就是專案 05-TimerInterrupt 的目的



# 在專案的 hello.c 裏有這段

```
void main(void)
{
        usart_init();

        print_str("Hello world!\n");

        /* SysTick configuration */
        *SYSTICK_LOAD = 7200000;
        *SYSTICK_VAL = 0;
        *SYSTICK_CTRL = 0x07;

        while (1); /* wait */
}

void __attribute__((interrupt)) systick_handler(void)
{
        print_str("Interrupt from System Timer\n");
}
```

// 因為系統頻率為 72M ，所以設定 7.2M 的話
// 每 0.1 秒會時間中斷一次！

# 當然、這些變數的記憶體映射

- 都定義在 reg.h 檔案裏了！

```c
/* SysTick Memory Map */
#define SYSTICK         ((__REG_TYPE) 0xE000E010)
#define SYSTICK_CTRL    ((__REG) (SYSTICK + 0x00))
#define SYSTICK_LOAD    ((__REG) (SYSTICK + 0x04))
#define SYSTICK_VAL     ((__REG) (SYSTICK + 0x08))
#define SYSTICK_CALIB   ((__REG) (SYSTICK + 0x0C))
```

# 學會強制時間中斷之後

- 我們就可以把那個《不夠好的排程系統》，加入《強制時間中斷》的功能了！

# 這就是下一個專案的任務了

Branch: master ▾ **mini-arm-os** / 06-Preemptive /

jserv use consistent naming

..

| | |
|---|---|
| Makefile | revise target dependency |
| asm.h | Initial the enviroment for task before create the task |
| context_switch.S | Initial the enviroment for task before create the task |
| os.c | use consistent naming |
| os.ld | Prevent delay from incorrect optimizations |
| reg.h | reg: fix incidental incompatibility |
| startup.c | 06-Preemptive: syntax tweak |
| syscall.S | Preemptive: use svc rather than swi |

# 專案 06-preemtive 一樣啟動兩個 task

```
usertasks[0] = create_task(user_stacks[0], &task1_func);
task_count += 1;
print_str("OS: Back to OS, create task 2\n");
usertasks[1] = create_task(user_stacks[1], &task2_func);
task_count += 1;

print_str("\nOS: Start round-robin scheduler!\n");

/* SysTick configuration */
*SYSTICK_LOAD = (CPU_CLOCK_HZ / TICK_RATE_HZ) - 1UL;
*SYSTICK_VAL = 0;
*SYSTICK_CTRL = 0x07;
current_task = 0;

while (1) {
	print_str("OS: Activate next task\n");
	usertasks[current_task] = activate(usertasks[current_task]);
	print_str("OS: Back to OS\n");

	current_task = current_task == (task_count - 1) ? 0 : current_task + 1;
}
```

# 而且一樣用大輪迴排班

```
usertasks[0] = create_task(user_stacks[0], &task1_func);
task_count += 1;
print_str("OS: Back to OS, create task 2\n");
usertasks[1] = create_task(user_stacks[1], &task2_func);
task_count += 1;

print_str("\nOS: Start round-robin scheduler!\n");

/* SysTick configuration */
*SYSTICK_LOAD = (CPU_CLOCK_HZ / TICK_RATE_HZ) - 1UL;
*SYSTICK_VAL = 0;
*SYSTICK_CTRL = 0x07;
current_task = 0;

while (1) {
        print_str("OS: Activate next task\n");
        usertasks[current_task] = activate(usertasks[current_task]);
        print_str("OS: Back to OS\n");

        current_task = current_task == (task_count - 1) ? 0 : current_task + 1;
}
```

# 而且每 0.1 秒觸發一次時間中斷

```c
usertasks[0] = create_task(user_stacks[0], &task1_func);
task_count += 1;
print_str("OS: Back to OS, create task 2\n");
usertasks[1] = create_task(user_stacks[1], &task2_func);
task_count += 1;

print_str("\nOS: Start round-robin scheduler!\n");

/* SysTick configuration */
*SYSTICK_LOAD = (CPU_CLOCK_HZ / TICK_RATE_HZ) - 1UL;
*SYSTICK_VAL = 0;
*SYSTICK_CTRL = 0x07;
current_task = 0;

while (1) {
        print_str("OS: Activate next task\n");
        usertasks[current_task] = activate(usertasks[current_task]);
        print_str("OS: Back to OS\n");

        current_task = current_task == (task_count - 1) ? 0 : current_task + 1;
}
```

```c
/* 72MHz */
#define CPU_CLOCK_HZ 72000000

/* 100 ms per tick. */
#define TICK_RATE_HZ 10
```

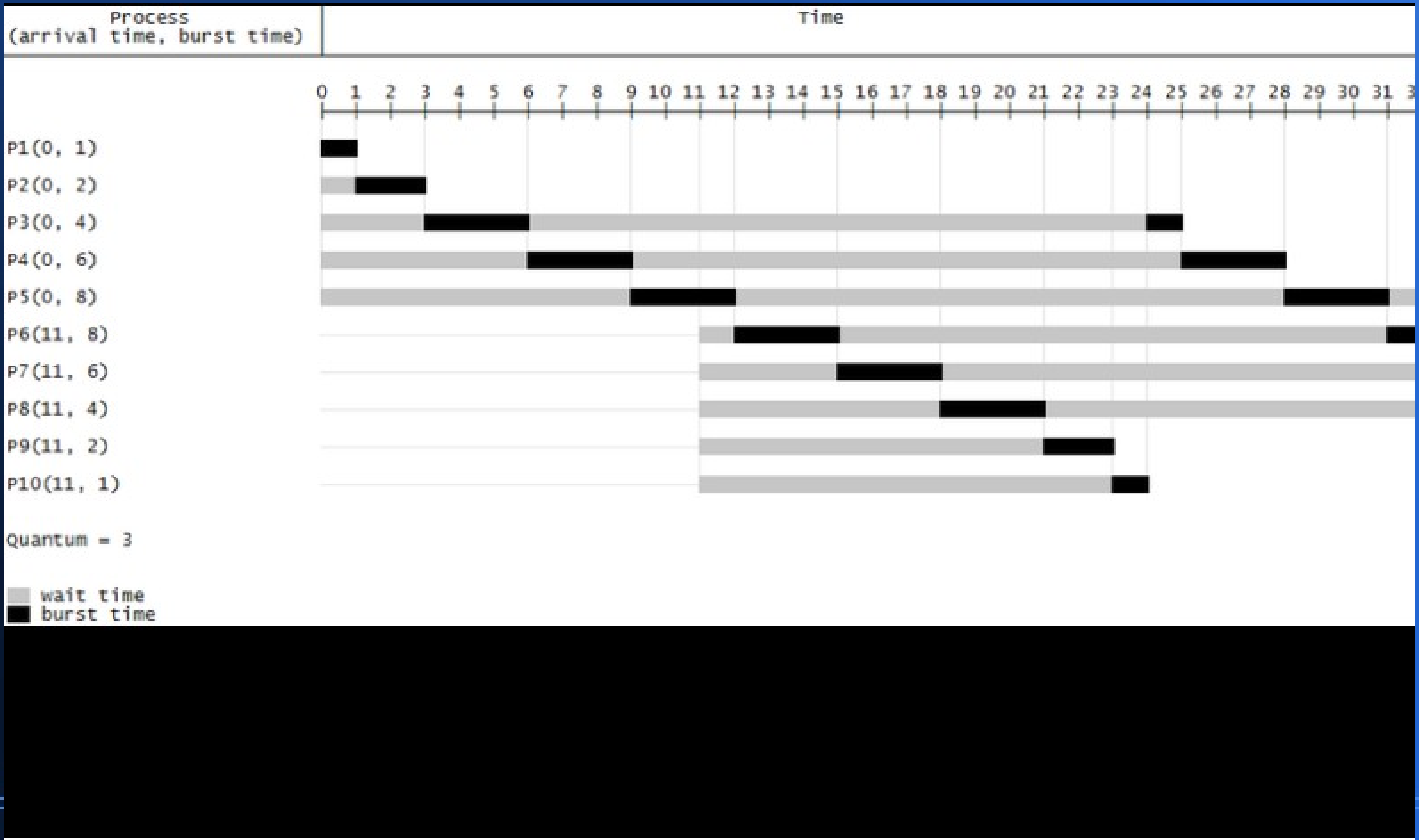
每秒觸發十次

# 然後開始放下兩個死賴很久的行程
# 讓他們會常常被中斷

```c
void task1_func(void)
{
        print_str("task1: Created!\n");
        print_str("task1: Now, return to kernel mode\n");
        syscall();
        while (1) {
                print_str("task1: Running...\n");
                delay(1000);
        }
}

void task2_func(void)
{
        print_str("task2: Created!\n");
        print_str("task2: Now, return to kernel mode\n");
        syscall();
        while (1) {
                print_str("task2: Running...\n");
                delay(1000);
        }
```

# 這種大輪迴排程

- 就是所謂的 round-robin scheduler 了！




A round-robin example with quantum=3.

# 現在我們看完一個

- 完整的《嵌入式作業系統》了！

# 不過目前的寫法

- 模組化還不夠好！
- 另外功能也還太弱！

# 所以接下來的專案

- 所以我們需要進一步模組化
- 然後加入像《記憶體管理》之類的功能！

# 這就是專案 07-Threads 的任務了

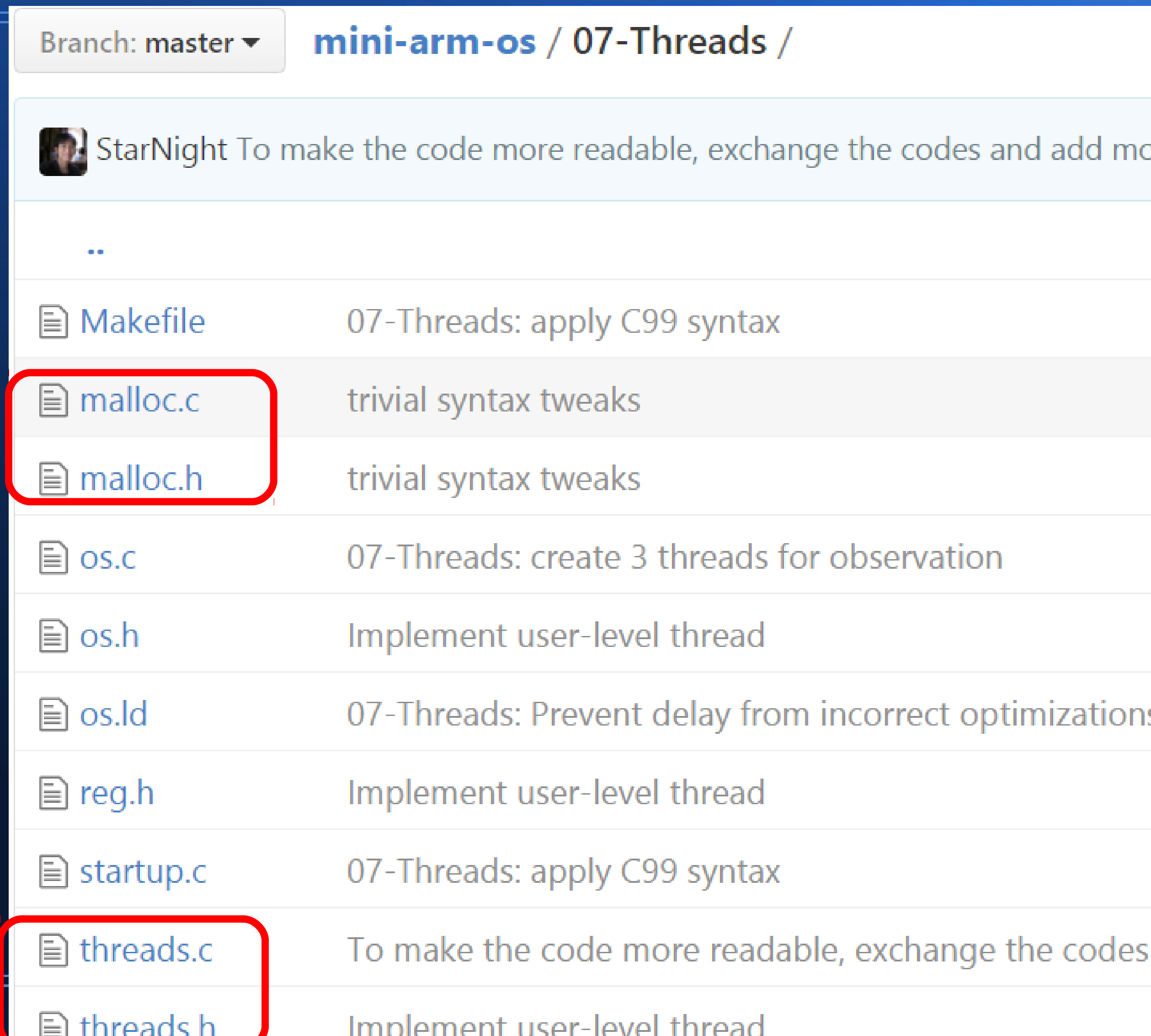
Branch: master ▾ mini-arm-os / 07-Threads /

StarNight To make the code more readable, exchange the codes and add mor

..

| | |
|---|---|
| Makefile | 07-Threads: apply C99 syntax |
| malloc.c | trivial syntax tweaks |
| malloc.h | trivial syntax tweaks |
| os.c | 07-Threads: create 3 threads for observation |
| os.h | Implement user-level thread |
| os.ld | 07-Threads: Prevent delay from incorrect optimizations |
| reg.h | Implement user-level thread |
| startup.c | 07-Threads: apply C99 syntax |
| threads.c | To make the code more readable, exchange the codes |
| threads.h | Implement user-level thread |

# 這專案包含了 malloc 記憶體管理以及 threads 的管理



# 每個模組都有適當的API

nch: master

mini-arm-os / 07-Threads / malloc.h

jserv trivial syntax tweaks

ontributors

ines (5 sloc) | 105 Bytes

```c
#ifndef __MALLOC_H_
#define __MALLOC_H_

void *malloc(unsigned int nbytes);
void free(void *ap);

#endif
```

nch: master

mini-arm-os / 07-Threads / threads.h

rampant1018 Implement user-level thread

ontributor

lines (7 sloc) | 191 Bytes

```c
#ifndef __THREADS_H__
#define __THREADS_H__

void thread_start();
int thread_create(void (*run)(void*), void* userdata);
void thread_kill(int thread_id);
void thread_self_terminal();

#endif
```

# 還不錯的資料結構

```c
#include <stddef.h>
#include "malloc.h"
#include "os.h"

typedef long Align;

union header {
        struct {
                union header *ptr;
                unsigned int size;
        } s;
        Align x;
};

typedef union header Header;

static unsigned char heaps[MAX_HEAPS];
static unsigned char *program_break = heaps;

static Header base; /* empty list to get started */
static Header *freep = NULL; /* start of free list */
```

```c
#include <stdint.h>
#include "threads.h"
#include "os.h"
#include "malloc.h"
#include "reg.h"

#define THREAD_PSP      0xFFFFFFFD

/* Thread Control Block */
typedef struct {
        void *stack;
        void *orig_stack;
        uint8_t in_use;
} tcb_t;

static tcb_t tasks[MAX_TASKS];
static int lastTask;
static int first = 1;
```

# 還有相當精簡的實作

```c
int thread_create(void (*run)(void *), void *userdata)
{
	/* Find a free thing */
	int threadId = 0;
	uint32_t *stack;

	for (threadId = 0; threadId < MAX_TASKS; threadId++) {
		if (tasks[threadId].in_use == 0)
			break;
	}

	if (threadId == MAX_TASKS)
		return -1;

	/* Create the stack */
	stack = (uint32_t *) malloc(STACK_SIZE * sizeof(uint32_t));
	tasks[threadId].orig_stack = stack;
	if (stack == 0)
		return -1;
```

```c
void thread_kill(int thread_id)
{
	tasks[thread_id].in_use = 0;

	/* Free the stack */
	free(tasks[thread_id].orig_stack);
}

void thread_self_terminal()
{
	/* This will kill the stack.
	 * For now, disable context switches to save ourselves.
	 */
	asm volatile("cpsid i\n");
	thread_kill(lastTask);
	asm volatile("cpsie i\n");

	/* And now wait for death to kick in */
	while (1);
}
```

# 並且把組合語言全改內嵌式寫法

```c
void thread_start()
{
	lastTask = 0;

	/* Save kernel context */
	asm volatile("mrs ip, psr\n"
	             "push {r4-r11, ip, lr}\n");

	/* To bridge the variable in C and the register in ASM,
	 * move the task's stack pointer address into r0.
	 * http://www.ethernut.de/en/documents/arm-inline-asm.html
	 */
	asm volatile("mov r0, %0\n" : : "r" (tasks[lastTask].stack));
	/* Load user task's context and jump to the task */
	asm volatile("msr psp, r0\n"
	             "mov r0, #3\n"
	             "msr control, r0\n"
	             "isb\n"
	             "pop {r4-r11, lr}\n"
	             "pop {r0}\n"
	             "bx lr\n");
}
```

```c
void __attribute__((naked)) pendsv_handler()
{
	/* Save the old task's context */
	asm volatile("mrs   r0, psp\n"
	             "stmdb r0!, {r4-r11, lr}\n");
	/* To get the task pointer address from result r0 */
	asm volatile("mov   %0, r0\n" : "=r" (tasks[lastTask].stack));

	/* Find a new task to run */
	while (1) {
		lastTask++;
		if (lastTask == MAX_TASKS)
			lastTask = 0;
		if (tasks[lastTask].in_use) {
			/* Move the task's stack pointer address into r0 */
			asm volatile("mov r0, %0\n" : : "r" (tasks[lastTask].stack));
			/* Restore the new task's context and jump to the task */
			asm volatile("ldmia r0!, {r4-r11, lr}\n"
			             "msr psp, r0\n"
			             "bx lr\n");
		}
	}
```

# 於是只剩下了 C 語言的檔案

Branch: **master** ▾ **mini-arm-os** / 07-Threads /

StarNight To make the code more readable, exchange the codes and add more

..

| | |
|---|---|
| Makefile | 07-Threads: apply C99 syntax |
| malloc.c | trivial syntax tweaks |
| malloc.h | trivial syntax tweaks |
| os.c | 07-Threads: create 3 threads for observation |
| os.h | Implement user-level thread |
| os.ld | 07-Threads: Prevent delay from incorrect optimizations |
| reg.h | Implement user-level thread |
| startup.c | 07-Threads: apply C99 syntax |
| threads.c | To make the code more readable, exchange the codes a |
| threads.h | Implement user-level thread |

# 這些就是

- 我從 jserv 的 Mini-ARM OS  專案上，所學到的作業系統設計實務！

# 歡迎直接到下列網址

- https://github.com/embedded2015/mini-arm-os

# 閱讀並執行那個寫得非常棒的

- mini-arm-os　專案！

# 相信您會有很大的收穫才對！

# 這就是

- 我們今天的十分鐘系列！

# 我們下回見！