《操作系统》

实验一报告

目录

1 遇到的问题及解决方案

1.1 虚拟机安装及版本问题

光盘映射文件从 ubuntu 官网下载，下载使用过 18 版本和 22 版本，由于老师的实验步骤提到过使用 18 版本测试过，本来意图使用 18 版本，但该实验写到一半的时候，大概做了一周左右，我的 ubuntu18 出现一个问题，上网搜索调试了两天也没有解决。向同学求助后对方说 18 版本有一些问题，后续进行另外的实验步骤时可能会下载非常慢，且 22 版本比较新，一些配置做的也更好，用起来更顺手，故转而使用 22 版本重新开始，很好用。

开始时想要使用 VirtualBox，安装并配置后一直卡在启动进入的界面，报的错我上网搜索了很久尝试重新分配也没能解决，上课的时候请教老师和同学也没有得到解决方案，同学推荐我用 wmware，下载使用后没有遇到问题，所以后续的实验我都使用 wmware 的 ubuntu22 版本进行。（这两个问题就解决了好几天……）

1.2 一些自定义配置

做了一些比较好用的自定义配置：

1. 更改快捷键，在终端的 preferences-shortcuts 中更改一些配置，比如复制粘贴。
2. 配置了基于输入的上下键自动补全功能。
3. 使用 vim，这部分看了 MIT 的网课 The Missing Semester of Your CS Education，学了一些基本操作，并且使用第三节课程资源的~/.vimrc 作为自己的配置，同时在提供的资源中下载了【ctrlp.vim: 模糊文件查找】（一个通过 github 可以下载的插件），实现在 vim 中使用 ctrl-p 快捷切换文件的功能。
4. 下载了一些可爱的小插件，比如 history、tree 等，tree 可以输出和老师给的文件结构一样的形式，比如：

5. 主机和虚拟机间文件拖拽传送，下载 vmware-tools

6. 还有一些想不起来了的配置，做了太久了。额外安装了一个 g++。

接下来将按照实验步骤逐步回忆。

1.3 安装工具链

交叉编译工具链：

交叉编译工具用于在一个架构上编译另一个架构的代码，例：在 x86 架构上编译 arm 架构的代码。

1. 目录 /path/to/your/aarch64-none-elf/bin，问题是如何找到自己的该目录，一开始指导书没有更新的时候（没有提到要修改为自己的路径），往后做了两步一直报错，那个下午我都在搞这个问题，有怀疑过这个路径，但我用 where 和其他指令搜过，没有搜到，自己尝试改过，没有成功，也没想到点进去到处找找，在网上到处搜也没找到，最后同学提供了一个 csdn 链接，根据它参考配置的。

2. 添加到 path 变量中，根据 csdn 的教程修改了这三个文件，重启使其生效。

```
gedit .bashrc
gedit .profile
gedit /etc/profile
```

QEMU 模拟器：安装顺利。

CMake：

这里看起来直接使用 apt 安装成功了，但是后面会有版本问题，需要卸载重新安装，到后面的部分我再详细说明。

1.4 创建裸机程序

操作上只需要创建各个文件和目录并将内容放进去即可，困难且重要的地方在于阅读源码并理解。辅助：可以扔给 chatgpt 让它一行行解释。

这是老师给的 lab1 的初始目录结构，两个 CMakeList 都是用于编译的配置文件。

aarch64-qemu.ld 是创建的自定义链接器脚本。链接器脚本通常包含对内存布局的描述、符号解析的规则、代码和数据段的布局以及如何将这些段映射到输出文件中的特定位置。它们对于嵌入式系统、操作系统内核以及任何需要精确控制内存布局和符号解析的项目来说至关重要。

bsp 目录存放与硬件紧密相关的代码，include 目录中存放项目的大部分头文件。

prt_typedef.h 头文件：它是 UniProton 所使用的基本数据类型和结构的定义，如 U8、U16、U32、U64 等。

main.c 是我们的主函数，执行的一部分。

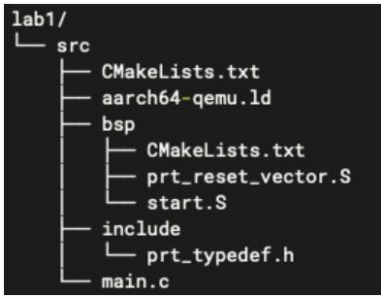
执行时从 start.S 跳转到 prt_reset_vector.S，再跳转到 main.c，执行完后返回 prt_reset_vector.S，陷入无限循环。OxElxState 为系统入口。

```
10  OsElxState:
11      MRS     x6, CurrentEL // 把系统寄存器 CurrentEL 的值读入到通用寄存器 x
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
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
23
24      B       OsEnterMain
25
26  OsEnterReset:
27      B       OsEnterReset
```

```
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
```

start.S 中，OsElxState 主要用于检查当前处理器的异常级别是否为 EL1，并据此决定是

否跳转到 Start 执行后续的代码。Start 设置栈指针、清零并跳转到 prt_reset_vector.S 中的 OsEnterMain。它跳转到 main.c 中的 main 函数，执行后返回，禁用中断并陷入无限循环。



1.5 工程构建

1. CMakeLists.txt

这是用于编译链接等操作的配置文件。记得修改交叉工具链实际所在目录，我记得这里我也出了问题，最后参考之前的解决的。

2. 编译运行

在这里遇到了 cmake 的版本问题，默认下载的 cmake 版本太老了，上网搜索了许久如何卸载和重新下载更新的版本，下载安装包并解压，我记得还在设置里更改了 cmake 的默认版本，设置为最新版本，并且设置了 PATH，这样使用 cmake --version 显示的版本就被更新了。但是使用 cmake ../src 仍然出现了问题，尝试在讨论区中提问，后来重新查找了工具链地址并更新在上一个文件中，还修改了一个文件的命名错误。

配置成功后成功运行。

查看 history 的时候发现我在这里还花了很久下载 libpython3.6，但是真的想不起来是哪里需要这个配置了（可能是下一步启动客户调试端的时候需要用？）

1.6 调试支持

GDB 简单调试方法

这里有一个坑！执行

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

这一条命令的时候，命令中有-S 参数，当前对话框会静止，无法操作。我在执行的时候查过这个参数的含义是：使 QEMU（Quick EMUlator）在启动后暂停（等待）执行。但是

并不知道需要新开一个终端进行下一步操作，在这里又卡了一个晚上。

将调试集成到 vscode：

也需要先执行 QEMU 指令再使用 vscode 才能连接上端口。

1.7 自动化脚本调试

```
rm -rf build/*
mkdir build
cd build
cmake ../src
cmake --build . $1
```

```
virt,gic-version=2 -m 1024M -cpu cortex-a53 -nographic -kernel build/miniEuler  -s $1

gic-version=2 -m 1024M -cpu cortex-a53 -nographic -kernel build/miniEuler  -s $1
```

这里的$1 代表的是外界调用时输入的第一个参数，不可省略，验收时有一个同学脚本中的$1 都没有写，执行直接出结果，不会等待，我帮她检查的时候发现了这个问题。

1.8 执行结果

gdb 简单调试:

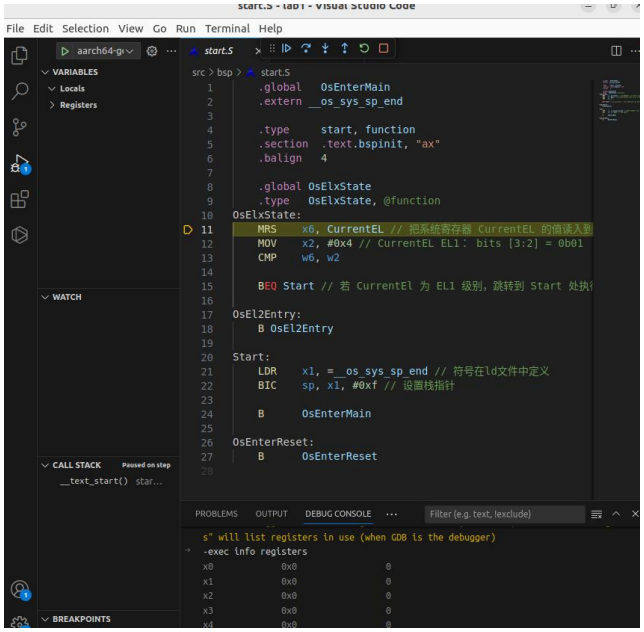
```
guoruiling@guoruiling-virtual-machine:~$ cd lab1
guoruiling@guoruiling-virtual-machine:~/lab1$ ls
build  makeMiniEuler.sh  runMiniEuler.sh  src
guoruiling@guoruiling-virtual-machine:~/lab1$ sh makeMiniEuler.sh
mkdir: cannot create directory ‘build’: File exists
-- The C compiler identification is GNU 11.2.1
-- The ASM compiler identification is GNU
-- Found assembler: /home/guoruiling/aarch64-none-elf/gcc-arm-11.2-2022.02-x86_64-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/guoruiling/aarch64-none-elf/gcc-arm-11.2-2022.02-x86_64-
aarch64-none-elf/bin/aarch64-none-elf-gcc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/guoruiling/lab1/build
Scanning dependencies of target bsp
[ 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
guoruiling@guoruiling-virtual-machine:~/lab1$ sh runMiniEuler.sh -S
qemu-system-aarch64 -machine virt,gic-version=2 -m 1024M -cpu cortex-a53 -nographic -kernel bui
ld/miniEuler -s -S
AArch64 Bare Metal
```

```
guoruiling@guoruiling-virtual-machine:~/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.
For bug reporting instructions, please see:
<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"...
Reading symbols from build/miniEuler...
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
__text_start () at /home/guoruiling/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
(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
12          MOV     x2, #0x4 // CurrentEL EL1： bits [3:2] = 0b01
(gdb) n
13          CMP     w6, w2
(gdb) n
15          BEQ Start // 若 CurrentEl 为 EL1 级别，跳转到 Start 处执行，否则死循环。
(gdb) n
21          LDR     x1, =__os_sys_sp_end // 符号在ld文件中定义
(gdb) n
22          BIC     sp, x1, #0xf // 设置栈指针
(gdb) n
^Z
[1]+  Stopped                 aarch64-none-elf-gdb build/miniEuler
guoruiling@guoruiling-virtual-machine:~/lab1$
```

vscode:

```
guoruiling@guoruiling-virtual-machine:~/lab1$ sh runMiniEuler.sh -S
qemu-system-aarch64 -machine virt,gic-version=2 -m 1024M -cpu cortex-a53 -nographic -kernel bui
ld/miniEuler -s -S
AArch64 Bare MetalQEMU: Terminated
```



2 实验任务

2.1 作业 1

题目：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。

提示：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。

在网络上寻找资源下载 UniProton，解压的文件夹里有 build.py 的代码，我将简要地分块分析。

```
#!/usr/bin/env python3
# coding=utf-8
# The build entrance of UniProton.
# Copyright (c) 2009-2023 Huawei Technologies Co., Ltd. All rights reserved.

import os
import sys
import time
import shutil
import subprocess
import platform
from sys import argv
UniProton_home = os.path.dirname(os.path.abspath(__file__))
sys.path.insert(0, "%s/cmake/common/build_auxiliary_script"%UniProton_home)
from make_buildef import make_buildef
sys.path.insert(0, "%s/build/uniproton_ci_lib"%UniProton_home)
import globle
from logs import BuilderNolog, log_msg
from get_config_info import *
```

导入模块:

1.导入了 os, sys, time, shutil, subprocess, platform 等 Python 标准库模块。

2.导入了自定义的模块，如 make_buildef, globle, logs, get_config_info 等，这些模块包含 UniProton 项目特定的功能。

设置 UniProton 项目的路径:

1.UniProton_home 变量被设置为当前脚本文件的绝对路径。

2.修改了 sys.path 以包含额外的模块路径，使得脚本可以导入这些路径下的模块。

```
class Compile:

    # 根据makechoice获取config的配置的环境，compile_mode，  lib_type，
    def get_config(self, cpu_type, cpu_plat):
        self.compile_mode = get_compile_mode()
        self.lib_type, self.plam_type, self.hcc_path, self.kconf_dir, self.system, self.core =
get_cpu_info(cpu_type, cpu_plat, self.build_machine_platform)
        if not self.compile_mode and self.lib_type and self.plam_type and self.hcc_path and
self.kconf_dir:
            log_msg('error', 'load config.xml env error')
            sys.exit(0)
        self.config_file_path = '%s/build/uniproton_config/config_%s'%(self.home_path,
self.kconf_dir)

        self.objcopy_path = self.hcc_path

    def setCmdEnv(self):
        self.build_time_tag = time.strftime('%Y-%m-%d_%H:%M:00')
        self.log_dir = '%s/logs/%s' % (self.build_dir, self.cpu_type)
        self.log_file = '%s.log' % self.kconf_dir

    def SetCMakeEnviron(self):
        os.environ["CPU_TYPE"] = self.cpu_type
        os.environ["PLAM_TYPE"] = self.plam_type
        os.environ["LIB_TYPE"] = self.lib_type
        os.environ["COMPILE_OPTION"] = self.compile_option
        os.environ["HCC_PATH"] = self.hcc_path
        os.environ["UNIPROTON_PACKING_PATH"] = self.UniProton_packing_path
        os.environ["CONFIG_FILE_PATH"] = self.config_file_path
        os.environ["LIB_RUN_TYPE"] = self.lib_run_type
        os.environ["HOME_PATH"] = self.home_path
        os.environ["COMPILE_MODE"] = self.compile_mode
        os.environ["BUILD_MACHINE_PLATFORM"] = self.build_machine_platform
        os.environ["SYSTEM"] = self.system
        os.environ["CORE"] = self.core
        os.environ["OBJCOPY_PATH"] = self.objcopy_path
        os.environ['PATH'] = '%s:%s' % (self.hcc_path, os.getenv('PATH'))
```

定义 Compile 类，它包含用于编译 UniProton 项目的方法。

get_config 方法:

这个方法用于获取编译配置。它调用 get_compile_mode 和 get_cpu_info 函数来获取编译模式、库类型、平台类型、编译器路径、内核配置目录等信息。如果获取到的配置信息不完整或存在错误，它将记录一条错误日志并退出程序。另外，它还设置了配置文件的路径（黄色）。

setCmdEnv 方法:

这个方法用于设置命令环境。它获取当前的构建时间，并格式化为 YYYY-MM-DD_HH:MM:00 的形式。设置了日志目录和日志文件的路径，这些路径基于构建目录和 CPU 类型。

SetCMakeEnviron 方法：

这个方法用于设置 CMake 所需的环境变量。这些变量通常用于 CMake 配置和构建过程中，包含了项目编译所需的配置信息，如 CPU 类型、库类型、编译器路径等。PATH 环境变量被更新，以确保 CMake 和其他工具可以找到必要的路径，这里特别将 hcc_path（编译器路径）添加到 PATH 环境变量的开头。

```
    # 环境准备，准备执行cmake，make，makebuildfile，CmakeList需要的环境
    # 每次compile之前请调用该函数
    def prepare_env(self, cpu_type, choice):
        # makebuildfile需要的环境kconf_dir
        # cmake需要的环境cmake_env_path，home_path（cmakelist所在的路径），home_path，
        # make cmd拼接需要的环境：home_path,UniProton_make_jx,log_dir,log_file，build_time_tag，
UniProton_make_jx

        #根据cpu_type，choice从config文件中获取并初始化初始化hcc_path， plam_type，kconf_dir
        #根据输入分支获取
        #从编译镜像环境获取
        self.get_config(cpu_type, choice)
        self.setCmdEnv()
        self.SetCMakeEnviron()
```

prepare_env 方法：

这个方法用于准备编译环境，确保在调用 CMake、Make 或其他构建工具之前，所有必要的环境和配置都已正确设置。这个方法在每次编译之前调用，调用了前面三个方法进行总体构建。

```
    #获取编译环境是arm64还是x86，用户不感知，并将其写入环境变量。
    def getOsPlatform(self):
        self.cmake_env_path = get_tool_info('cmake', 'tool_path')

        if platform.uname()[-1] == 'aarch64':
            self.build_machine_platform = 'arm64'
        else:
            self.build_machine_platform = 'x86'

```

getOsPlatform 方法：

这个方法用于获取编译环境。

```
    # 获取当前编译的路径信息，配置文件信息，编译选项信息
    def __init__(self, cpu_type: str, make_option="normal", lib_run_type="FPGA", choice="ALL",
make_phase="ALL",
                 UniProton_packing_path=""):
        # 当前路径信息
        self.system = ""
        self.objcopy_path = ""
        self.builder = None
        self.compile_mode = ""
        self.core = ""
        self.plam_type = ""
        self.kconf_dir = ""
        self.build_tmp_dir = ""
        self.log_dir = ""
        self.lib_type = ""
        self.hcc_path = ""
        self.log_file = ""
        self.config_file_path = ""
        self.build_time_tag = ""
        self.build_dir = globle.build_dir
        self.home_path = globle.home_path
        self.kbuild_path = globle.kbuild_path
        # 当前选项信息
        self.cpu_type = cpu_type
        self.compile_option = make_option
        self.lib_run_type = lib_run_type
        self.make_choice = choice.lower()
        self.make_phase = make_phase
        self.UniProton_packing_path = UniProton_packing_path if make_phase == "CREATE_CMAKE_FIL
else '%s/output'%self.home_path
        self.UniProton_binary_dir = os.getenv('RPROTON_BINARY_DIR')
        self.UniProton_install_file_option = os.getenv('RPROTON_INSTALL_FILE_OPTION')
        self.UniProton_make_jx = 'VERBOSE=1' if self.UniProton_install_file_option ==
'SUPER_BUILD' else 'VERBOSE=1 -j' + str(os.cpu_count())
        # 当前编译平台信息
        self.getOsPlatform()
```

__init__方法：

这个方法用于初始化。为 Compile 类提供了初始化逻辑，在创建 Compile 对象时，能够

基于传入的参数和全局配置来设置对象的初始状态。

```
    #调用cmake生成Makefile文件，需要
    def CMake(self):
        if self.UniProton_binary_dir:
            self.build_tmp_dir = '%s/output/tmp/%s' % (self.UniProton_binary_dir, self.kconf_dir)
        else:
            self.build_tmp_dir = '%s/output/tmp/%s' % (self.build_dir, self.kconf_dir)
        os.environ['BUILD_TMP_DIR'] = self.build_tmp_dir

        if not os.path.exists(self.build_tmp_dir):
            os.makedirs(self.build_tmp_dir)
        if not os.path.exists(self.log_dir):
            os.makedirs(self.log_dir)

        log_msg('info', 'BUILD_TIME_TAG %s' % self.build_time_tag)
        self.builder = BuilderNolog(os.path.join(self.log_dir, self.log_file))
        if self.make_phase in ['CREATE_CMAKE_FILE', 'ALL']:
            real_path = os.path.realpath(self.build_tmp_dir)
            if os.path.exists(real_path):
                shutil.rmtree(real_path)
            os.makedirs(self.build_tmp_dir)
            #拼接cmake命令
            if self.compile_option == 'fortify':
                cmd = '%s/cmake %s -DCMAKE_TOOLCHAIN_FILE=%s/cmake/tool_chain/
uniproton_tool_chain.cmake ' \
                      '-DCMAKE_C_COMPILER_LAUNCHER="sourceanalyzer;-b;%sproject" ' \
                      '-DCMAKE_INSTALL_PREFIX=%s &> %s/%s' % (
                self.cmake_env_path, self.home_path, self.home_path, self.cpu_type,
                self.UniProton_packing_path, self.log_dir, self.log_file)
            elif self.compile_option == 'hllt':
                cmd = '%s/cmake %s -DCMAKE_TOOLCHAIN_FILE=%s/cmake/tool_chain/
uniproton_tool_chain.cmake ' \
                      '-DCMAKE_C_COMPILER_LAUNCHER="lltwrapper" -DCMAKE_INSTALL_PREFIX=%s &> %s/
%s' % (
                self.cmake_env_path, self.home_path, self.home_path, self.UniProton_packing_path,
self.log_dir, self.log_file)
            else:
                cmd = '%s/cmake %s -DCMAKE_TOOLCHAIN_FILE=%s/cmake/tool_chain/
uniproton_tool_chain.cmake ' \
                      '-DCMAKE_INSTALL_PREFIX=%s &> %s/%s' % (
                self.cmake_env_path, self.home_path, self.home_path, self.UniProton_packing_path,
self.log_dir, self.log_file)

            #执行cmake命令
            if self.builder.run(cmd, cwd=self.build_tmp_dir, env=None):
                log_msg('error', 'generate makefile failed!')
                return False

        log_msg('info', 'generate makefile succeed.')
        return True
```

CMake 方法：

这个方法的主要目的是使用 cmake 来生成 Makefile 文件，用于后续的编译过程。

确定构建临时目录：

如果 self.UniProton_binary_dir 存在，则使用它作为构建临时目录的一部分；否则，使用默认的 self.build_dir。设置环境变量 BUILD_TMP_DIR 为构建临时目录。

创建目录：

如果构建临时目录或日志目录不存在，则创建它们。

日志记录：

使用 log_msg 函数记录构建时间标签 self.build_time_tag。

设置构建器：

创建一个 BuilderNolog 对象（这个类也未在这段代码中定义），用于后续的构建过程，并将日志记录到指定的日志文件中。

处理特定的编译阶段：

如果 self.make_phase 是 CREATE_CMAKE_FILE 或 ALL，则执行以下操作：删除并重新创建构建临时目录，以确保目录是干净的。

根据 self.compile_option 的值构建 cmake 命令;如果 self.compile_option 是 fortify，则使用 sourceanalyzer 作为 C 编译器的启动器，用于静态代码分析；如果 self.compile_option 是 hllt，则使用 lltwrapper 作为 C 编译器的启动器。

对于其他选项，不设置特定的编译器启动器。

cmake 命令还指定了工具链文件的位置、安装前缀以及日志文件的输出位置。

拼接 cmake 命令：

根据 self.compile_option 的值，拼接不同的 cmake 命令。如果 self.compile_option 为 fortify，命令中包含-DCMAKE_C_COMPILER_LAUNCHER="sourceanalyzer;-b;%sproject"，用于静态代码分析;如果 self.compile_option 为 hllt，命令中包含 -DCMAKE_C_COMPILER_LAUNCHER="lltwrapper"，用于启动特定的编译器包装器。

对于其他选项，不设置特定的编译器启动器。

命令中还包括 cmake 的路径、源代码目录、工具链文件路径、安装前缀以及日志文件的输出位置。

执行 cmake 命令：

使用 self.builder.run()方法执行拼接好的 cmake 命令。执行目录设置为 self.build_tmp_dir。如果命令执行失败（返回值为 False），则记录错误日志并返回 False；如果命令执行成功，则记录信息日志并返回 True。

```
157     def UniProton_clean(self):
158         for foldername,subfoldernames,filenames in os.walk(self.build_dir):
159             for subfoldername in subfoldernames:
160                 if subfoldername in ['logs','output','__pycache__']:
161                     folder_path = os.path.join(foldername,subfoldername)
162                     shutil.rmtree(os.path.relpath(folder_path))
163             for filename in filenames:
164                 if filename == 'prt_buildef.h':
165                     file_dir = os.path.join(foldername,filename)
166                     os.remove(os.path.relpath(file_dir))
167         if os.path.exists('%s/cmake/common/build_auxiliary_script/__pycache__'%self.home_path):
168             shutil.rmtree('%s/cmake/common/build_auxiliary_script/__pycache__'%self.home_path)
169         if os.path.exists('%s/output'%self.home_path):
170             shutil.rmtree('%s/output'%self.home_path)
171         if os.path.exists('%s/tools/SRE/x86-win32/sp_makepatch/makepatch'%self.home_path):
172             os.remove('%s/tools/SRE/x86-win32/sp_makepatch/makepatch'%self.home_path)
173         if os.path.exists('%s/build/prepare/__pycache__'%self.home_path):
174             shutil.rmtree('%s/build/prepare/__pycache__'%self.home_path)
175         return True
```

UniProton_clean 方法：

用于清理构建过程中产生的某些文件和目录。

遍历构建目录：

使用 os.walk(self.build_dir)遍历 self.build_dir 目录及其所有子目录。os.walk 会生成一个

三元组，包括当前目录的路径、当前目录下的子目录名列表和当前目录下的文件名列表。

清理特定子目录：

对于遍历到的每个子目录，检查其名称是否在['logs', 'output', '__pycache__']列表中。如果是，则构造该子目录的绝对路径，并使用 shutil.rmtree 方法删除该目录及其所有内容。这里使用 os.path.join 拼接路径，使用 os.path.relpath 将绝对路径转换为相对路径。

清理特定文件：

对于遍历到的每个文件，检查其名称是否为'prt_buildef.h'。如果是，则构造该文件的绝对路径，并使用 os.remove 方法删除该文件。

清理特定固定路径的目录和文件：

代码检查了几个固定路径的目录和文件是否存在，并使用 shutil.rmtree 或 os.remove 删除它们。

返回清理成功状态：

返回 True，表示清理操作已完成。

```
178     def make(self):
179         if self.make_phase in ['EXECUTING_MAKE', 'ALL']:
180             self.builder.run('make clean', cwd=self.build_tmp_dir, env=None)
181             tmp = sys.argv
182             if self.builder.run(
183                     'make all %s &>> %s/%s' % (
184                     self.UniProton_make_jx, self.log_dir, self.log_file), cwd=self.build_tmp_dir,
    env=None):
185                 log_msg('error', 'make %s %s  failed!' % (self.cpu_type, self.plam_type))
186                 return False
187             sys.argv = tmp
188             if self.compile_option in ['normal', 'coverity', 'single']:
189                 if self.builder.run('make install %s &>> %s/%s' % (self.UniProton_make_jx,
    self.log_dir, self.log_file), cwd=self.build_tmp_dir, env=None):
190                     log_msg('error', 'make install failed!')
191                     return False
192             if os.path.exists('%s/%s' % (self.log_dir, self.log_file)):
193                 self.builder.log_format()
194 
195         log_msg('info', 'make %s %s succeed.' % (self.cpu_type, self.plam_type))
196         return True
```

make 方法：

该方法主要用于执行 make 命令来构建项目，并根据不同的编译选项执行额外的步骤。

检查当前构建阶段：

如果 self.make_phase 是['EXECUTING_MAKE', 'ALL']中的任何一个，那么执行构建过程。

清理构建目录：

使用 self.builder.run 方法执行 make clean 命令，清理之前的构建结果。执行目录是 self.build_tmp_dir。

保存命令行参数：

将当前的命令行参数（sys.argv）保存在 tmp 变量中，以便在之后恢复。

执行 make all 命令：

使用 self.builder.run 方法执行 make all 命令，并附加了 self.UniProton_make_jx（可能是某些额外的构建参数或目标）。命令的输出被重定向到日志文件中。如果命令执行失败，记录错误日志并返回 False。

恢复命令行参数：

将命令行参数恢复为之前保存的值。

根据编译选项执行 make install：

如果 self.compile_option 是['normal', 'coverity', 'single']中的任何一个，执行 make install 命令来安装构建好的项目。如果安装失败，记录错误日志并返回 False。

处理日志文件：

如果日志文件存在（%s/%s % (self.log_dir, self.log_file)），则调用 self.builder.log_format() 方法处理日志文件。这可能涉及到日志的格式化、解析或其他后处理操作。

记录成功日志：

如果构建成功完成，记录成功日志，并返回 True。

这个方法的主要目的是自动化构建过程，包括清理、编译和安装步骤，并根据不同的编译选项和构建阶段执行相应的操作。同时，它还负责处理构建过程中的日志记录。如果构建过程中的任何步骤失败，该方法将记录错误并返回 False。

```python
    def SdkCompaile(self)->bool:
        # 判断该环境中是否需要编译
        if self.hcc_path == 'None':
            return True

        self.MakeBuildef()
        if self.CMake() and self.make():
            log_msg('info', 'make %s %s lib succeed!' % (self.cpu_type, self.make_choice))
            return True
        else:
            log_msg('info', 'make %s %s lib failed!' % (self.cpu_type, self.make_choice))
            return False

    # 对外函数，调用后根据类初始化时的值进行编译
    def UniProtonCompile(self):
        #清除UniProton缓存
        if self.cpu_type == 'clean':
            log_msg('info', 'UniProton clean')
            return self.UniProton_clean()
        # 根据cpu的编译平台配置相应的编译环境。
        if self.make_choice == "all":
            for make_choice in globle.cpu_plat[self.cpu_type]:
                self.prepare_env(self.cpu_type, make_choice)
                if not self.SdkCompaile():
                    return False
        else:
            self.prepare_env(self.cpu_type, self.make_choice)
            if not self.SdkCompaile():
                return False
        return True

    def MakeBuildef(self):

        if not make_buildef(globle.home_path,self.kconf_dir,"CREATE"):
            sys.exit(1)
        log_msg('info', 'make_buildef_file.sh %s successfully.' % self.kconf_dir)
```

SdkCompaile 方法：

负责执行编译过程。它首先判断是否需要编译（基于 self.hcc_path 的值），然后调用 self.MakeBuildef()生成构建文件，最后使用 CMake 和 make 工具来编译项目。

UniProtonCompile 方法：

是编译的入口点。它根据 CPU 类型和编译选项配置编译环境，并调用 SdkCompaile 进行编译。它支持“clean”操作来清除缓存，并允许为所有支持的 CPU 平台或特定平台编译。

MakeBuildef 方法：

生成构建定义文件。它调用一个外部函数 make_buildef 来完成这个任务。

这些方法使得可以根据不同的 CPU 类型和编译选项自动化地配置编译环境并执行编译过程。通过封装这些逻辑到类中，代码更加模块化和可维护。

```python
# argv[1]: cpu_plat 表示要编译的平台；
# argv[2]: compile_option 控制编译选项，调用不同的cmake参数，目前只有normal coverity hllt fortify single(是否编译安全c，组件化独立构建需求)
# argv[3]: lib_run_type lib库要跑的平台 faga sim等
# argv[4]: make_choice
# argv[5]: make_phase 全量构建选项
# argv[6]: UniProton_packing_path lib库的安装位置
if __name__ == "__main__":
    default_para = ("all", "normal", "FPGA", "ALL", "ALL", "")
    if len(argv) == 1:
        para = [default_para[i] for i in range(0, len(default_para))]
    else:
        para = [argv[i+1] if i < len(argv) - 1 else default_para[i] for i in range(0,len(default_para))]

    cur_cpu_type = para[0].lower()
    cur_compile_option = para[1].lower()
    cur_lib_run_type = para[2]
    cur_make_choice = para[3]
    cur_make_phase = para[4]
    cur_UniProton_packing_path = para[5]
    for plat in globle.cpus_[cur_cpu_type]:
        UniProton_build = Compile(plat, cur_compile_option, cur_lib_run_type, cur_make_choice, cur_make_phase, cur_UniProton_packing_path)
        if not UniProton_build.UniProtonCompile():
            sys.exit(1)
    sys.exit(0)
```

这段代码是该脚本的主要执行部分，包括编译平台、编译选项、库运行平台、构建选项、构建阶段以及库的安装位置。以实现自动化编译过程，允许用户通过命令行参数来定制编译设置。它创建 Compile 类的实例，并调用其方法来执行实际的编译工作。如果在任何平台上编译失败，脚本将立即停止并退出。

UniProton 的构建系统是可以实现自动化编译和构建的工具，它用于编译和构建针对不同平台和选项的库或应用程序。它支持跨平台编译，并提供了丰富的编译选项和构建设置。

2.2 作业 2

学习如何调试项目。

GDB 调试。

```
guoruiling@guoruiling-virtual-machine:~/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.
For bug reporting instructions, please see:
<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"...
Reading symbols from build/miniEuler...
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
EXITLOOP () at /home/guoruiling/lab1/src/bsp/prt_reset_vector.S:14
14          B EXITLOOP
(gdb) disassemble
Dump of assembler code for function EXITLOOP:
=> 0x000000004000003c <+0>:     b       0x4000003c <EXITLOOP>
End of assembler dump.
(gdb) n
n
```

VScode：

- 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
- 显示所有寄存器。-exec info all-registers
- 查看寄存器内容。-exec p/x $pc
- 修改寄存器内容。-exec set $x24 = 0x5
- 修改指定内存位置的内容。-exec set {int}0x4000000 = 0x1 或者 -exec set *((int *) 0x4000000) = 0x1
- 修改指定MMIO 寄存器的内容。 -exec set *((volatile int *) 0x08010004) = 0x1
- 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。