1 实验项目

1.1 项目名称

环境配置

1.2 实验目的

1) 安装、配置实验环境

2) 创建一个独立于操作系统的可执行程序（裸机程序）

3) 学会利用构建系统构建工程

4) 了解并掌握 GDB 简单调试方法

5) 为后续实验做好准备

1.3 实验资源

实验指导书 https://os2024lab.readthedocs.io/zh-cn/latest/

2 实验任务

2.1 安装工具链

（一）安装交叉编译工具链 (aarch64)

1、下载工具链

2、解压工具链

3、重命名工具链目录

下载工具链，以下载工具链版本为 11.2，宿主机为 x86 64 位 Linux 机器为例

$wget

https://developer.arm.com/-/media/Files/downloads/gnu/11.2-2022.02/binrel/gcc-arm-11.2-2022.02-x86_64-aarch64-none-elf.tar.xz

$ tar -xf gcc-arm-(按 Tab 键补全)

$ mv gcc-arm-(按 Tab 键补全) aarch64-none-elf

4、将目录/.../aarch64-none-elf/bin（可用 pwd 查看）加入到环境变量 PATH 中

export PATH="/home/xiaoye/OSlab/aarch64-none-elf/bin:$PATH"

5、测试工具链是否安装成功

$ aarch64-none-elf-gcc --version

```
(base) xiaoye@长乐:~/OSlab$ aarch64-none-elf-gcc --version
aarch64-none-elf-gcc (GNU Toolchain for the Arm Architecture 11.2-2022.02 (arm-11.14)) 11.2.1 20220111
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

(二) 安装 QEMU 模拟器

$ sudo apt-get update

$ sudo apt-get install qemu

$ sudo apt-get install qemu-system

（三）安装 CMake

$ sudo apt-get install cmake

2.2 创建裸机(BareMetal)程序

（一） 创建项目

【裸机程序】

裸机程序（Bare-Metal Program）是指直接运行在计算机硬件上，不依赖操作系统（OS）的软件程序。它绕过了操作系统的抽象层，直接与硬件交互。

【目录结构】

src 目录：所有源代码均放在此处；

bsp 目录：存放与硬件紧密相关的代码；

include 目录：存放项目的大部分头文件。

1、 在 src/下创建 main.c

```c
#include "prt_typedef.h"

#define UART_REG_WRITE(value, addr)  (*(volatile U32 *)((uintptr_t)addr) = (U32)value)
char out_str[] = "AArch64 Bare Metal";

S32 main(void)
{
    int length = sizeof(out_str) / sizeof(out_str[0]);

    // 逐个输出字符
    for (int i = 0; i < length - 1; i++) {
        UART_REG_WRITE(out_str[i], 0x9000000);
    }
}
```

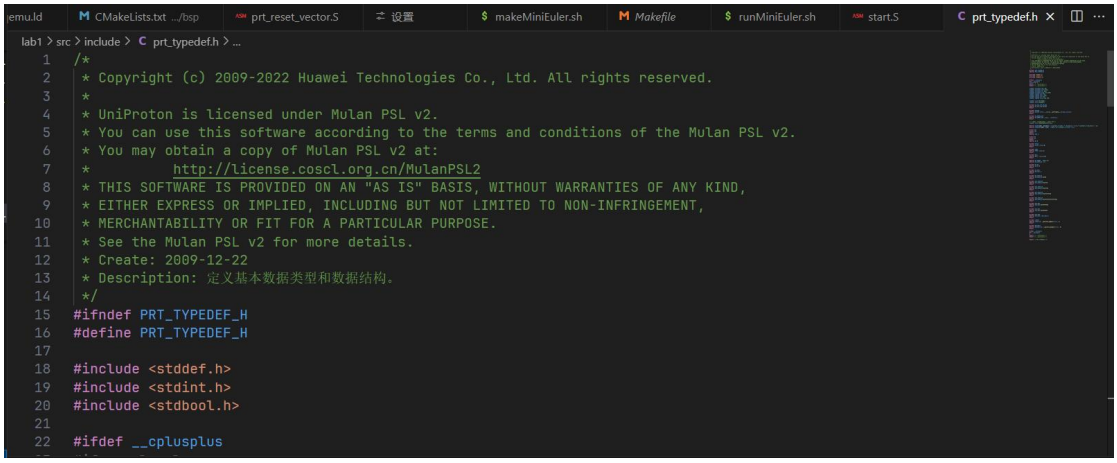
【注】:

（1）S32 是在 prt_typedef.h 中定义的基本类型，这是为了屏蔽各硬件系统的区别，方便操作系统移植到多种不同规格的硬件上；

（2）main 函数的主要功能（L12-L13）：out_str 中的字符通过宏 UART_REG_WRITE 逐个写入地址 0x9000000 的地方。其作用将在实验二 Hello,miniEuler 部分详细解释。

2、在 src/include/下创建 prt_typedef.h

https://os2024lab.readthedocs.io/zh-cn/latest/_static/prt_typedef.h



3、在 src/bsp/下创建 start.S 和 prt_reset_vector.S 两个文件

start.S 源码

```
1     .global   OsEnterMain
2     .extern __os_sys_sp_end
3
4     .type     start, function
5     .section  .text.bspinit, "ax"
6     .balign   4
7
8     .global OsElxState
9     .type   OsElxState, @function
10  OsElxState:
11      MRS    x6, CurrentEL // 把系统寄存器 CurrentEL 的值读入到通用寄存器 x6 中
12      MOV    x2, #0x4 // CurrentEL EL1:  bits [3:2] = 0b01
13      CMP    w6, w2
14
15      BEQ Start // 若 CurrentEl 为 EL1 级别，跳转到 Start 处执行，否则死循环。
16
17  OsEl2Entry:
18      B OsEl2Entry
19
20  Start:
21      LDR    x1, =__os_sys_sp_end // 符号在ld文件中定义
22      BIC    sp, x1, #0xf // 设置栈指针, BIC: bit clear
23
24      B      OsEnterMain
25
26  OsEnterReset:
27      B      OsEnterReset
```

【注】:

（1）L1，L2 两行声明 OsEnterMain 和__os_sys_sp_end 是外部定义的符号，其中 OsEnterMain 在 prt_reset_vector.S 中定义， __os_sys_sp_end 在链接脚本 aarch64-qemu.ld 定义。

（2）L5 声明这部分代码段(section)的名字是 .text.bspinit

（3）L10 为系统入口，即系统一启动就会执行从 L10 开始的代码，其原因在随后的链接脚本中说明。

（4）L11-L15 检测当前 CPU 的 Exception Level 是否为 EL1（将在 实验四 异常处理 部分详细解释），如果是 EL 则通过 L15 的 BEQ Start 跳转到标号 Start(L20)处开始执行，否则执行 L17 开始的指令，它和 L18 一起构成死循环。

（5）L11 中的 CurrentEL 是 AArch64 架构的系统寄存器。这些寄存器不能直接操作，需要通过 MRS 指令（把系统寄存器的值读入到通用寄存器）或 MSR 指令（把通用寄存器的值写入到系统寄存器）借助通用寄存器来访问。

（6）L21-L22 用链接文件定义的地址初始化栈指针 sp，然后 L24 跳转到 prt_reset_vector.S 的 L7 行 OsEnterMain 处开始执行。

prt_reset_vector.S 源码

```
1   DAIF_MASK = 0x1C0       // disable SError Abort, IRQ, FIQ
2
3       .global  OsVectorTable
4       .global  OsEnterMain
5
6       .section .text.startup, "ax"
7   OsEnterMain:
8       BL      main
9
10      MOV     x2, DAIF_MASK // bits [9:6] disable SError Abort, IRQ, FIQ
11      MSR     DAIF, x2 // 把通用寄存器 x2 的值写入系统寄存器 DAIF 中
12
13  EXITLOOP:
14      B EXITLOOP
```

【注】:

（1）目前，完全可以把 start.S 和 prt_reset_vector.S 合成一个文件，但为了将来扩展且与 UniProton 保持一致选择保留 2 个文件。

（2）L8 行跳转到通过 BL main 跳转到 main.c 中的 main 函数执行，main 函数执行完后会回到 L10 继续执行。

（3）L10-L11 禁用了 Debug、SError、IRQ 和 FIQ，因为中断处理尚未设置，详细参见 实验四 异常处理

（4）L10 中的 DAIF 是 AArch64 架构的系统寄存器，完整的寄存器列表可参考 Arm 官网的 AArch64 System Registers 页面。

（5）L13-L14 进入死循环。

（6）在上面两个汇编文件中出现了两种不同的跳转指令 B 和 BL，其中 B 跳转后不返回调用位置，BL 跳转后执行完函数后会回到调用位置继续执行。

4、在 src/下创建链接文件 aarch64-qemu.ld

```
1  ENTRY(__text_start)
2
3  _stack_size = 0x10000;
4  _heap_size = 0x10000;
5
6  MEMORY
7  {
8      IMU_SRAM (rwx) : ORIGIN = 0x40000000, LENGTH = 0x800000 /* 内存区域 */
9      MMU_MEM (rwx) : ORIGIN = 0x40800000, LENGTH = 0x800000 /* 内存区域 */
10 }
11
12 SECTIONS
13 {
14     text_start = .;
15     .start_bspinit :
16     {
17         __text_start = .; /* __text_start 指向当前位置， "." 表示当前位置 */
18         KEEP(*(.text.bspinit))
19     } > IMU_SRAM
20
21     ... ... ...
22
23     .heap (NOLOAD) :
24     {
25         . = ALIGN(8);
26         PROVIDE (__HEAP_INIT = .);
27         . = . + _heap_size; /* 堆空间 */
28         . = ALIGN(8);
29         PROVIDE (__HEAP_END = .);
30     } > IMU_SRAM
31
32     .stack (NOLOAD) :
33     {
34         . = ALIGN(8);
35         PROVIDE (__os_sys_sp_start = .);
36         . = . + _stack_size; /* 栈空间 */
37         . = ALIGN(8);
38         PROVIDE (__os_sys_sp_end = .);
39     } > IMU_SRAM
40     end = .;
41
42     ... ... ...
43 }
```

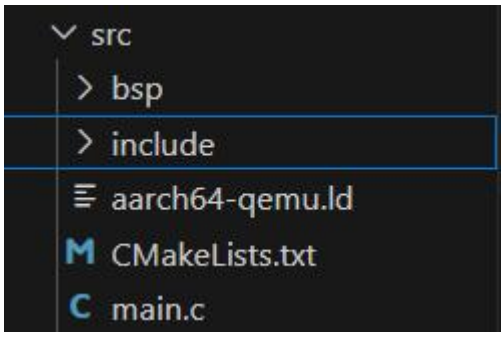
【注】:

（1）L1 的 ENTRY(__text_start)中指明系统入口为 __text_start 。L17-L18 表明 __text_start 为 .text.bspinit 段的起始位置。而在 start.S 中 L5 处定义了 .text.bspinit 段，其入口为 L10 处的 OsElxState 标号。因此系统的入口实际上就是 start.S 中的 L10 处的 OsElxState 标号处。

（2）链接脚本中通过 PROVIDE 定义的符号 __os_sys_sp_end 是全局符号，可以在程序中使用（如 start.s 中），其定义的是栈底的位置。

（3）L26-L29，L35-L38 处分别定义了堆空间和栈空间。

【目录概览】



2.3 工程构建

（一） CMakeLists.txt

1、 src/下的 CMakeLists.txt

```
cmake_minimum_required(VERSION 3.12)

set(CMAKE_SYSTEM_NAME "Generic") # 目标系统(baremental): cmake/tool_chain/uniproton_tool_chain_gcc_arm64.cmake 写的是 Linux
set(CMAKE_SYSTEM_PROCESSOR "aarch64") # 目标系统 CPU

set(TOOLCHAIN_PATH "/usr/local/aarch64-none-elf") # 修改为交叉工具链实际所在目录 build.py config.xml 中定义
set(CMAKE_C_COMPILER ${TOOLCHAIN_PATH}/bin/aarch64-none-elf-gcc)
set(CMAKE_CXX_COMPILER ${TOOLCHAIN_PATH}/bin/aarch64-none-elf-g++)
set(CMAKE_ASM_COMPILER ${TOOLCHAIN_PATH}/bin/aarch64-none-elf-gcc)
set(CMAKE_LINKER ${TOOLCHAIN_PATH}/bin/aarch64-none-elf-ld)

# 定义编译和链接等选项
set(CC_OPTION "-Os   -Wformat-signedness    -Wl,--build-id=none   -fno-PIE -fno-PIE --specs=nosys.specs -fno-builtin -fno-dwarf2-cfi-asm -fomit-frame-pointer -fzero-initialized-in-bss -fdollars-in-identifiers -ffunction-sections -fdata-sections -fno-aggressive-loop-optimizations -fno-optimize-strlen -fno-schedule-insns -fno-inline-small-functions -fno-inline-functions-called-once -fno-strict-aliasing -finline-limit=20  -mlittle-endian -nostartfiles -funwind-tables")
set(AS_OPTION "-Os   -Wformat-signedness    -Wl,--build-id=none   -fno-PIE -fno-PIE --specs=nosys.specs -fno-builtin -fno-dwarf2-cfi-asm -fomit-frame-pointer -fzero-initialized-in-bss -fdollars-in-identifiers -ffunction-sections -fdata-sections -fno-aggressive-loop-optimizations -fno-optimize-strlen
```

```
-fno-schedule-insns -fno-inline-small-functions -fno-inline-functions-called-once
-fno-strict-aliasing -finline-limit=20  -mlittle-endian -nostartfiles
-funwind-tables")
15set(LD_OPTION " ")
16set(CMAKE_C_FLAGS "${CC_OPTION} ")
17set(CMAKE_ASM_FLAGS "${AS_OPTION} ")
18set(CMAKE_LINK_FLAGS "${LD_OPTION} -T
${CMAKE_CURRENT_SOURCE_DIR}/aarch64-qemu.ld") # 指定链接脚本
19set(CMAKE_EXE_LINKER_FLAGS "${LD_OPTION} -T
${CMAKE_CURRENT_SOURCE_DIR}/aarch64-qemu.ld") # 指定链接脚本
20set (CMAKE_C_LINK_FLAGS " ")
21set (CMAKE_CXX_LINK_FLAGS " ")
22
23set(HOME_PATH ${CMAKE_CURRENT_SOURCE_DIR})
24
25set(APP "miniEuler") # APP 变量，后面会用到 ${APP}
26project(${APP} LANGUAGES C ASM) # 工程名及所用语言
27set(CMAKE_BUILD_TYPE Debug) # 生成 Debug 版本
28
29include_directories( # include 目录
30    ${CMAKE_CURRENT_SOURCE_DIR}/include
31    ${CMAKE_CURRENT_SOURCE_DIR}/bsp
32)
33
34add_subdirectory(bsp) # 包含子文件夹的内容
35
36list(APPEND OBJS $<TARGET_OBJECTS:bsp>)
37add_executable(${APP} main.c ${OBJS}) # 可执行文件
```

【注】:

修改 set(TOOLCHAIN_PATH “/usr/local/aarch64-none-elf”) 中的目录为：

```
set(TOOLCHAIN_PATH "/home/xiaoye/OS/aarch64-none-elf")
```

2、 src/bsp/下的 CMakeLists.txt

```
1set(SRCS start.S prt_reset_vector.S )
2add_library(bsp OBJECT ${SRCS})  # OBJECT 类型只编译生成.o 目标文件，但不实际链接成库
```

【注】: L36-L37 中指明需链接的目标对象 ${OBJS} 包括 $<TARGET_OBJECTS:bsp>，而 $<TARGET_OBJECTS:bsp> 在 src/bsp/下的 CMakeLists.txt 中定义。这样 main.c、prt_reset_vector.S、start.S 都将被包含在可执行文件中。

（二） 编译运行

1.编译
首先在项目目录 lab1 下创建 build 目录用于编译生成，然后进入 build 目录执行。

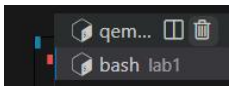
```
$ cmake ../src
$ cmake --build .
```

2.执行
在项目目录 lab1 下执行

```
$ qemu-system-aarch64 -machine virt -m 1024M -cpu cortex-a53 -nographic -kernel build/miniEuler  -s
```

```
(base) xiaoye@长乐:~/OS/lab1$ qemu-system-aarch64 -machine virt -m 1024M -cpu cortex-a53 -nographic -kernel build/miniEuler  -s
qemu-system-aarch64: -s: Failed to find an available port: Address already in use
```

问题：QEMU 尝试在 1234 端口启动 GDB 服务器失败，因为该端口已被其他进程占用。解决：关闭其他进程。



输出：

```
(base) xiaoye@长乐:~/OS/lab1$ qemu-system-aarch64 -machine virt -m 1024M -cpu cortex-a53 -nographic -kernel build/miniEuler  -s
AArch64 Bare Metal
```

3.退出 Qemu
在 qemu 中，输入 crtl+a 抬起后，再输入 x.
【注】：CMake 项目的构建过程可以分为**配置**、**生成**和**编译**三个核心阶段。

`cmake --build` 是 CMake 构建流程的核心命令，推荐始终通过它调用构建系统。其优势在于：

- 统一不同生成器（如 Makefile、Ninja）的操作接口。
- 支持并行编译、自定义目标、配置切换等高级功能。
- 避免直接操作生成器命令（如 `make` ）的兼容性问题。

对于嵌入式开发（如裸机程序），可结合交叉编译工具链，通过 `cmake --build` 一键生成目标平台的可执行文件。

2.4 调试支持

（一）GDB 简单调试方法

1. 通过 QEMU 运行程序并启动调试服务器，默认端口 1234

$ qemu-system-aarch64 -machine virt,gic-version=2 -m 1024M -cpu cortex-a53 -nographic -kernel build/miniEuler -s -S

【注】：
qemu 的参数说明：

查看相关参数的作用可在命令行执行：qemu-system-aarch64 --help

（1）qemu-system-aarch64：启动 QEMU 模拟器，模拟 ARMv8/aarch64 架构的 CPU 和硬件环境。

（2）-machine virt,gic-version=2：（-machine virt 选择 QEMU 的通用虚拟化机器类型（virt））（gic-version=2 指定 Generic Interrupt Controller（GIC）版本为 2）

（3）-m 1024M：分配 1GB 物理内存给虚拟机。

（4）-cpu cortex-a53：指定虚拟 CPU 型号为 Cortex-A53。

（5）-nographic：禁用图形界面，将串口输出重定向到终端。

（6）-kernel build/miniEuler：指定客户机的内核镜像文件为 build/miniEuler。

（7）-s：在 1234 端口启动 GDB 服务器。

（8）-S：启动时冻结 CPU，等待 GDB 连接后再继续执行。

```
(base) xiaoye@长乐:~/OS/lab1$ qemu-system-aarch64 -machine virt,gic-version=2 -m 1024M -cpu cortex-a53 -nographic -kernel build/miniEuler  -s
  -S
```

【注】：-S 启动后，需要在另一个终端启动 GDB 并连接。

2. 启动调试客户端

$ aarch64-none-elf-gdb build/miniEuler

```
(base) xiaoye@长乐:~/OS/lab1$ aarch64-none-elf-gdb build/miniEuler
GNU gdb (GNU Toolchain for the Arm Architecture 11.2-2022.02 (arm-11.14)) 11.2.90.20220202-git
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=x86_64-pc-linux-gnu --target=aarch64-none-elf".
Type "show configuration" for configuration details.
```

3. 设置调试参数，开始调试

(gdb) target remote localhost:1234

(gdb) disassemble

(gdb) n

```
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
__text_start () at /home/xiaoye/OS/lab1/src/bsp/start.S:11
Python Exception <class 'UnicodeDecodeError'>: 'utf-8' codec can't decode byte 0xb0 in position 229: invalid start byte
11          MRS     x6, CurrentEL // ◆◆∈T◆Ĵ◆◆◆ CurrentEL ◆◆◆◆◆Ø◆Ø◆ŭĴ◆◆◆ x6 ◆◆
```

问题：GDB 在调试过程中尝试使用 UTF-8 编码解析源文件（start.S），但该文件实际编码为 GBK，导致在位置 229 字节处遇到无法解码的字节 0xb0。

解决：（1）GDB 和终端的编码环境设置为 UTF-8

```
(base) xiaoye@长乐:~/OS/lab1$ locale
LANG=C.UTF-8
LANGUAGE=
LC_CTYPE="C.UTF-8"
LC_NUMERIC="C.UTF-8"
LC_TIME="C.UTF-8"
LC_COLLATE="C.UTF-8"
LC_MONETARY="C.UTF-8"
LC_MESSAGES="C.UTF-8"
                                                                    行 12，列 1  空格: 4  UTF-8  LF  纯文本
```

（2）将源文件的编码转成 UTF-8

用 vs code 打开，可以直接更改保存文件的编码标识。

（3）重新编译运行，再开始调试

必须清理构建产物，确保生成最新的可执行文件。这里使用自动化脚本。

```
(base) xiaoye@长乐:~/OS/lab1$ sh makeMiniEuler.sh
mkdir: cannot create directory ‘build’: File exists
-- The C compiler identification is GNU 11.2.1
-- The ASM compiler identification is GNU
-- Found assembler: /home/xiaoye/OS/aarch64-none-elf/bin/aarch64-none-elf-gcc
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /home/xiaoye/OS/aarch64-none-elf/bin/aarch64-none-elf-gcc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Configuring done (2.1s)
-- Generating done (0.0s)
-- Build files have been written to: /home/xiaoye/OS/lab1/build
[ 25%] Building ASM object bsp/CMakeFiles/bsp.dir/start.S.obj
[ 50%] Building ASM object bsp/CMakeFiles/bsp.dir/prt_reset_vector.S.obj
[ 50%] Built target bsp
[ 75%] Building C object CMakeFiles/miniEuler.dir/main.c.obj
[100%] Linking C executable miniEuler
[100%] Built target miniEuler
(base) xiaoye@长乐:~/OS/lab1$ sh runMiniEuler.sh
qemu-system-aarch64 -machine virt,gic-version=2 -m 1024M -cpu cortex-a53 -nographic -kernel build/miniEuler -s
qemu-system-aarch64: -s: Failed to find an available port: Address already in use
(base) xiaoye@长乐:~/OS/lab1$ sh runMiniEuler.sh
qemu-system-aarch64 -machine virt,gic-version=2 -m 1024M -cpu cortex-a53 -nographic -kernel build/miniEuler -s
AArch64 Bare Metal
```

开始调试：

```
Reading symbols from build/miniEuler...
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0x0000000040000000 in __text_start ()
(gdb) disassemble
Dump of assembler code for function __text_start:
=> 0x0000000040000000 <+0>:     mrs     x6, currentel
   0x0000000040000004 <+4>:     mov     x2, #0x4                        // #4
   0x0000000040000008 <+8>:     cmp     w6, w2
   0x000000004000000c <+12>:    b.eq    0x40000014 <Start>  // b.none
End of assembler dump.
(gdb) n
Single stepping until exit from function __text_start,
which has no line number information.
0x0000000040000014 in Start ()
(gdb)
```

已为屏幕阅读器优化　行 14，列 2　空格: 4　UTF-8　LF　{} C　Linux

（二）　将调试集成到 vscode

1、打开 main.c，点击运行和调试按钮，创建 launch.json 文件，增加配置；

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "aarch64-gdb",
            "type": "cppdbg",
            "request": "launch",
            "program": "${workspaceFolder}/build/miniEuler",
```

//路径使用绝对路径，确保 program 路径与实际生成的 miniEuler 文件完全一致

program: “/home/xiaoye/OSlab/lab1/build/miniEuler”

```
            "stopAtEntry": true,
            "cwd": "${fileDirname}",
            "environment": [],
            "externalConsole": false,
            "launchCompleteCommand": "exec-run",
            "MIMode": "gdb",
            "miDebuggerPath":
"/usr/local/aarch64-none-elf/bin/aarch64-none-elf-gdb", // 修改成交叉调试器 gdb 对应位置
```

```
/home/xiaoye/OSlab/aarch64-none-elf/bin/aarch64-none-elf-gdb
            "miDebuggerServerAddress": "localhost:1234",
            "setupCommands": [
                {
                    "description": "Enable pretty-printing for gdb",
                    "text": "-enable-pretty-printing",
                    "ignoreFailures": true
                }
            ]
        }
    ],

}
```

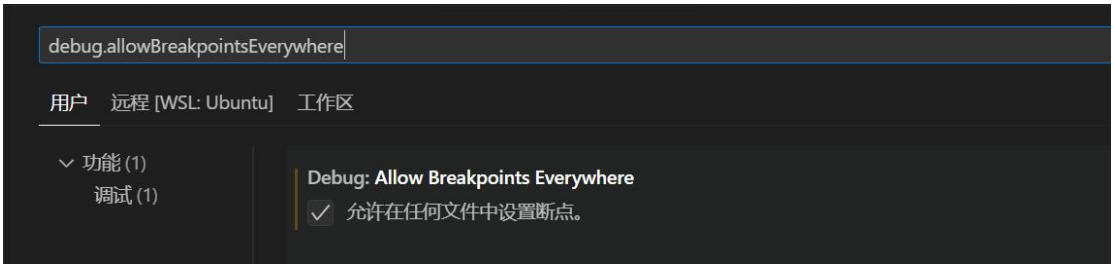
2、选择 aarch64-gdb ，开始调试（F5），在调试控制台执行 gdb 命令.
//在调试之前，需要通过 QEMU 运行程序并启动调试服务器，默认端口 1234。

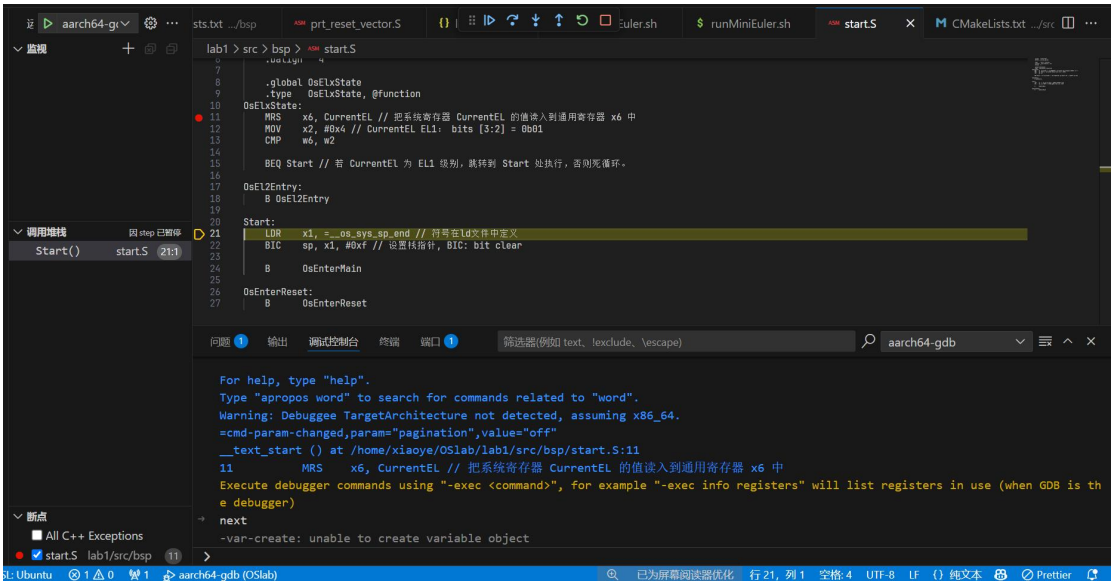
```
$1 = 0x40000014
-exec set $x24 = 0x5

-exec set {int}0x4000000 = 0x1
=memory-changed,thread-group="i1",addr="0x0000000004000000",len="0x4"
 -exec set *((volatile int *) 0x08010004) = 0x1
-var-create: unable to create variable object
-exec q
Error: GDB exited unexpectedly with exit code 0 (0x0).
The program '/home/xiaoye/OSlab/lab1/build/miniEuler' has exited with code 0 (0x00000000).
```

【注】：
（1）查看指定地址的内存内容。在调试控制台执行 -exec x/20xw 0x40000000 即可，其中 x 表示查看命令，20 表示查看数量，x 表示格式，可选格式包括 Format letters are o(octal), x(hex), d(decimal), u(unsigned decimal),t(binary), f(float), a(address), i(instruction), c(char) and s(string).Size letters are b(byte), h(halfword), w(word), g(giant, 8 bytes)，最后的 w 表示字宽，b 表示单字节，h 表示双字节，w 表示四字节，g 表示八字节。还可以是指令：
-exec x/20i 0x40000000; 字符串：-exec x/20s 0x40000000
（2）显示所有寄存器。-exec info all-registers
（3）查看寄存器内容。-exec p/x $pc
（4）修改寄存器内容。-exec set $x24 = 0x5
（5）修改指定内存位置的内容。-exec set {int}0x4000000 = 0x1 或者 -exec set *((int *) 0x4000000) = 0x1
（6）修改指定 MMIO 寄存器的内容。-exec set *((volatile int *) 0x08010004) = 0x1
（7）退出调试 -exec q
总之，可以通过 -exec 这种方式可以执行所有的 gdb 调试指令。

3、如需调试 .s 文件，需在 vscode 中打开允许在任何文件中设置断点选项。





4、QEMU 执行结果

```
(base) xiaoye@长乐:~/OSlab/lab1$ qemu-system-aarch64 -machine virt -m 1024M -cpu cortex-a53 -nographic -kernel build/miniEuler -s -S
AArch64 Bare Metal
```

2.5 自动化脚本

1、在 lab1 下新建 makeMiniEuler.sh 脚本来编译项目，新建 runMiniEuler.sh 脚本来运行项目

```
lab1 > $ runMiniEuler.sh
1   #sh runMiniEuler.sh 直接运行
2   #sh runMiniEuler.sh -S 启动后在入口处暂停等待调试
3
4   echo qemu-system-aarch64 -machine virt,gic-version=2 -m 1024M -cpu cortex-a53 -nographic -kernel build/miniEuler  -s $1
5
6   qemu-system-aarch64 -machine virt,gic-version=2 -m 1024M -cpu cortex-a53 -nographic -kernel build/miniEuler  -s $1
```

2、之后编译及运行程序只需要执行：

$ sh makeMiniEuler.sh

$ sh runMiniEuler.sh

【注】：

（1）运行 sh makeMiniEuler.sh -v 将会显示实际执行的编译指令，方便了解编译的过程并查找编译错误原因。

（2）运行 sh runMiniEuler.sh -S 将在程序启动后在入口处暂停等待调试，此时可通过 aarch64-none-elf-gdb 或 vscode 连入调试服务器。

2.6 Lab1 作业

1、作业 1

请操作 NZCV 寄存器获取 start.S 中执行 *CMP w6, w2* 前后 NZCV 寄存器的变化。

（1）在 ARM 架构中，CPSR（Current Program Status Register）是 32 位的条件标志寄存器。它的位布局如下：

位 31	位 30	位 29	位 28	位 27	. . .	位 8	位 7	位 6 ~ 0
N	Z	C	V	Q	. . .	I	F	保留

条件标志位（N/Z/C/V）位于**高 4 位**（位 31~28）：

N （Negative）：位 31，表示运算结果为负。

Z （Zero）：位 30，表示运算结果为零。

C （Carry）：位 29，表示进位或借位（如加法溢出）。

V （Overflow）：位 28，表示有符号数溢出。

其他控制位　　：

Q　（Saturation）：位 27，表示饱和运算状态。

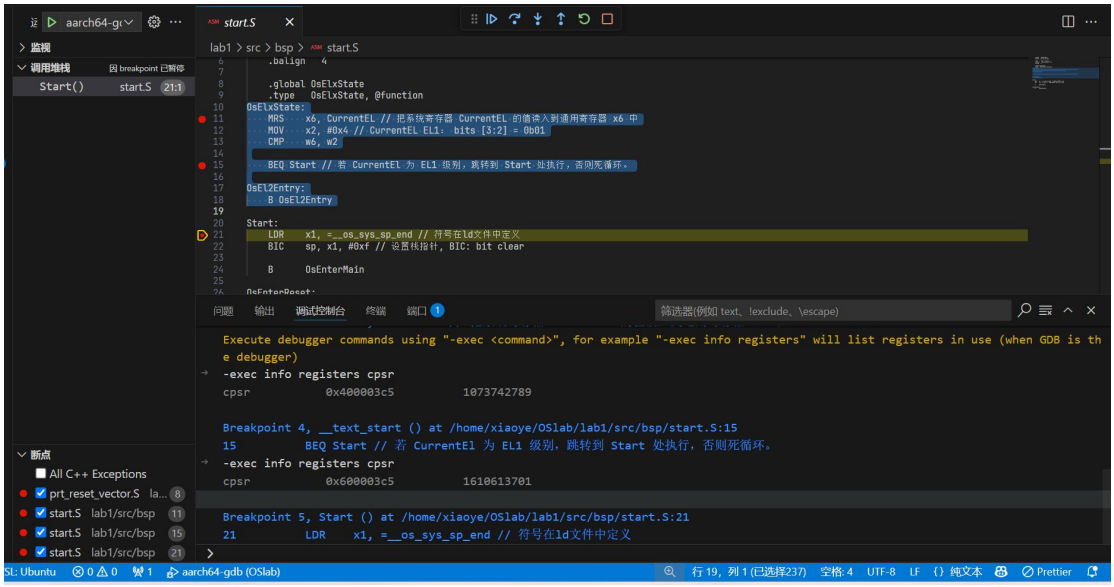
I　（IRQ 禁用）：位 8，控制 IRQ 中断。

F　（FIQ 禁用）：位 7，控制 FIQ 中断。

位 6~0：保留或未使用。

（2）调试并查看执行 CMP w6, w2 前后寄存器的变化

-exec info registers cpsr



（1）　执行之前：0x400003c5，其中高 4 位是 0100，初始时，N=0,Z=1,C=0,V=0。

（2）　执行之后，0x600003c5，其中高 4 位是 0110，表示结果：

N=0　：结果非负。

Z=1　：结果为零（w6 == w2）。

C=1　：无符号比较，w6 >= w2（未发生借位）。

V=0　：无符号溢出未发生。

根据 Z=1（表示 w6 == w2（即 CurrentEL == 0x4）），BEQ 跳转到 Start。

2、作业 2

商业操作系统都有复杂的构建系统，试简要分析 UniProton 的构建系统。

UniProton 通过在根目录下执行 python build.py m4 （m4 是指目标平台，还有如 hi3093 等）进行构建，所以构建系统的分析可从 build.py 入手进行。

答：

【系统架构 】UniProton 系统由 Mem、Arch、Kernel、IPC、OM 五大子系统构成，Mem、Arch 是整个系统的基石。

各子系统的职责如下：

Mem：实现内存分区的创建，内存块的申请和释放等功能。

Arch：实现和芯片强相关的硬件特性功能，如硬中断、异常接管等。

Kernel：实现任务、软中断、TICK 中断、软件定时器等功能。

IPC：实现事件、队列、信号量等功能。

OM：实现 cpup、hook 等调测功能。

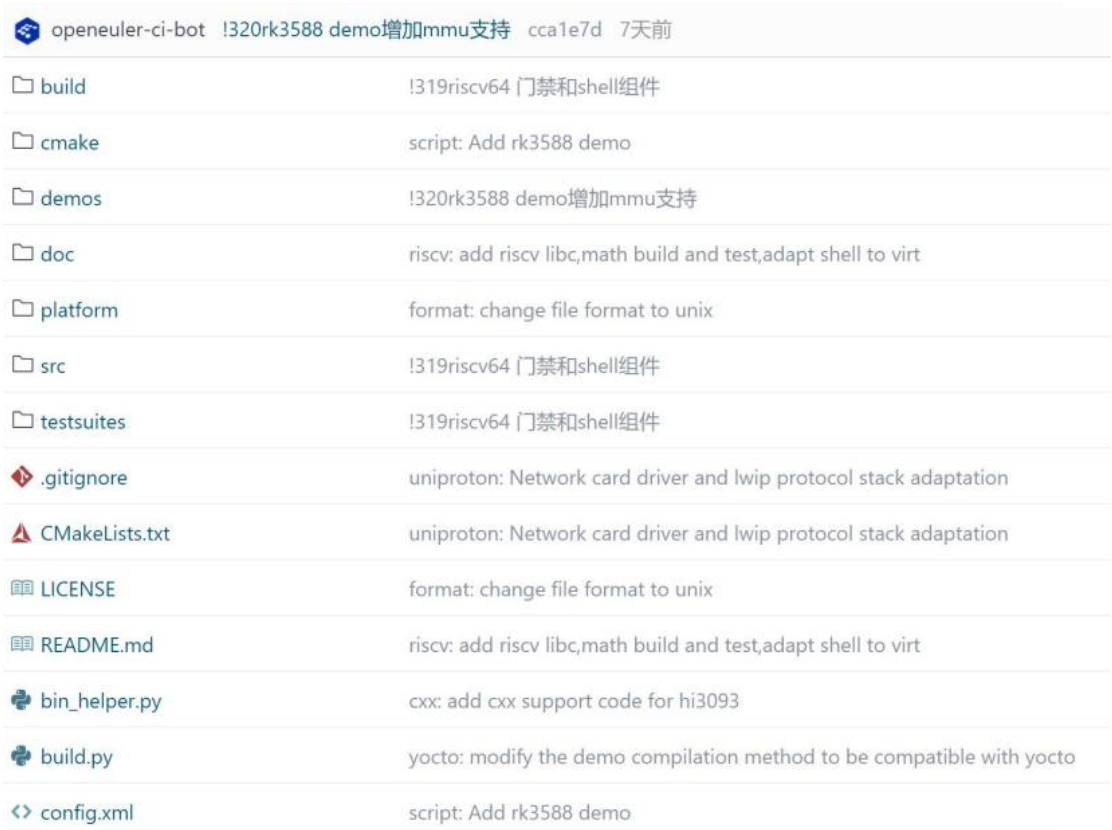
代码目录结构

一级目录	二级目录	三级目录	说明
build	uniproton_ci_lib		编译框架的公共脚本
	uniproton_config	config_m4	cortex_m4芯片的功能宏配置文件
cmake	common	build_auxiliary_script	转换Kconfig文件为buildef.h脚本
	functions		cmake的公共功能函数
	tool_chain		编译器和编译选项配置文件
demos	helloworld	apps	示例程序的main函数文件以及业务代码
		bsp	板级驱动适配代码
		build	demo构建及链接脚本
		config	用户配置文件，功能宏定制头文件
		include	src/include/uapi及posix目录下头文件拷贝目录
		libs	源码编译静态库文件存放目录
	Hi3093		与helloworld demo类似，详见对应目录下的readme文件
	RASPI4		与helloworld demo类似，详见对应目录下的readme文件
doc			项目配置、规范、协议等文档
	design		UniProton系统架构和特性说明
platform			libboundscheck使用说明
	libboundscheck		libboundscheck预留目录，用户将下载的源码放在此目录下
src	arch	cpu	cpu对应架构的功能适配代码
		include	cpu对应架构的头文件
	config		用户main函数入口
		config	用户配置功能宏开关
	core	ipc	事件、队列、信号量等功能
		kernel	任务、中断、异常、软件定时器等功能

	fs	littlefs	littlefs适配层代码，不包含完整littlefs代码
		vfs	文件系统vfs层接口代码
	include	uapi	对外头文件
		posix	posix接口头文件
	mem		内存管理基本功能
		fsc	内存管理FSC算法代码
		include	内存管理头文件
	net	lwip-2.1	lwip适配层代码，不包含完整lwip代码
	om	cpup	cpu占用率统计功能
		err	错误处理功能
		hook	钩子函数功能
		include	系统管理头文件
	kal		posix功能适配中间层
	libc		posix功能实现源码
		litelibc	musl接口适配UniProton源码
		musl	musl 1.2.3源码，不包含litelibc重复源码
	osal	cxx	支持c++接口
		linux	支持linux接口
	drivers		驱动相关实现
	security	rnd	随机化功能
	utility	lib	公共库函数
	testsuites		测试用例
		bsp	板级驱动适配代码(m4)
		build	构建及链接脚本(仅适用于m4)
		config	用户配置文件，功能宏定制头文件
		drivers	驱动框架测试用例
		libc-test	开源musl libc-test适配的测试用例，用于测试部分posix接口
		posixtestsuite	开源posixtestsuite适配的测试用例，用于测试部分posix接口
		rhealstone	m4性能测试适配用例
		shell-test	shell测试用例
		support	测试main入口

`UniProton` 通过在根目录下执行 `python build.py m4` （`m4` 是指目标平台，还有如 `hi3093` 等）进行构建。

根目录文件夹内容如下所示：



下面是 build.py 的源码：

https://gitee.com/openeuler/UniProton/blob/master/build.py

该脚本是用 Python 编写的，旨在自动执行不同配置和平台的构建过程。它包括用于设置构建环境、准备编译环境以及执行构建过程本身的功能。

下面是脚本的关键组件和功能的细分：

① 初始化和配置：脚本首先导入必要的模块并设置环境。它定义了 UniProton 的主目录， 并将辅助脚本和库的路径插入到系统路径中。它还导入一个名为 globle 和日志记录模块的自定义模块。

② 编译类：脚本的主要功能封装在类中 Compile 。此类负责设置生成环境、准备用于编译的环境以及执行生成过程。它包括用于获取配置设置、设置命令环境变量、准备编译环境以及执行生成过程的方法。

③ 环境准备：该 prepare_env 方法至关重要，因为它根据提供的 CPU 类型和选择设置环境。它检索配置设置，设置命令环境变量，并为生成过程准备环境。

④ 构建过程执行：该 CMake 方法用于使用 CMake 生成 Makefile，该 make 方法用于执行构建过程。该脚本还包括用于清理构建环境（ UniProton_clean ）和编译 SDK（ SdkCompaile ）的方法。

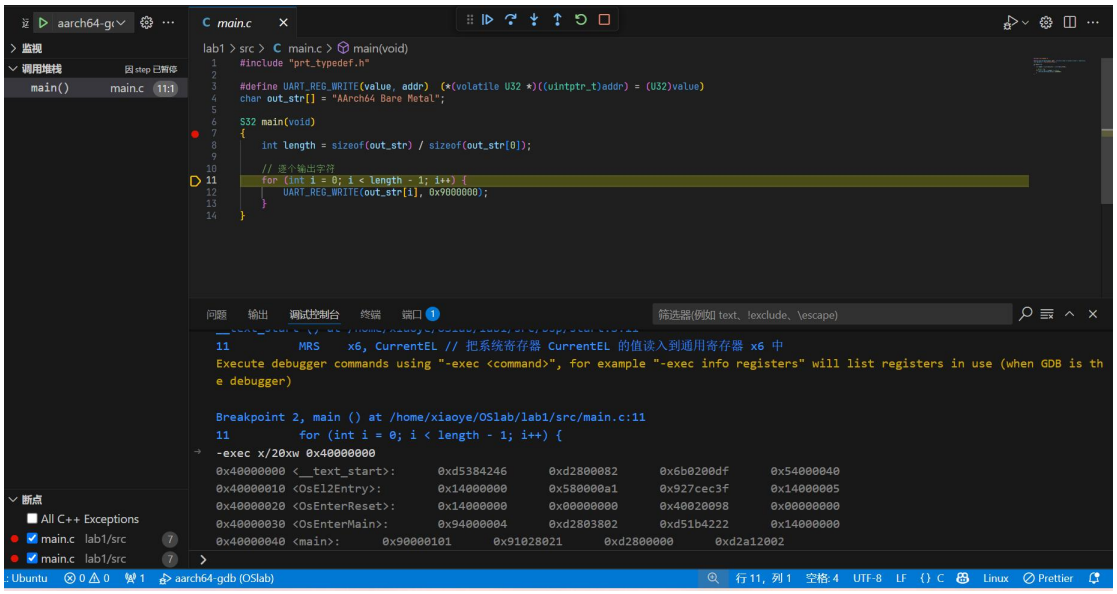
⑤ 主执行：脚本检查命令行参数以确定 CPU 类型、编译选项、库运行类型、make choice、make 阶段和 UniProton 打包路径。然后，它使用这些参数初始化类，Compile 并调用该 UniProtonCompile 方法来启动生成过程。

此脚本旨在灵活地适应不同的生成配置和平台。它演示了一种自动化构建过程的复杂方法，包括处理不同的编译模式、管理环境变量和执行复杂的构建命令。

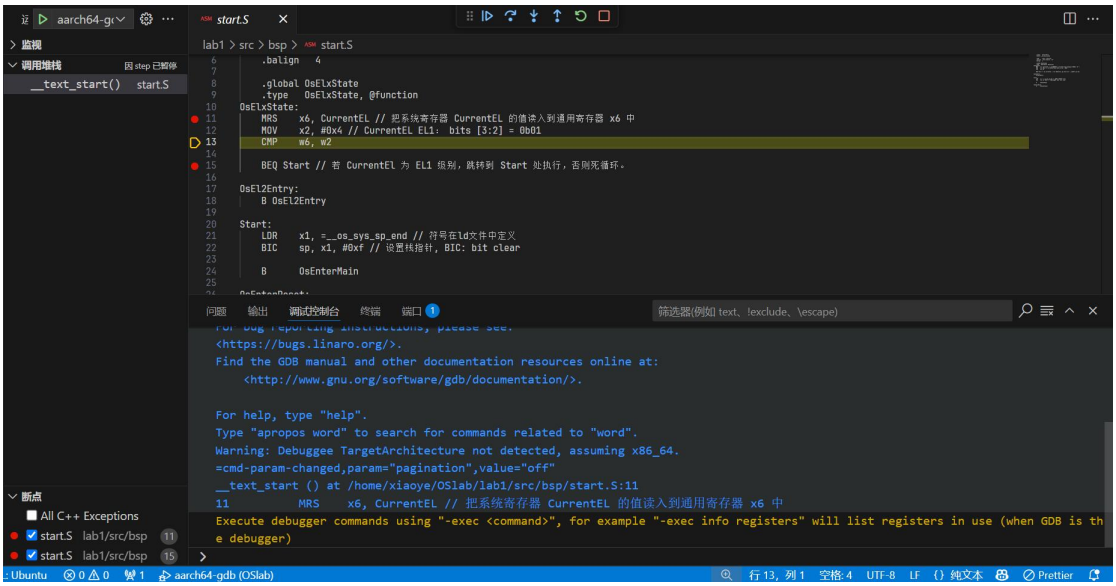
3、 作业 3

学习如何调试项目。

（一）调试.c 文件



（1）查看指定地址的内存内容。在调试控制台执行 -exec x/20xw 0x40000000 即可，其中 x 表示查看命令，20 表示查看数量，x 表示格式，可选格式包括 Format letters are o(octal), x(hex), d(decimal), u(unsigned decimal),t(binary), f(float), a(address), i(instruction), c(char) and s(string).Size letters are b(byte), h(halfword), w(word), g(giant, 8 bytes)，最后的 w 表示字宽，b 表示单字节，h 表示双字节，w 表示四字节，g 表示八字节。还可以是指令：
-exec x/20i 0x40000000; 字符串：-exec x/20s 0x40000000
（2）显示所有寄存器。-exec info all-registers
（3）查看寄存器内容。-exec p/x $pc
（4）修改寄存器内容。-exec set $x24 = 0x5
（5）修改指定内存位置的内容。-exec set {int}0x4000000 = 0x1 或者 -exec set *((int *) 0x4000000) = 0x1
（6）修改指定 MMIO 寄存器的内容。 -exec set *((volatile int *) 0x08010004) = 0x1
（7）退出调试 -exec q
总之，可以通过 -exec 这种方式可以执行所有的 gdb 调试指令。
(二) 调试.s 文件


start.S
lab1 > src > bsp > start.S
.balign 4
.global OsElxState
.type OsElxState, @function
OsElxState:
MRS x6, CurrentEL // 把系统寄存器 CurrentEL 的值读入到通用寄存器 x6 中
MOV x2, #0x4 // CurrentEL EL1: bits [3:2] = 0b01
CMP w6, w2
BEQ Start // 若 CurrentEL 为 EL1 级别，就转到 Start 处执行，否则死循环。
OsEl2Entry:
B OsEl2Entry
Start:
LDR x1, =__os_sys_sp_end // 符号在ld文件中定义
BIC sp, x1, #0xf // 设置栈指针, BIC: bit clear
B OsEnterMain
问题 输出 调试控制台 终端 端口
筛选器(例如 text、!exclude、\escape)
<https://bugs.linaro.org/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
Warning: Debuggee TargetArchitecture not detected, assuming x86_64.
=cmd-param-changed,param="pagination",value="off"
__text_start () at /home/xiaoye/OSlab/lab1/src/bsp/start.S:11
11 MRS x6, CurrentEL // 把系统寄存器 CurrentEL 的值读入到通用寄存器 x6 中
Execute debugger commands using "-exec <command>", for example "-exec info registers" will list registers in use (when GDB is the debugger)
aarch64-gdb (OSlab)
行 13，列 1 空格: 4 UTF-8 LF 纯文本 Prettier
监视
调用堆栈
因 step 已暂停
__text_start() start.S
断点
All C++ Exceptions
start.S lab1/src/bsp 11
start.S lab1/src/bsp 15

3 心得体会

3.1 心得体会

（1）配置环境的过程中耐心，可能会出现很多莫名奇妙的报错，上网检索、前人经验都值得参考学习，要善于利用资源、学会解决问题；

（2）初步了解了裸机程序和工程构建，和操作系统的架构，建立起了一个较为具体和直观的概念；

（3）进一步加深了对 gdb 断点调试的理解；