

练习 3-1

使用 gcc 编译代码并使用 binutils 工具对生成的目标文件和可执行文件（ELF 格式）进行分析。具体要求如下：

- 编写一个简单的打印“hello world！”的程序源文件：`hello.c`
- 对源文件进行本地编译，生成针对支持 x86_64 指令集架构处理器的目标文件 `hello.o`。
- 查看 `hello.o` 的文件的文件头信息。
- 查看 `hello.o` 的 Section header table。
- 对 `hello.o` 反汇编，并查看 `hello.c` 的 C 程序源码和机器指令的对应关系。

我原本用的是 32 位元的 MinGW，題目說要能編譯出 x86_64 指令及架構的程式，所以我找到了 MinGW-w64 這組工具鏈

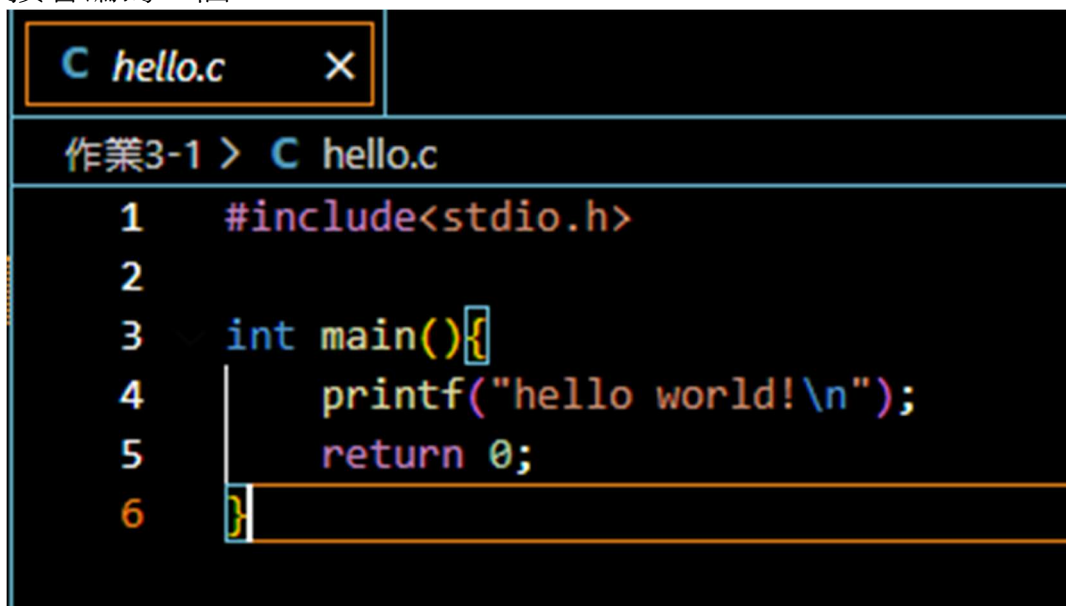
<https://sourceforge.net/projects/mingw-w64/files/>

Posix 是支持多線程的意思，seh 是異常處理機制，網路上說 sjlj 效能較差開銷大，所以選擇 she

MinGW-W64 GCC-8.1.0

- x86_64-posix-sjlj
- x86_64-posix-seh
- x86_64-win32-sjlj
- x86_64-win32-seh
- i686-posix-sjlj
- i686-posix-dwarf
- i686-win32-sjlj
- i686-win32-dwarf

接著編寫一個 `hello.c`



```
C hello.c ×
作業3-1 > C hello.c
1  #include<stdio.h>
2
3  int main(){
4      printf("hello world!\n");
5      return 0;
6  }
```

進行編譯

```
問題 輸出 偵錯主控台 終端機
Lyciih@DESKTOP-CR5NUFU MINGW64 ~/Desktop/OS/作業3-1
$ ls
hello.c

Lyciih@DESKTOP-CR5NUFU MINGW64 ~/Desktop/OS/作業3-1
$ gcc -c hello.c

Lyciih@DESKTOP-CR5NUFU MINGW64 ~/Desktop/OS/作業3-1
$ ls
hello.c hello.o
```

查看 hello.o 的檔頭訊息，使用 `objdump -f hello.o`

```
Lyciih@DESKTOP-CR5NUFU MINGW64 ~/Desktop/OS/作業3-1
$ objdump -f hello.o

hello.o:      file format pe-x86-64
architecture: i386:x86-64, flags 0x00000039:
HAS_RELOC, HAS_DEBUG, HAS_SYMS, HAS_LOCALS
start address 0x0000000000000000

Lyciih@DESKTOP-CR5NUFU MINGW64 ~/Desktop/OS/作業3-1
$ █
```

查看 hello.o 的 Section header table，使用 `objdump -h hello.o`

```
Lyciih@DESKTOP-CR5NUFU MINGW64 ~/Desktop/OS/作業3-1
$ objdump -h hello.o

hello.o:      file format pe-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text          00000030  0000000000000000  0000000000000000  0000012c  2**4
CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data           00000000  0000000000000000  0000000000000000  00000000  2**4
ALLOC, LOAD, DATA
  2 .bss            00000000  0000000000000000  0000000000000000  00000000  2**4
ALLOC
  3 .rdata          00000010  0000000000000000  0000000000000000  0000015c  2**4
CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .xdata           0000000c  0000000000000000  0000000000000000  0000016c  2**2
CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .pdata           0000000c  0000000000000000  0000000000000000  00000178  2**2
CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
  6 .rdata$zzz      00000040  0000000000000000  0000000000000000  00000184  2**4
CONTENTS, ALLOC, LOAD, READONLY, DATA

Lyciih@DESKTOP-CR5NUFU MINGW64 ~/Desktop/OS/作業3-1
$ █
```

對 hello.o 進行反組譯，使用 objdump -S hello.o

```
Lyciih@DESKTOP-CR5NUFU MINGW64 ~/Desktop/OS/作業3-1
$ objdump -S hello.o

hello.o:      file format pe-x86-64

Disassembly of section .text:

0000000000000000 <main>:
 0: 55                push    %rbp
 1: 48 89 e5          mov     %rsp,%rbp
 4: 48 83 ec 20       sub     $0x20,%rsp
 8: e8 00 00 00 00    callq   d <main+0xd>
 d: 48 8d 0d 00 00 00 00 lea     0x0(%rip),%rcx    # 14 <main+0x14>
14: e8 00 00 00 00    callq   19 <main+0x19>
19: b8 00 00 00 00    mov     $0x0,%eax
1e: 48 83 c4 20       add     $0x20,%rsp
22: 5d                pop     %rbp
23: c3                retq
24: 90                nop
25: 90                nop
26: 90                nop
27: 90                nop
28: 90                nop
29: 90                nop
2a: 90                nop
2b: 90                nop
2c: 90                nop
2d: 90                nop
2e: 90                nop
2f: 90                nop
```

問題來了，竟然沒有出現對應的 c code!!!

上網查詢，才知道要在編譯時加入 -g 的參數，這樣才會把 c code 的資訊一起編譯進去
使用 gcc -c -g hello.c 重新編譯一次，成功

```
Lyciih@DESKTOP-CR5NUFU MINGW64 ~/Desktop/OS/作業3-1
$ gcc -c -g hello.c

Lyciih@DESKTOP-CR5NUFU MINGW64 ~/Desktop/OS/作業3-1
$ objdump -S hello.o

hello.o:      file format pe-x86-64

Disassembly of section .text:

0000000000000000 <main>:
#include<stdio.h>

int main(){
 0: 55                push    %rbp
 1: 48 89 e5          mov     %rsp,%rbp
 4: 48 83 ec 20       sub     $0x20,%rsp
 8: e8 00 00 00 00    callq   d <main+0xd>
 d: 48 8d 0d 00 00 00 00 lea     0x0(%rip),%rcx    # 14 <main+0x14>
14: e8 00 00 00 00    callq   19 <main+0x19>
19: b8 00 00 00 00    mov     $0x0,%eax
1e: 48 83 c4 20       add     $0x20,%rsp
22: 5d                pop     %rbp
23: c3                retq
24: 90                nop
25: 90                nop
26: 90                nop
27: 90                nop
28: 90                nop
29: 90                nop
2a: 90                nop
2b: 90                nop
2c: 90                nop
2d: 90                nop
2e: 90                nop
2f: 90                nop
```

练习 3-2

如下例子 C 语言代码：

```
#include <stdio.h>

int global_init = 0x11111111;
const int global_const = 0x22222222;

void main()
{
    static int static_var = 0x33333333;
    static int static_var_uninit;

    int auto_var = 0x44444444;

    printf("hello world!\n");
    return;
}
```

请问编译为 .o 文件后，`global_init`, `global_const`, `static_var`, `static_var_uninit`, `auto_var` 这些变量分别存放在那些 section 里，`"hello world!\n"` 这个字符串又在哪里？并尝试用工具查看并验证你的猜测。

先建立檔案並編譯，再查看 section header

```
Lyciih@DESKTOP-CR5NUFU MINGW64 ~/Desktop/OS
$ cd 作業3-2

Lyciih@DESKTOP-CR5NUFU MINGW64 ~/Desktop/OS/作業3-2
$ ls
hello.c

Lyciih@DESKTOP-CR5NUFU MINGW64 ~/Desktop/OS/作業3-2
$ gcc -c -g hello.c

Lyciih@DESKTOP-CR5NUFU MINGW64 ~/Desktop/OS/作業3-2
$ objdump -h hello.o

hello.o:      file format pe-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text          00000030  0000000000000000  0000000000000000  0000021c  2**4
CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data          00000010  0000000000000000  0000000000000000  0000024c  2**4
CONTENTS, ALLOC, LOAD, DATA
  2 .bss           00000010  0000000000000000  0000000000000000  00000000  2**4
ALLOC
  3 .rdata         00000020  0000000000000000  0000000000000000  0000025c  2**4
CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .xdata         0000000c  0000000000000000  0000000000000000  0000027c  2**2
CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .pdata         0000000c  0000000000000000  0000000000000000  00000288  2**2
CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
  6 .debug_info    000001b8  0000000000000000  0000000000000000  00000294  2**0
CONTENTS, RELOC, READONLY, DEBUGGING
  7 .debug_abbrev  00000073  0000000000000000  0000000000000000  0000044c  2**0
CONTENTS, READONLY, DEBUGGING
  8 .debug_aranges 00000030  0000000000000000  0000000000000000  000004bf  2**0
CONTENTS, RELOC, READONLY, DEBUGGING
  9 .debug_line    00000046  0000000000000000  0000000000000000  000004ef  2**0
CONTENTS, RELOC, READONLY, DEBUGGING
10 .debug_str     00000000  0000000000000000  0000000000000000  00000000  2**0
READONLY, DEBUGGING
11 .rdata$zzz     00000040  0000000000000000  0000000000000000  00000535  2**4
CONTENTS, ALLOC, LOAD, READONLY, DATA
12 .debug_frame   00000040  0000000000000000  0000000000000000  00000575  2**3
CONTENTS, RELOC, READONLY, DEBUGGING
```


為了弄懂這題，首先要先了解組合語言中 **section** 到底是什麼，於是我查出每個片段名的意思

<https://learn.microsoft.com/en-us/windows/win32/debug/pe-format?redirectedfrom=MSDN#section-flags>
在微軟的網頁找到了 **PE fromat**，裡面有介紹每個片段的表格

The reserved sections and their attributes are described in the table below, followed by detailed descriptions for the section types that are persisted into executables and the section types that contain metadata for extensions.

Section Name	Content	Characteristics
.bss	Uninitialized data (free format)	IMAGE_SCN_CNT_UNINITIALIZED_DATA IMAGE_SCN_MEM_READ IMAGE_SCN_MEM_WRITE
.cormeta	CLR metadata that indicates that the object file contains managed code	IMAGE_SCN_LNK_INFO
.data	Initialized data (free format)	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ IMAGE_SCN_MEM_WRITE
.debug\$F	Generated FPO debug information (object only, x86 architecture only, and now obsolete)	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ IMAGE_SCN_MEM_DISCARDABLE
.debug\$P	Precompiled debug types (object only)	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ IMAGE_SCN_MEM_DISCARDABLE
.debug\$S	Debug symbols (object only)	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ IMAGE_SCN_MEM_DISCARDABLE
.debug\$T	Debug types (object only)	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ IMAGE_SCN_MEM_DISCARDABLE
.drective	Linker options	IMAGE_SCN_LNK_INFO
.edata	Export tables	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ
.idata	Import tables	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ IMAGE_SCN_MEM_WRITE
.idsym	Includes registered SEH (image only) to support IDL attributes. For information, see "IDL Attributes" in References at the end of this topic.	IMAGE_SCN_LNK_INFO
.pdata	Exception information	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ
.rdata	Read-only initialized data	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ
.reloc	Image relocations	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ IMAGE_SCN_MEM_DISCARDABLE
.rsrc	Resource directory	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ
.sbss	GP-relative uninitialized data (free format)	IMAGE_SCN_CNT_UNINITIALIZED_DATA IMAGE_SCN_MEM_READ IMAGE_SCN_MEM_WRITE IMAGE_SCN_GPREL The IMAGE_SCN_GPREL flag should be set for IA64 architectures only; this flag is not valid for other architectures. The IMAGE_SCN_GPREL flag is for object files only; when this section type appears in an image file, the IMAGE_SCN_GPREL flag must not be set.
.sdata	GP-relative initialized data (free format)	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ IMAGE_SCN_MEM_WRITE IMAGE_SCN_GPREL The IMAGE_SCN_GPREL flag should be set for IA64 architectures only; this flag is not valid for other architectures. The IMAGE_SCN_GPREL flag is for object files only; when this section type appears in an image file, the IMAGE_SCN_GPREL flag must not be set.
.srdata	GP-relative read-only data (free format)	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ IMAGE_SCN_GPREL The IMAGE_SCN_GPREL flag should be set for IA64 architectures only; this flag is not valid for other architectures. The IMAGE_SCN_GPREL flag is for object files only; when this section type appears in an image file, the IMAGE_SCN_GPREL flag must not be set.
.sxdata	Registered exception handler data (free format and x86/object only)	IMAGE_SCN_LNK_INFO Contains the symbol index of each of the exception handlers being referred to by the code in that object file. The symbol can be for an UNDEF symbol or one that is defined in that module.
.text	Executable code (free format)	IMAGE_SCN_CNT_CODE IMAGE_SCN_MEM_EXECUTE IMAGE_SCN_MEM_READ
.tls	Thread-local storage (object	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ

.tls\$	Thread-local storage (object only)	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ IMAGE_SCN_MEM_WRITE
.vsdata	GP-relative initialized data (free format and for ARM, SH4, and Thumb architectures only)	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ IMAGE_SCN_MEM_WRITE
.xdata	Exception information (free format)	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	00000030	0000000000000000	0000000000000000	0000021c	2**4
	CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE					
1	.data	00000010	0000000000000000	0000000000000000	0000024c	2**4
	CONTENTS, ALLOC, LOAD, DATA					
2	.bss	00000010	0000000000000000	0000000000000000	00000000	2**4
	ALLOC					
3	.rdata	00000020	0000000000000000	0000000000000000	0000025c	2**4
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
4	.xdata	0000000c	0000000000000000	0000000000000000	0000027c	2**2
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
5	.pdata	0000000c	0000000000000000	0000000000000000	00000288	2**2
	CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA					
6	.debug_info	000001b8	0000000000000000	0000000000000000	00000294	2**0
	CONTENTS, RELOC, READONLY, DEBUGGING					
7	.debug_abbrev	00000073	0000000000000000	0000000000000000	0000044c	2**0
	CONTENTS, READONLY, DEBUGGING					
8	.debug_aranges	00000030	0000000000000000	0000000000000000	000004bf	2**0
	CONTENTS, RELOC, READONLY, DEBUGGING					
9	.debug_line	00000046	0000000000000000	0000000000000000	000004ef	2**0
	CONTENTS, RELOC, READONLY, DEBUGGING					
10	.debug_str	00000000	0000000000000000	0000000000000000	00000000	2**0
	READONLY, DEBUGGING					
11	.rdata\$zzz	00000040	0000000000000000	0000000000000000	00000535	2**4
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
12	.debug_frame	00000040	0000000000000000	0000000000000000	00000575	2**3
	CONTENTS, RELOC, READONLY, DEBUGGING					

.text 段：用來保存實際的程式碼的地方

.data 段：已經初始化的資料

.bss 段：存放宣告變數的地方(還沒有初始值)

.rdata 段：已初始化的唯讀資料

.xdata 段：儲存異常訊息的地方(自由格式)

.pdata 段：儲存異常訊息的地方

剩下的目前還沒找到

所以我猜測：

global_init 在.data 段、global_const 在.rdata 段、static_var 在.data 段、static_var_uninit 在.bss 段

auto_var 在.data 段、"hello world!\n"在.data 段


```
#include <stdio.h>

int global_init = 0x11111111;
const int global_const = 0x22222222;

void main()
{
    static int static_var = 0x33333333;
    static int static_var_uninit;

    int auto_var = 0x44444444;

    printf("hello world!\n");
    return;
}
```

為了驗證我的想法，我使用以下指令

```
Lyciih@DESKTOP-CR5NUFU MINGW64 ~/Desktop/OS/作業3-2
$ objdump -D hello.o
```

global_init 和 static_var 都在.data 段沒錯

```
Disassembly of section .data:

0000000000000000 <global_init>:
 0:  11 11                adc    %edx,%rcx
 2:  11 11                adc    %edx,%rcx

0000000000000004 <static_var.3223>:
 4:  33 33                xor    (%rbx),%esi
 6:  33 33                xor    (%rbx),%esi
...
```

static_var_uninit 在.bss 段沒錯

```
Disassembly of section .bss:

0000000000000000 <static_var_uninit.3224>:
...
```

global_const 在.rdata 段沒錯

```
Disassembly of section .rdata:

0000000000000000 <global_const>:
 0:  22 22                and    (%rdx),%ah
 2:  22 22                and    (%rdx),%ah
 4:  68 65 6c 6c 6f        pushq  $0x6f6c6c65
 9:  20 77 6f                and    %dh,0x6f(%rdi)
 c:  72 6c                jnb    7a <static_var.3223+0x76>
 e:  64 21 00                and    %eax,%fs:(%rax)
...
```

用了另一個指令參數 -s(小寫) hello world 在.rdata 段

```
$ objdump -s hello.o

hello.o:      file format pe-x86-64

Contents of section .text:
 0000 554889e5 4883ec30 e8000000 00c745fc  UH..H..0.....E.
 0010 44444444 488d0d04 000000e8 00000000  DDDDH.....
 0020 904883c4 305dc390 90909090 90909090  .H..0].....
Contents of section .data:
 0000 11111111 33333333 00000000 00000000  ....3333.....
Contents of section .rdata:
 0000 22222222 68656c6c 6f20776f 726c6421  """"hello world!
 0010 00000000 00000000 00000000 00000000  .....
Contents of section .xdata:
```

但是卻找不到 auto_var

於是用 objdump -S(大寫) hello.o 再看一次程式碼

```
Lyciih@DESKTOP-CR5NUFU MINGW64 ~/Desktop/OS/作業3-2
$ objdump -S hello.o

hello.o:      file format pe-x86-64

Disassembly of section .text:

0000000000000000 <main>:

int global_init = 0x11111111;
const int global_const = 0x22222222;

void main()
{
    0:  55                      push    %rbp
    1:  48 89 e5                mov     %rsp,%rbp
    4:  48 83 ec 30             sub     $0x30,%rsp
    8:  e8 00 00 00 00          callq   d <main+0xd>
    static int static_var = 0x33333333;
    static int static_var_uninit;

    int auto_var = 0x44444444;
    d:  c7 45 fc 44 44 44 44    movl    $0x44444444,-0x4(%rbp)

    printf("hello world!\n");
```

發現 int auto_var = 0x44444444 這段竟然直接放在程式碼段裡

查詢到維基百科 .bss 和.data 所存的變數都是全域變數

在採用段式內存管理的架構中，**BSS**段（bss segment）或**DATA?**段（data? segment）通常是指用來存放程序中未初始化的全局變量的一塊內存區域。BSS是英文Block Started by Symbol的簡稱。BSS段屬於靜態內存分配。 .bss section 的空間結構類似於 stack

在採用段式內存管理的架構中，數據段（data segment）通常是指用來存放程序中已初始化的全局變量的一塊內存區域。數據段屬於靜態記憶體分配。

我的推論

Global 是全域的意思，const 是常量要放在唯讀，static 的記憶體位置是固定的所以也當作全域
int auto_var 是區域變數，在執行時才宣告。

练习 4-1

熟悉交叉编译概念，使用 `riscv gcc` 编译代码并使用 `binutils` 工具对生成的目标文件和可执行文件（ELF 格式）进行分析。具体要求如下：

- 编写一个简单的打印“hello world！”的程序源文件：`hello.c`
- 对源文件进行编译，生成针对支持 `rv32ima` 指令集架构处理器的目标文件 `hello.o`。

2 / 10

exercises.md

4/15/2021

- 查看 `hello.o` 的文件的文件头信息。
- 查看 `hello.o` 的 Section header table。
- 对 `hello.o` 反汇编，并查看 `hello.c` 的 C 程序源码和机器指令的对应关系。

首先寫一個 `hello.c`

```
C hello.c x
作業4-1 > C hello.c > main()
1  #include<stdio.h>
2
3  int main(){
4      printf("hello world!\n");
5      return 0;
6  }
```

為了不需要每次都打很長的 `riscv64-unknown-elf-gcc` 指令及參數，我練習寫一個 `Makefile`，並且先只用 `-march=rv32ima` 參數試試

```
M Makefile x
M Makefile
1  CC = riscv64-unknown-elf-gcc
2  FLAG = -march=rv32ima
3
4  all: hello.o
5  hello.o:
6      $(CC) $(FLAG) -c hello.c
7
8
9  clean:
10     rm -f *.o
11     rm -f *.exe
12

問題 輸出 偵錯主控台 終端機
hello.c Makefile

Lyciih@DESKTOP-CR5NMFU MINGW64 ~/Desktop/OS/test
$ make
riscv64-unknown-elf-gcc -march=rv32ima -c hello.c
cc1.exe: error: requested ABI requires -march to subsume the 'D' extension
cc1.exe: error: ABI requires -march=rv64
make: *** [Makefile:6: hello.o] Error 1

Lyciih@DESKTOP-CR5NMFU MINGW64 ~/Desktop/OS/test
$
```

結果產生錯誤，上面說還需要 ABI 的參數

查詢一下 ABI 是什麼東西

<https://blog.csdn.net/zoomdy/article/details/79353313>

RISC-V GCC 通过 `-mabi` 选项指定数据模型和浮点参数传递规则。有效的选项值包括 `ilp32`、`ilp32f`、`ilp32d`、`lp64`、`lp64f` 和 `lp64d`。前半部分指定数据模型，后半部分指定浮点参数传递规则。

数据模型:

x	int字长	long字长	指针字长
ilp32/ilp32f/ilp32d	32bits	32bits	32bits
lp64/lp64f/lp64d	32bits	64bits	64bits

浮点参数传递规则:

x	需要浮点扩展指令?	float参数	double参数
ilp32/lp64	不需要	通过整数寄存器 (a0-a1) 传递	通过整数寄存器 (a0-a3) 传递
ilp32f/lp64f	需要F扩展	通过浮点寄存器 (fa0-fa1) 传递	通过整数寄存器 (a0-a3) 传递
ilp32d/lp64d	需要F扩展和D扩展	通过浮点寄存器 (fa0-fa1) 传递	通过浮点寄存器 (fa0-fa1) 传递

浮点参数传递规则只跟 `-mabi` 选项有关, 和 `-march` 选项没有直接关系, 但是部分 `-mabi` 选项需要浮点寄存器, 浮点寄存器是通过浮点扩展指令引入的, 这就需要在 `-march` 选项中指定浮点扩展。

於是在 `makefile` 上加入 `-mabi=ilp32` (`ilp32` 意思是說, `float` 和 `double` 數都由整數寄存器傳遞)

```
2 FLAG = -march=rv32ima -mabi=ilp32
```

編譯成功

```
Lyciih@DESKTOP-CR5NUFU MINGW64 ~/Desktop/OS/test
$ make
riscv64-unknown-elf-gcc -march=rv32ima -nostdlib -c hello.c
cc1.exe: error: requested ABI requires -march to subsume the 'D' extension
cc1.exe: error: ABI requires -march=rv64
make: *** [Makefile:7: hello.o] Error 1

Lyciih@DESKTOP-CR5NUFU MINGW64 ~/Desktop/OS/test
$ make
riscv64-unknown-elf-gcc -march=rv32ima -mabi=ilp32 -c hello.c

Lyciih@DESKTOP-CR5NUFU MINGW64 ~/Desktop/OS/test
$ ls
hello.c hello.o Makefile
```

在 `Makefile` 中加入 `riscv64-unknown-elf-objdump` 把它設為偽目標

```
4 DUMP = riscv64-unknown-elf-objdump
5
6 all: hello.o
7 hello.o:
8 | $(CC) $(FLAG) -c -g hello.c
9 |
10
11 clean:
12 | rm -f *.o
13 | rm -f *.exe
14
15 dump:
16 | $(DUMP) -f hello.o
17
```

查看檔頭訊息

```
Lyciih@DESKTOP-CR5NUFU MINGW64 ~/Desktop/OS/test
$ make dump
riscv64-unknown-elf-objdump -f hello.o

hello.o:      file format elf32-littleriscv
architecture: riscv:rv32, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x00000000
```

網路上有另一種作法是用 readelf 這個命令

```
Lyciih@DESKTOP-CR5NUFU MINGW64 ~/Desktop/OS/作業4-1
$ riscv64-unknown-elf-readelf -h hello.o
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                           ELF32
  Data:                               2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                               REL (Relocatable file)
  Machine:                           RISC-V
  Version:                           0x1
  Entry point address:                0x0
  Start of program headers:           0 (bytes into file)
  Start of section headers:          10484 (bytes into file)
  Flags:                              0x0
  Size of this header:                52 (bytes)
  Size of program headers:            0 (bytes)
  Number of program headers:          0
  Size of section headers:            40 (bytes)
  Number of section headers:          21
  Section header string table index: 20
```

查看 Section header table

```
Lyciih@DESKTOP-CR5NUFU MINGW64 ~/Desktop/OS/test
$ make dump
riscv64-unknown-elf-objdump -h hello.o

hello.o:      file format elf32-littleriscv

Sections:
Idx Name          Size      VMA           LMA           File off  Algn
 0 .text          00000038  00000000  00000000  00000034  2**2
CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
 1 .data          00000000  00000000  00000000  0000006c  2**0
CONTENTS, ALLOC, LOAD, DATA
 2 .bss           00000000  00000000  00000000  0000006c  2**0
ALLOC
 3 .rodata        0000000d  00000000  00000000  0000006c  2**2
CONTENTS, ALLOC, LOAD, READONLY, DATA
 4 .debug_info    00000092c  00000000  00000000  00000079  2**0
CONTENTS, RELOC, READONLY, DEBUGGING
 5 .debug_abbrev  0000001bc  00000000  00000000  000000a5  2**0
CONTENTS, READONLY, DEBUGGING
 6 .debug_aranges 000000020  00000000  00000000  00000b61  2**0
CONTENTS, RELOC, READONLY, DEBUGGING
 7 .debug_line    000000167  00000000  00000000  00000b81  2**0
CONTENTS, RELOC, READONLY, DEBUGGING
 8 .debug_str     0000004d6  00000000  00000000  00000ce8  2**0
CONTENTS, READONLY, DEBUGGING
 9 .comment       000000029  00000000  00000000  000011be  2**0
CONTENTS, READONLY
10 .debug_frame   000000038  00000000  00000000  000011e8  2**2
CONTENTS, RELOC, READONLY, DEBUGGING
11 .riscv.attributes 000000026  00000000  00000000  00001220  2**0
CONTENTS, READONLY
```


進行反組譯

```
Lyciih@DESKTOP-CR5NUFU MINGW64 ~/Desktop/OS/test
$ make dump
riscv64-unknown-elf-objdump -S hello.o

hello.o:      file format elf32-littleriscv

Disassembly of section .text:

00000000 <main>:
#include<stdio.h>

int main(){
    0:  ff010113      addi    sp,sp,-16
    4:  00112623      sw      ra,12(sp)
    8:  00812423      sw      s0,8(sp)
   c:  01010413      addi    s0,sp,16
      printf("hello world!\n");
  10:  000007b7      lui     a5,0x0
  14:  00078513      mv      a0,a5
  18:  00000097      auipc   ra,0x0
  1c:  000080e7      jalr    ra # 18 <main+0x18>
      return 0;
  20:  00000793      li      a5,0
  24:  00078513      mv      a0,a5
  28:  00c12083      lw      ra,12(sp)
  2c:  00812403      lw      s0,8(sp)
  30:  01010113      addi    sp,sp,16
  34:  00008067      ret
```

最後我決定來查詢每個參數的意義

riscv64-unknown-elf-gcc -nostdlib -fno-builtin -march=rv32ima -mabi=ilp32 -g -Wall

-nostdlib	Do not look for object files in standard path. 不要在基本的路徑中尋找.o 檔
(-fno-pic) -fno-builtin	don't generate position-independent code (default) 預設不生成與位置無關的代碼，這裡使用 builtin 選項
-march	Generate code for given RISC-V ISA (e.g. RV64IM) 指定 RISC-V 的指令集架構，這裡使用 rv32ima
-mabi	Specify integer and floating-point calling convention. 指定整數和浮點數的表示方式
-g	Generate debug information in default format. 產生 gdb 中可以看到 debug 資訊
-Wall	Enable most warning messages. 打開全部的警告訊息

rv32ima: 由 rv32i、m、a 三個選項組成

rv32i: RV32I Base Integer Instruction Set (32-bit 基本整數指令集，有 32 個 32-bit 暫存器)

m: Standard Extension for Integer Multiplication and Divison (新增了整數乘法除法指令)

a: Standard Extension for Atomic Instructions (新增了 Atomic 相關的指令)

练习 4-2

基于 练习 4-1 继续熟悉 qemu/gdb 等工具的使用，具体要求如下：

- 将 `hello.c` 编译成可调试版本的可执行程序 `a.out`
- 先执行 `qemu-riscv32` 运行 `a.out`。
- 使用 `qemu-riscv32` 和 `gdb` 调试 `a.out`。

由於要直接輸出 `a.out`，於是把 `-c` 參數拿掉，要用 `qemu` 運行，所以將起始點設在 `0x80000000`

```
all: a.out
a.out:
$(CC) $(FLAG) -Ttext=0x80000000 -g hello.c
```

進行編譯

```
Lyciih@DESKTOP-CR5NUFU MINGW64 ~/Desktop/OS/作業4-2 (main)
$ make
riscv64-unknown-elf-gcc -march=rv32ima -mabi=ilp32 -Ttext=0x80000000 -g hello.c
```

看一下檔頭資訊

```
Lyciih@DESKTOP-CR5NUFU MINGW64 ~/Desktop/OS/作業4-2 (main)
$ make dump
riscv64-unknown-elf-objdump -f a.out

a.out:      file format elf32-littleriscv
architecture: riscv:rv32, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x8000001c
```

接著使用 `qemu` 來執行看看

先來了解教材中 `qemu` 的參數

`qemu-system-riscv32 -nographic -smp 1 -machine virt -bios none -kernel os.elf -s -S &`

-nographic	disable graphical output and redirect serial I/Os to console 關閉圖形輸出，並將 I/O 對接到當前的終端機
-smp	set the number of CPUs 設定 cpu 的核心數
-machine	selects emulated machine 指定模擬的機器，這裡選 virt(虛擬的)
-bios	set the filename for the BIOS 指定 bios 設定檔，這裡沒有，所以是 none
-kernel	指定使用的 kernel 這裡用我們寫的 os.elf
-s	shorthand for -gdb tcp::1234 縮寫(設定讓 gdb 用來除錯的端口為 1234 號)
-S	freeze CPU at startup (use 'c' to start execution) 剛啟動時凍結 CPU 的運行(用來等 gdb 打開)
&	(Linux 指令參數)在背景執行

設定完 Makefile 檔後便執行看看

```
Lyciih@DESKTOP-CR5NUFU MINGW64 ~/Desktop/OS/作業4-2 (main)
$ make qemu
qemu-system-riscv32 -nographic -smp 1 -machine virt -bios none -kernel a.out -s -S &
```

使用 ps 指令查看運行中的程式，qemu 執行中，等待我們用 gdb 去測試

```
$ ps
PID      PPID     PGID     WINPID   TTY        UID       STIME  COMMAND
109        1       109      10264  cons0     197609  22:51:14 /usr/bin/bash
340       294      340      14688  cons1     197609  23:09:00 /usr/bin/ps
293       109      293      10616  cons0     197609  23:07:08 /c/ProgramData/chocolatey/bin/make
294        1       294       9116  cons1     197609  23:07:22 /usr/bin/bash
252        1       250       7460  cons0     197609  23:05:36 /c/FreedomStudio-2020-06-3-win64/SiFive/riscv-qemu-4.2.0-2020.04.0/bin/qemu-system-riscv32
```

另外我們查看 1234port 是不是真的被我們的 qemu 使用。

PS 中的 winpid 是 7460，而使用 netstat 查詢占用 port 的程式，也是 7460，沒錯。

```
Lyciih@DESKTOP-CR5NUFU MINGW64 ~/Desktop/OS/作業4-2 (main)
$ netstat -a -o | grep :1234
TCP        0.0.0.0:1234          DESKTOP-CR5NUFU:0      LISTENING      7460
TCP        0.0.0.0:1234          DESKTOP-CR5NUFU:0      LISTENING      7460
```

接著使用 gdb 來調試。

一樣先瞭解 gdb 的各種參數

riscv64-unknown-elf-gdb os.elf -q -x gdbinit

-q	Do not print version number on startup. 在啟動時不顯示版本資訊
-x	Execute GDB commands from FILE. 啟動 gdb 後執行指定檔案中的指令

Gdbinit 檔中的 gdb 指令

set disassemble-next-line on	反組譯下一行要執行的程式
b _start	在 _start 的位置設立斷點
target remote : 1234	設定遠端除錯的端口 這裡是 1234 號
c	讓 CPU 開始執行程式

```
Lyciih@DESKTOP-CR5NUFU MINGW64 ~/Desktop/OS/作業4-2 (main)
$ make gdb
riscv64-unknown-elf-gdb a.out -q -x gdbinit
Reading symbols from a.out...
Breakpoint 1 at 0x8000002c
0x00001000 in ?? ()
=> 0x00001000:  97 02 00 00      auipc    t0,0x0
|
```

成功運行反組譯的功能，但是卻卡住了。

課堂上詢問老師，原來還缺乏啟動的開機程式以及可以輸出文字的 uart 驅動，所以我們回頭來把這兩個程式補上。

下一步要用組合語言來寫一個開機的程序。

```
start.s
1  .global _start          #讓linker可以找到_start
2
3  _start:                 #這是編譯器所認的程序進入點名字
4      csrr t0, mhartid    #將控制狀態暫存器的hartid讀入t0暫存器
5      bnez t0, stop       #因為我們目前只用第0顆CPU，所以如果ID判斷不為0則跳入stop程式
6
7
8      slli t0, t0, 10      #ID為0的CPU繼續執行這一段，我們的設每顆CPU的記憶體空間為1024，
9                          #為了讓日後多顆CPU的堆疊指針起始都可以在自己CPU所擁有的記憶體空間
10                         #我們必須取得每顆CPU專屬空間的起始位置
11                         #slli將自己往左偏移10bit，等同於乘以1024的效果
12                         #(此處為第0顆CPU，乘以1024還是0，是第0顆CPU記憶體的起始位置)
13
14
15
16
17      la sp, stacks       #ID為0的CPU繼續執行這一段，因為我們的設每顆CPU的記憶體空間為1024，
18      addi sp, sp, 1024   #stacks這個label將跳到下面的stacks副程式。la這個指令將stacks運作的起始位置返回給sp
19                          #而堆疊是由數字大的位址開始往下，因此我們要把堆疊指針sp先偏移1024
20
21
22      add sp, sp, t0      #將t0(當前CPU記憶體起始位置)加上sp(堆疊指針的相對位置)=該CPU堆疊指針真實的位置
23      j main             #跳到kernel(hello.c)中的main
24
25
26
27  stop:                  #ID不是0的CPU會跳來這裡
28      wfi                #這是讓CPU休眠的指令
29      j stop             #如果CPU因為中斷被喚醒，讓它繼續回去睡
30
31
32
33  stacks:                #建立記憶體空間的程式(類似開檔)
34      .skip 1024 * 8     #每顆CPU給1024，共有8顆，乘以8
35      .end               #結束開檔
```

翻閱 ref 資料，可以查到每個指令的意思

7.39 .global symbol, .globl symbol

.global makes the symbol visible to ld. If you define *symbol* in your partial program, its value is made available to other partial programs that are linked with it. Otherwise, *symbol* takes its attributes from a symbol of the same name from another file linked into the same program.

CSrr rd, csr x[rd] = CSRs[csr]
讀控制狀態寄存器 (*Control and Status Register Read*). 偽指令(Pseudoinstruction), RV32I and RV64I.

把控制狀態寄存器 *csr* 的值寫入 x[rd]，等同於 **csrrs** rd, csr, x0.

bnez rs1, offset if (rs1 \neq 0) pc += sext(offset)
不等於零時分支 (*Branch if Not Equal to Zero*). 偽指令(Pseudoinstruction), RV32I and RV64I.
可視為 **bne** rs1, x0, offset.

slli rd, rs1, shamt x[rd] = x[rs1] << shamt
立即數邏輯左移(*Shift Left Logical Immediate*). I-type, RV32I and RV64I.
把寄存器 x[rs1] 左移 *shamt* 位，空出的位置填入 0，結果寫入 x[rd]。對於 RV32I，僅當 *shamt*[5]=0 時，指令才是有效的。

壓縮形式: **c.slli** rd, shamt

31	26 25	20 19	15 14	12 11	7 6	0
000000	shamt	rs1	001	rd	0010011	

la rd, symbol x[rd] = &symbol

地址加载 (Load Address). 伪指令(Pseudoinstruction), RV32I and RV64I.

将 *symbol* 的地址加载到 x[rd]中。当编译位置无关的代码时，它会被扩展为对全局偏移量表(Global Offset Table)的加载。对于 RV32I，等同于执行 **auipc** rd, offsetHi，然后是 **lw** rd, offsetLo(rd);对于 RV64I，则等同于 **auipc** rd, offsetHi 和 **ld** rd, offsetLo(rd)。如果 offset 过大，开始的算加载地址的指令会变成两条，先是 **auipc** rd, offsetHi 然后是 **addi** rd, rd, offsetLo。

add rd, rs1, rs2 x[rd] = x[rs1] + x[rs2]

加 (Add). R-type, RV32I and RV64I.

把寄存器 x[rs2]加到寄存器 x[rs1]上，结果写入 x[rd]。忽略算术溢出。

压缩形式: **c.add** rd, rs2; **c.mv** rd, rs2

31	25 24	20 19	15 14	12 11	7 6	0
0000000	rs2	rs1	000	Rd	0110011	

wfi while (noInterruptPending) idle

等待中断(Wait for Interrupt). R-type, RV32I and RV64I 特权指令。

如果没有待处理的中断，则使处理器处于空闲状态。

31	25 24	20 19	15 14	12 11	7 6	0
0001000	00101	00000	000	00000	1110011	

j offset pc += sext(offset)

跳转 (Jump). 伪指令(Pseudoinstruction), RV32I and RV64I.

把 *pc* 设置为当前值加上符号位扩展的 *offset*，等同于 **jal** x0, offset.

將開機程式單獨編譯，再用 gdb 測試，成功!!

gdb 偵測到的最後一行是 **wfi**，因為我們的開機程式最後並沒有跳到 **kernel** 的指令，所以 CPU 執行完最後一行就休息了

```
Lyciih@DESKTOP-CR5NUFU MINGW64 ~/Desktop/OS/作業4-2 (main)
$ make qemu
qemu-system-riscv32 -nographic -smp 1 -machine virt -bios none -kernel a.out -s -S &
riscv64-unknown-elf-gdb a.out -q -x gdbinit
Reading symbols from a.out...
Breakpoint 1 at 0x80000000: file start.S, line 4.
0x00001000 in ?? ()
=> 0x00001000: 97 02 00 00      auipc    t0,0x0

Breakpoint 1, _start () at start.S:4
4      csrr t0, mhartid      #hartidt0
=> 0x80000000 <_start+0>:    f3 22 40 f1      csrr     t0,mhartid
(gdb) n
5      bnez t0, stop         #0CPUID0stop
=> 0x80000004 <_start+4>:    63 98 02 00      bnez     t0,0x80000014 <stop>
(gdb) n
8      slli t0, t0, 10       #ID0CPU1024
=> 0x80000008 <_start+8>:    93 92 a2 00      slli     t0,t0,0xa
(gdb) n
16     la sp, 1024           #ID0CPU1024
=> 0x8000000c <_start+12>:   13 01 00 40      li       sp,1024
(gdb) n
_start () at start.S:21
21     add sp, sp, t0        #t0(CPU)sp()=CPU
=> 0x80000010 <_start+16>:   33 01 51 00      add      sp,sp,t0
(gdb) n
stop () at start.S:25
25     wfi                  #CPU
=> 0x80000014 <stop+0>:    73 00 50 10      wfi
(gdb) n
```

接著來寫 kernel

因為還不能用 printf，先將 hello.c 改為 while 就好

```
C hello.c > main(void)
1 void main(void)
2 {
3
4     while (1) {}
5 }
```

然後在開機程式的最後一行加上跳到 main 的指令

```
add sp, sp, t0    #將t0(當前CPU記憶體起始位置)加上
j main            #跳到kernel(hello.c)中的main
```

用 gdb 測試，成功跳到 main 了!!

```
_start () at start.S:21
21      add sp, sp, t0      #t0(CPU)sp()=CPU
=> 0x80000010 <_start+16>:  33 01 51 00      add      sp,sp,t0
(gdb) n
22      j main              #kernel(hello.c)main
=> 0x80000014 <_start+20>:  6f 20 c0 00      j          0x80002020 <main>
(gdb) n
main () at hello.c:2
2      {
=> 0x80002020 <main+0>:  13 01 01 ff      addi      sp,sp,-16
    0x80002024 <main+4>:  23 26 81 00      sw        s0,12(sp)
    0x80002028 <main+8>:  13 04 01 01      addi      s0,sp,16
(gdb) n
```


接著來寫 uart 驅動程式

```
C uart.c > ...
1 //首先，我們先把會用到的uart暫存器位址define成好用的名字
2
3 //Interrupt Enable Register 是用來設定中斷模式的暫存器
4 #define IER ((volatile unsigned char *) (0x10000000L + 1))
5
6 //Line Control Register 用來控制傳輸的速率、校驗方式、字節長度
7 #define LCR ((volatile unsigned char *) (0x10000000L + 3))
8
9 //divisor latch less 用來設定最小傳輸速率的暫存器(與Receive Holding Register共用同個位址，只在LCR的第8個bit設為1時才可存取)
10 #define DLL ((volatile unsigned char *) (0x10000000L + 0))
11
12 //divisor latch most 用來設定最大傳輸速率的暫存器(與Interrupt Enable Register共用同個位址，只在LCR的第8個bit設為1時才可存取)
13 #define DLM ((volatile unsigned char *) (0x10000000L + 1))
14
15 //Transmit Holding Register用來存放要送出的資料的暫存器，與Receive Holding Register、DLL共用同個位址
16 #define THR ((volatile unsigned char *) (0x10000000L + 0))
17
18 //Line Status Register 存放uart傳輸狀態的暫存器
19 #define LSR ((volatile unsigned char *) (0x10000000L + 5))
20
21 //將uart初始化的函數
22 int uart_init()
23 {
24
25     *IER = 0x00; //存取IER，把所有中斷關掉
26     *LCR = *LCR | (1 << 7); //存取LCR，與10000000(二進位)做or運算，把最左邊的bit設為1，打開DLL、DLM的設定模式
27     *DLL = 0x03; //將DLL設為3，代表38.4k
28     *DLM = 0x00; //DLM則不設定上限
29
30     *LCR = 0; //設定完DLL、DLM，把LCR全部設為0，關閉速率設定模式
31     *LCR = *LCR | (3 << 0); //將LCR設為3(二進位00000011)，代表每次傳輸字長8個bit
32
33 }
34 //將傳輸資料寫入THR的函數
35 int uart_c(char ch)
36 {
37     while(*LSR & (1 << 5) == 0); //判斷LSR中記錄傳輸暫存器是否為空的bit，當它為1時離開迴圈往下執行
38     return *THR = ch; //將資料寫入THR
39 }
40 //將一整串字串依序寫入THR的函數
41 int uart_puts(char* s)
42 {
43     while(*s) //當字串指針所指的位元值不為0時(代表還有字沒傳完)就持續執行
44     {
45         uart_c(*s++); //使用uart_c函數將當前指到的字寫入THR，再將指針往下一個字移動
46     }
47 }
```

回到我們的 hello.c

使用剛寫好的 uart 驅動函數來輸出文字

```
C hello.c > main(void)
1 extern void uart_init(void); //extern void 告訴編譯器 uart_init(void) 函數在其他檔案已被宣告
2 extern void uart_puts(char* s); //extern void 告訴編譯器 uart_puts(char* s) 函數在其他檔案已被宣告
3
4 void main(void)
5 {
6     uart_init(); //執行uart_init函數初始化UART
7     uart_puts("hello\n"); //使用uart_puts函數傳輸 hello 字串
8     while (1) {}; //進入迴圈
9 }
```

make 完用 qemu 執行看看

```
user@DESKTOP-6RDGTQ7 MINGW64 ~/Downloads/測試/OS/作業4-2 (main)
$ make qemu
qemu-system-riscv32 -nographic -smp 1 -machine virt -bios none -kernel a.out
hello
```

成功!! 接著用 gdb 執行看看

```
user@DESKTOP-6RDGTQ7 MINGW64 ~/Downloads/測試/OS/作業4-2 (main)
$ make gdb
qemu-system-riscv32 -nographic -smp 1 -machine virt -bios none -kernel a.out -s -S &
riscv64-unknown-elf-gdb a.out -q -x gdbinit
Reading symbols from a.out...
Breakpoint 1 at 0x80000000: file start.S, line 4.
0x00001000 in ?? ()
=> 0x00001000: 97 02 00 00      auipc    t0,0x0

Breakpoint 1, _start () at start.S:4
4      csrr t0, mhartid      #hartidt0
=> 0x80000000 <_start+0>:  f3 22 40 f1      csrr     t0,mhartid
(gdb) n
5      bnez t0, stop         #0CPUID0stop
=> 0x80000004 <_start+4>:  63 9e 02 00      bnez     t0,0x80000020 <stop>
(gdb) n
8      slli t0, t0, 10       #ID0CPU0CPU1024
=> 0x80000008 <_start+8>:  93 92 a2 00      slli     t0,t0,0xa
(gdb) n
17     la sp, stacks         #stackslabelstackslastackssp
=> 0x8000000c <_start+12>:  17 01 00 00      auipc    sp,0x0
0x80000010 <_start+16>:  13 01 c1 01      addi     sp,sp,28 # 0x80000028 <stacks>
(gdb) n
_start () at start.S:18
18     addi sp, sp, 1024     #sp1024
=> 0x80000014 <_start+20>:  13 01 01 40      addi     sp,sp,1024
(gdb) n
_start () at start.S:22
22     add sp, sp, t0       #t0(CPU)sp()=CPU
=> 0x80000018 <_start+24>:  33 01 51 00      add      sp,sp,t0
(gdb) n
23     j main               #kernel(hello.c)main
=> 0x8000001c <_start+28>:  6f 20 80 13      j        0x80002154 <main>
(gdb) n
main () at hello.c:5
warning: Source file is more recent than executable.
5      {
=> 0x80002154 <main+0>:  13 01 01 ff      addi     sp,sp,-16
0x80002158 <main+4>:  23 26 11 00      sw       ra,12(sp)
0x8000215c <main+8>:  23 24 81 00      sw       s0,8(sp)
0x80002160 <main+12>:  13 04 01 01      addi     s0,sp,16
(gdb) n
6      uart_init();        //uart_initUART
=> 0x80002164 <main+16>:  ef f0 5f ec      jal      ra,0x80002028 <uart_init>
(gdb) n
7      uart_puts("hello\n"); //uart_puts hello
=> 0x80002168 <main+20>:  b7 27 00 80      lui      a5,0x80002
0x8000216c <main+24>:  13 85 87 17      addi     a0,a5,376 # 0x80002178
0x80002170 <main+28>:  ef f0 1f f9      jal      ra,0x80002100 <uart_puts>
(gdb) n
hello
8      while (1) {};        //
=> 0x80002174 <main+32>:  6f 00 00 00      j        0x80002174 <main+32>
(gdb) n
```

练习 4-3

自学 Makefile 的语法，理解在 `riscv-operating-system-mooc` 仓库的根目录下执行 `make` 会发生什么。

為避免排版上篇幅過長，我把我對命令的理解直接打成註釋

作業4-3 > Makefile

```
1  #設定一個存放資料夾路徑的變數
2  SECTIONS = \
3      code/asm \
4      code/os \
5
6
7  #告訴make,當沒有特別提到要做什么事情時,預設要執行的任務
8  .DEFAULT_GOAL := all
9
10
11 #宣告all任務要做的事
12 all :
13 #   echo的意思是顯示echo這條命令本身及其輸出結果,在前面加一個@的意思是不要顯示命令本身,只顯示結果
14 #   @echo "begin compile ALL exercises for assembly samples ....."
15
16 #   for dir的意思是說,對SECTIONS變數中儲存的每個路徑執行do之後描述的動作 $(MAKE)代表make這個命令
17 #   $$dir 第一個$代表擴展的功能,意思是說,make會把指令處理成
18 for dir in code/os; do make -C $dir || exit "$?"; done 再交給shell處理,這樣就能取用shell的 $dir、$?變數
19 #   所以就是對每個資料夾中的.c檔進行make,done就是等全部做完再進行下一步
20 for dir in $(SECTIONS); do $(MAKE) -C $$dir || exit "$?"; done
21 #   顯示echo命令中所包含的語句
22 @echo "compile ALL exercises finished successfully! ....."
23
24 #告訴make clean是偽目標,不要當作編譯來執行
25 .PHONY : clean
26
27 #設定一個clean偽目標,對每個資料夾執行專屬於它的clean偽目標
28 clean:
29     for dir in $(SECTIONS); do $(MAKE) -C $$dir clean || exit "$?"; done
30
31
32 #告訴make slides目標是偽目標,不要當作編譯來執行
33 .PHONY : slides
34
35 #設定一個slides偽目標,刪除所有slides資料夾中的PDF,用soffice指令將 ./docs/ppts/路徑下的PPT檔轉為PDF,放入 ./slides資料夾中
36 slides:
37     rm -f ./slides/*.pdf
38     soffice --headless --convert-to pdf:writer_pdf_Export --outdir ./slides ./docs/ppts/*.pptx
39
```