**題目一**

將檔案分類放好後，我開始改寫Makefile

因為把這個資料夾獨立拉出來，所以把之前include的上層common內容直接貼過來

CROSS\_COMPILE = riscv64-unknown-elf-

CFLAGS = -nostdlib -fno-builtin -march=rv32ima -mabi=ilp32 -g

QEMU = qemu-system-riscv32

QFLAGS = -nographic -smp 1 -machine virt -bios none

GDB = riscv64-unknown-elf-gdb

CC = ${CROSS\_COMPILE}gcc

OBJCOPY = ${CROSS\_COMPILE}objcopy

OBJDUMP = ${CROSS\_COMPILE}objdump

把最後的編譯目標修改為objs資料夾底下的.o檔

OBJS = \

    ./objs/start.o \

    ./objs/mem.o \

    ./objs/entry.o \

    ./objs/kernel.o \

    ./objs/uart.o \

    ./objs/printf.o \

    ./objs/page.o \

    ./objs/sched.o \

分別設定各個資料夾下的目標與依賴關係

os.elf: ${OBJS}

    ${CC} ${CFLAGS} -T os.ld -o os.elf $^

    ${OBJCOPY} -O binary os.elf os.bin

./objs/%.o : ./startup/%.s

    $(CC) $(CFLAGS) -c -o $@ $<

./objs/%.o : ./asm/%.s

    $(CC) $(CFLAGS) -c -o $@ $<

./objs/%.o : ./kernel/%.c

    $(CC) $(CFLAGS) -c -o $@ $<

./objs/%.o : ./lib/%.c

    $(CC) $(CFLAGS) -c -o $@ $<

將objs/\*.o 加到clean 偽目標中

clean:

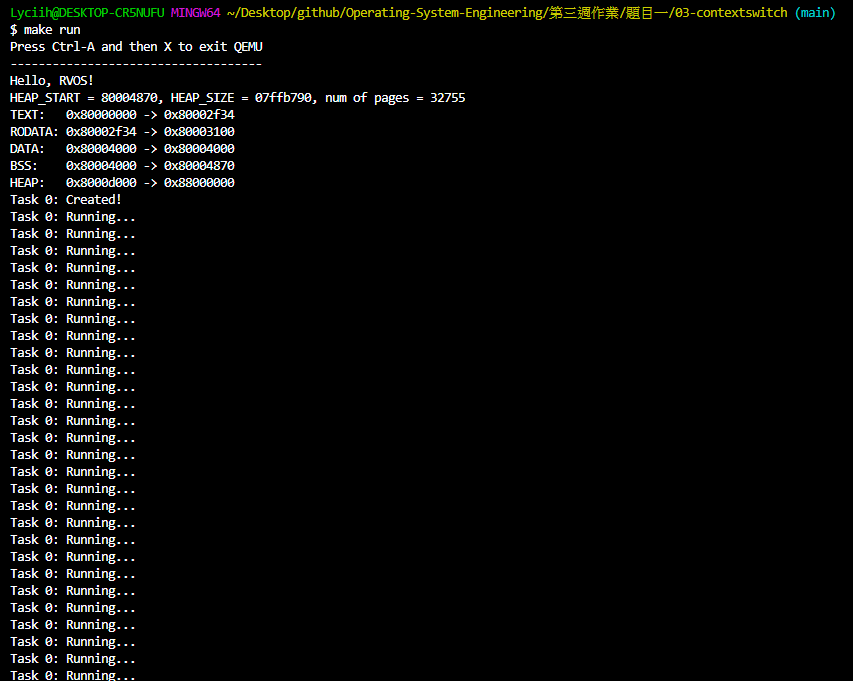
    rm -rf \*.o \*.bin \*.elf objs/\*.o

編譯看看



成功

測試看看是否能運作

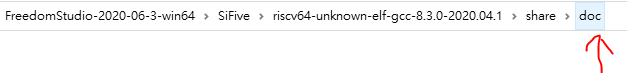
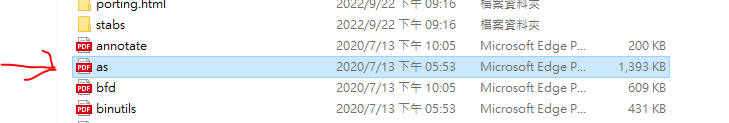


順利運行

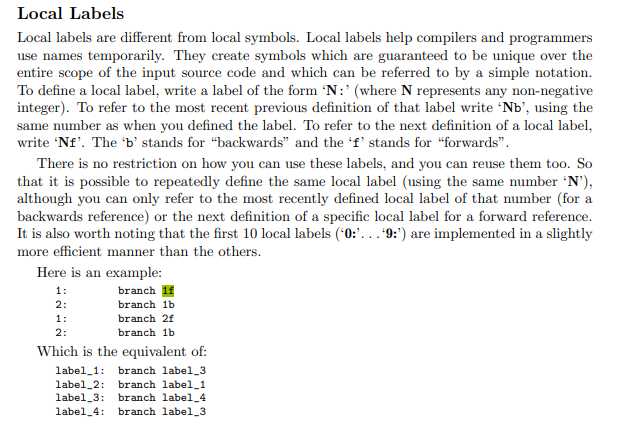
**題目二**

1. **請說明 beqz t6, 1f 指令會做何事?**

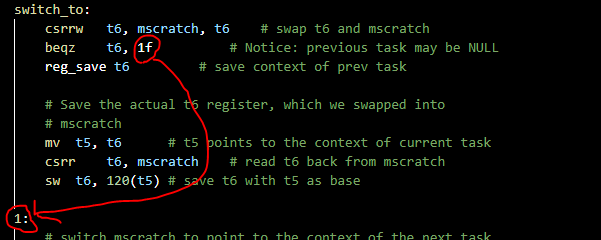
我檢查了教材中的各個檔案，最後找到一份名為 as.pdf 的文件，這是一份介紹組譯器指令的說明書

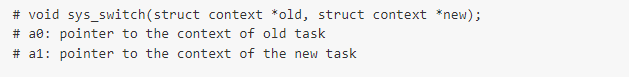
在裡面我找到了以下敘述



簡單來說，這是一個可以在label之間跳躍的指令，數字代表label名，< f > 代表forward ，< b >代表 back，所以beqz t6, 1f 的意思就是判斷如果t6為0，則往前跳到下一個名為1:的標籤



1. **請修改 entry.S 裡的switch\_to 函數，修改成 kernel start後switch到第一個Task用  
   並在entry.S裡增加一個 sys\_switch函數，主要讓kernel start以後 contextSwitch用**



為了能模擬從kernel跳到task，以及task跳到另一個task的狀況，我製作了兩個task

//為兩個task產生context，和他們各自的stack

struct context ctx\_task1;

struct context ctx\_task2;

uint8\_t task1\_stack[STACK\_SIZE];

uint8\_t task2\_stack[STACK\_SIZE];

//第一個task

void user\_task1(void)

{

    uart\_puts("Task 1: Running...|\n");

    uart\_puts("                  |...\n");

    uart\_puts("                     |...\n");

    uart\_puts("                        |...\n");

    task\_delay(6000);

    sys\_switch(&ctx\_task1, &ctx\_task2);

}

//第二個task

void user\_task2(void)

{

    uart\_puts("Task 2: Running............|.........\n");

    uart\_puts("                        ...|\n");

    uart\_puts("                     ...|\n");

    uart\_puts("                  ...|\n");

    task\_delay(6000);

    sys\_switch(&ctx\_task2, &ctx\_task1);

}

//將兩個task初始化

void schedule\_init()

{

    ctx\_task1.sp = (reg\_t) &task1\_stack[STACK\_SIZE-1];

    ctx\_task1.ra = (reg\_t) user\_task1;

    ctx\_task2.sp = (reg\_t) &task2\_stack[STACK\_SIZE-1];

    ctx\_task2.ra = (reg\_t) user\_task2;

}

接著修改組合語言

#------------------------從kernel跳到task的程式----------------------------------

.globl switch\_to

.align 4

switch\_to:                      # a0 代表第一個參數，是指向第一個task的context的指針

    mv  t6, a0                  #把 a0 寫入t6，用來當作載入第一個task的context的基址

    reg\_restore t6              #利用 t6 當基址載入第一個task的context

    ret                         #用ra覆蓋PC，跳到task的起始位置開始執行

#--------------------------從task跳到task的程式----------------------------------

.globl  sys\_switch

.align 4

sys\_switch:

    csrrw   t6, mscratch, t6    #將 t6 與 mscratch 做交換，暫時存放t6的值

    mv  t6, a0                  #把 a0 參數寫入t6

    reg\_save t6                 #利用 t6 中的地址儲當前的暫存器內容到自己的context

    mv  t5, t6                  #因為 t5 已經存過了，複寫沒關係，把 t6 中的地址寫到t5中

    csrr    t6, mscratch        #把 t6 跟 mscratch 交換前的值換回來

    sw  t6, 120(t5)             #利用 t5 中的地址儲存換回來的 t6 到自己context的 t6 位置

                                #以上完成存context的動作，下一階段開始載入目標task的context

                                # a1 代表第二個參數，是指向下一個task的context的指針

    mv  t6, a1                  #把 a1 寫入t6

    reg\_restore t6              #用 t6 中的地址載入下一個task的context

    ret                         #用ra覆蓋PC，跳到下一個task的起始位置開始執行

.end

將初始化及測試的task帶入os中執行

    printf("\n\n\n----- schedule\_init -----\n");

    schedule\_init();

    printf("\n\n\n----- schedule start -----\n\n\n");

    schedule();

測試成功

