实验 2 MIPS 汇编语言程序设计实验

环境准备

注：请学生跳过本节，因为发放的虚拟机中已经搭建好了相关环境，无需进行此部分操作！

交叉编译环境

交叉编译为在一种指令集机器上编译出另一个目标指令集机器上运行的程序提供了可能，为了在没有实体 MIPS 指令集的机器上编写 MIPS 指令集的程序，我们就需要用到交叉编译工具链来做到程序的交叉编译。本实验中将用到的交叉编译工具包括：

- mips-linux-gnu-as：MIPS 交叉汇编器
- mips-linux-gnu-ld：MIPS 交叉链接器
- mips-linux-gnu-objdump：MIPS 交叉反汇编器
- mips-linux-gnu-readelf：MIPS ELF 文件查看器

现在国内大多数 APT 镜像站均收录了编译好的 MIPS 交叉编译工具链，直接用 APT 软件包管理器安装即可：

```
$ sudo apt install binutils-mips-linux-gnu gcc-mips-linux-gnu gdb-multiarch
```

安装完成后，可以直接调用上述共军进行面向 MIPS 架构的编译汇编和链接过程。

模拟运行环境

在 X86 机器上，我们只能采用模拟器（如 QEMU，Bochs）运行 MIPS 程序，而不能使用虚拟机。虚拟机和模拟器原理不同，前者通过虚拟化硬件运行程序，只能运行和宿主机同架构系统，后者则采用软件模拟目标体系硬件。

QEMU 是一款开源的、广为使用的跨平台模拟器，可以模拟出 x86，x86_64，MIPS，ARM 等等诸多指令集体系架构。QEMU 有用户模式和系统模式两种运行模式，前者是提供了一个类似于仅有 Linux 内核的运行环境，后者则需要自行指定启动系统所需的内核文件和 init 文件等，本实验中采取用户模式即可。

现在国内大多数 APT 镜像站均收录了 QEMU 的各项组件，直接用 APT 软件包管理器安装 QEMU 的用户模式组件即可：

```
$ sudo apt install qemu-user
```

另外，还需要安装跨体系调试工具 `gdb-multiarch` 用于程序调试：

```
$ sudo apt install gdb-multiarch
```

模拟运行和调试

输入输出

从前面的描述可以看出，基于 QEMU 模拟环境运行的 MIPS 程序，是拥有一定的运行时环境的，即可以使用 Linux 内核所提供的系统调用，根据 MIPS 汇编语言知识可知，当 MIPS 程序直接运行于硬件上（裸机运行）时，指令 `syscall` 其实是使用微处理器提供的系统调用（后文称“处理器调用”），然而在 QEMU 中由于增加了操作系统内核层面，`syscall` 将不能再直接使用处理器调用，而必须使用 Linux 内核所提供的系统调用（后文称“内核调用”），由于 MIPS 的商业化程度不高，其文档资源和社区生态都不尽人意，于是只有从内核代码入手寻求内核调用的更多细节，经阅读内核代码，根据 MIPS ABI 规定，摘取本实验中可能用到的部分内核调用如下：

```
//Defined in asm/unistd.h
#define __NR_Linux      4000
#define __NR_syscall      (__NR_Linux + 0)
#define __NR_exit     (__NR_Linux + 1)
#define __NR_fork     (__NR_Linux + 2)
#define __NR_read     (__NR_Linux + 3)
#define __NR_write      (__NR_Linux + 4)
#define __NR_open     (__NR_Linux + 5)
#define __NR_close      (__NR_Linux + 6)
```

MIPS ABI 所规定的 Linux 内核系统调用的规定是将调用号放入 `$v0` 寄存器，然后将系统调用的参数依次放入 `$a0`, `$a1`, `$a2` 和 `$a3` 中，准备好调用号和参数后，通过 `syscall` 指令进入内核态由内核接管具体的调用过程（这部分涉及操作系统课程知识了，本实验中就当做黑盒来使用即可），内核将合法地完成程序请求的调用，并将返回值放入 `$v0` 中。具体地，每个调用号所对应的参数列表也在内核代码中封装：

```
//Defined in unistd.h
/* Terminate program execution with the low-order 8 bits of STATUS.  */
extern void _exit (int __status) __attribute__ ((__noreturn__));
/* Read NBYTES into BUF from FD.  Return the
   number read, -1 for errors or 0 for EOF.
   This function is a cancellation point and therefore not marked with
   __THROW.  */
extern ssize_t read (int __fd, void *__buf, size_t __nbytes) __wur;
/* Write N bytes of BUF to FD.  Return the number written, or -1.
   This function is a cancellation point and therefore not marked with
   __THROW.  */
extern ssize_t write (int __fd, const void *__buf, size_t __n) __wur;
```

在本实验中，我们将用且仅用三个最简单的调用，即：

- `exit`：程序退出调用，用于当程序需要结束运行时结束程序运行；
  - 参数 1：返回值；
- `read`：读调用，用于从给定文件号 `fd` 所指向文件或设备中读取 `nbytes` 个字节到 `buf` 所指向的缓冲区中；
  - 参数 1：文件号，本实验中只需要从 `0` 号文件（标准输入）中读取即可；
  - 参数 2：缓冲区首地址指针，即将读入的字符流所存入的内存地址，通常是一个字符数组；
  - 参数 3：读入的字节数；
  - 返回值：实际读取的字节数，返回 `-1` 则代表出错，返回 `0` 则代表遇到文件尾（EOF）；
- `write`：写调用，用于把 buf 所指向缓冲区开始的 nbytes 个字节输出到给定文件号 fd 所指向文件或设备中；
  - 参数 1：文件号，本实验中只需要往 `1` 号文件（标准输出）中输出即可；
  - 参数 2：缓冲区首地址指针，即将读入的字符流所存入的内存地址，通常是一个字符数组；
  - 参数 3：读入的字节数；

- 返回值：实际写入的字节数，返回 `-1` 则代表出错；

一个简单而完整的 `helloworld` 程序（存为 `example.S`）：

```
  .data                         # 定义数据段，数据将存放于此
stringaa1:                      # 定义变量名，实质是一个地址
  .asciiz "Print this.\n"       # 伪指令.asciiz定义的字符串会自动在其后附上`\0`
  .set noreorder                # 编译用伪指令，不关心
  .set noat                     # 编译用伪指令，不关心
.global main                    # 指出main为全局作用域符号
  .text                         # 代码段，可执行的代码存放于此
main:                           # main过程的入口点
  li $v0, 4004                  # 调用号4004（write调用）放入$v0
  li $a0, 1                     # 目标是1号文件（标准输出）放入$a0
  la $a1, stringaa1             # 待输出的字符串首地址放入$a1，注意此处`la`指令为载入地址指令(Load Address)
                                # 该指令亦可用以下两条指令替代
  #lui $t0, %hi(stringaa1)      # 将字符串地址的高位部分放入$t0中
  #addiu $a1, $t0, %lo(stringaa1) # 将字符串地址的低位部分和$t0相加，放入$a1中
  li $a2, 12                    # 要输出的字节数为12，放入$a2
  syscall                       # 准备就绪，交付内核完成调用
  li $v0, 4001                  # 调用号4001（exit调用）放入$v0
  syscall                       # 准备就绪，交付内核完成调用
```

交叉编译和运行程序

代码写好后，我们将在 x86 环境下编译上述代码并生成 MIPS 可执行文件，随后使用 QEMU 来运行。

由于本实验中代码不是采用 C 语言等高级语言所写，因此不需要预编译和预编译两个步骤，直接从汇编步骤开始：

```
$ mips-linux-gnu-as -g example.S -o ex.o
```

上述命令将 `example.S` 汇编为可重定位目标文件 `ex.o`，并在 `-g` 的指示下加入了符号信息以便于调试。由于我们的实验程序只由这一个模块组成，因此仅链接 `ex.o` 足矣：

```
$ mips-linux-gnu-ld -g ex.o -e main -o ./ex
```

上述命令将 `ex.o` 链接生成可执行文件 `./ex`，并指定了程序入口点为 `maih` 过程。最后使用 QEMU 所提供的 MIPS 用户模式运行交叉编译好的程序 `./ex`：

```
$ qemu-mips ./ex
```

运行结果如下：

```
$ qemu-mips ./ex
Print this.
$ _
```

调试程序

生而为人，没有可以一次跑通的代码，因此就需要使用调试工具 GDB 对所写代码进行调试。首先用 `qemu-mips` 运行程序并打开远程调试端口等待调试：

```
$ qemu-mips -g 1234 ./ex
```

上述命令的 `-g` 表示 `debuG`，其后的数字表示调试服务器端口为 `1234`，该端口将是我们后续步骤中使用调试器连接的端口。如果在指定端口时出现错误，则可能是当前系统中有其他程序占用了该端口，更换任一端口号位于 `1025~65535` 之间的数字作为调试端口均可。

然后运行跨架构调试器 `gdb-multiarch`：

```
$ gdb-multiarch ./ex
```

你将看到：

```
$ gdb-multiarch ./ex
GNU gdb (Ubuntu 7.11-0ubuntu1) 7.11
Copyright (C) 2016 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 "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
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 ./ex...(no debugging symbols found)...done.
(gdb) _
```

当看到 `(gdb)` 提示符时，表明已经成功进入了调试器，首先告诉调试器我们要使用其调试何指令集程序：

```
(gdb) set architecture mips
```

回车后你将看到：

```
(gdb) set architecture mips
The target architecture is assumed to be mips
(gdb) _
```

设定好架构后，连接到调试服务器端口 `1234`，若指定了其他端口，则连接指定的端口：

```
(gdb) target remote localhost:1234
```

若成功连接，程序将进入运行状态，停在程序入口点处：

```
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0x004000f0 in main ()
(gdb) _
```

此后，即可继续使用一般的 GDB 命令对程序进行调试：

```
(gdb) disas
Dump of assembler code for function main:
=> 0x004000f0 <+0>: li  v0,4004
   0x004000f4 <+4>: li  a0,1
   0x004000f8 <+8>: lui a1,0x41
   0x004000fc <+12>:  addiu a1,a1,272
   0x00400100 <+16>:  li  a2,12
   0x00400104 <+20>:  syscall
   0x00400108 <+24>:  li  v0,4001
   0x0040010c <+28>:  syscall
End of assembler dump.
(gdb) info r
          zero       at       v0       v1       a0       a1       a2
a3
 R0   00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
            t0       t1       t2       t3       t4       t5       t6
t7
 R8   00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
            s0       s1       s2       s3       s4       s5       s6
s7
 R16  00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000
            t8       t9       k0       k1       gp       sp       s8
ra
 R24  00000000 00000000 00000000 00000000 00000000 76fff070 00000000
00000000
            sr       lo       hi      bad    cause       pc
      20000010 00000000 00000000 00000000 00000000 004000f0
           fsr      fir
      00000000 00739300
(gdb) _
```

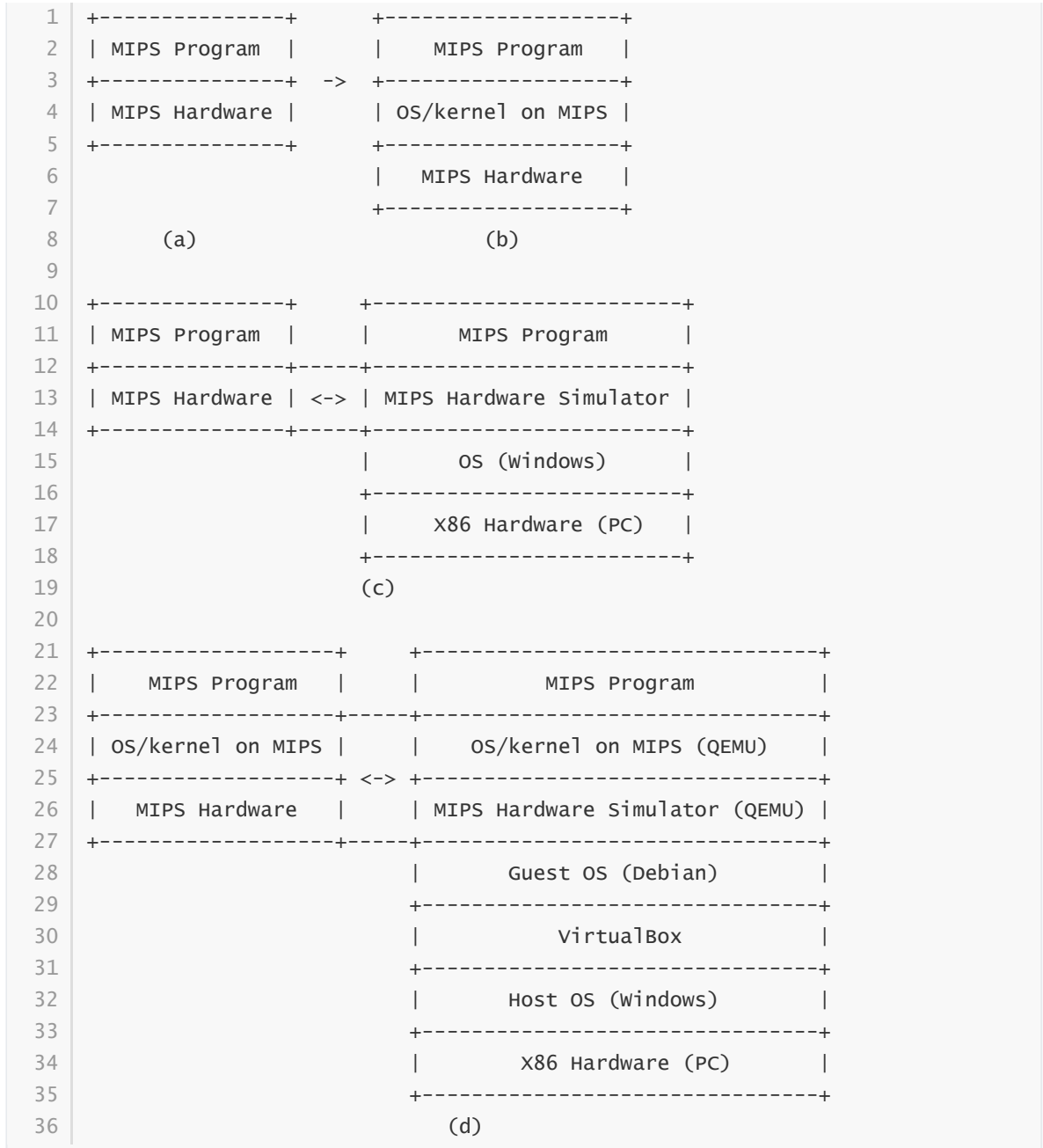
一些疑问

Q. 为什么要在 Linux 环境下使用 QEMU 模拟器运行 MIPS 汇编程序，而不简单采用如 PCSpim 等模拟器对编写的汇编程序进行模拟运行？二者的差别在哪里？为什么要这样做？

A. 要回答这个问题，首先需要了解汇编语言代码所写成程序的两种不同的运行环境。下面我们提供几种程序运行的架构图，注意相邻的两层中，下层向上层提供运行环境，即上层使用下层提供的运行环境。最传统最古老的运行环境即是直接将程序放在硬件上运行，此时的架构如图 (a) 所示，我们称程序直接在硬件上运行，此时程序的运行效率是最高的。由于硬件提供的运行环境有限，我们引入了操作系统层，操作系统层将硬件细节进行封装，对于 Linux 内核而言，则是使用**系统调用**来向用户程序提供便利的接口，如图 (b) 所示。

下面介绍常用的 MIPS 硬件模拟器 PCSpim 和多平台模拟器 QEMU 的差别。PCSpim 是一款纯正的 MIPS 硬件模拟器，它从性质上完全等价于裸 MIPS 硬件，因此如果将一个 MIPS 语言编写的汇编程序在其上运行，其运行层级就如图 (c) 所示，于是对于程序而言，其运行环境相当于在裸硬件上运行。

而 QEMU 则不一样了，QEMU 不仅可以模拟出 MIPS 硬件，还能为在其上运行的程序提供一个简单的操作系统接口，因此在本次实验中，QEMU 扮演了两个层次的角色，即既模拟硬件层，又模拟了 OS 层，那么用户程序其实是运行在一个由操作系统提供运行时环境的状态，如图 (d) 所示。



随着计算机技术的进步，我们本次实验也尝试在有 OS 环境下进行，意义如下：

1. 操作系统提供的运行环境（各种系统调用）可以帮助我们更快地编写和运行程序；
2. 能够提前对操作系统的角色和意义有所认知，为后续的操作系统专业课程打下基础（重要）；
3. 复用上一次实验的实验环境，无需另外搭建实验环境，将两个汇编语言实验整合在同一平台上；
4. PCSpim 模拟器的运行原理是直接对汇编语言文件进行文本解释，并没有生成二进制代码用于程序执行，这和汇编语言真正的执行方式有一定出入，而 QEMU 则是对真正的 MIPS 二进制文件进行文件分析和程序装载，程序的运行过程和运行环境显得更加真实。

Q. 什么是交叉编译？本实验中交叉编译是如何体现出来的？

A. 每个二进制程序都有自己的指令集编码模式，可执行文件的指令集与运行平台的指令集不一致时，程序将无法运行。同时，通常只能在一个平台上生成本平台所能运行的指令集代码，即一般编译。那么和一般的编译相对，我们可以在一台某种指令集的机器上，生成另一种指令集的程序，这个过程就成为**交叉编译**。在本实验中，我们的代码是在 VirtualBox 虚拟机中编写和编译的，然而虚拟机的特性决定了其平台指令集与宿主机指令集相同（目前一般 PC 使用的都是 Intel/AMD 的 x86 体系 CPU），要编译能在 MIPS 指令集平台运行的程序，就必须用到交叉编译，在 x86 平台下生成 MIPS 指令集的二进制可执行文件。而我们实验中所用到的汇编器 `mips-linux-gnu-as` 和链接器 `mips-linux-gnu-ld` 就是在 x86 环境下生成 MIPS 格式二进制可执行程序的必备工具。

下面这幅图展示了一个程序从高级语言代码到成功运行的生命周期，图中含有三条线路 A，B 和 C：

由于我们使用汇编语言编写程序，就没有了预编译（将 C 语言宏进行展开和替换）和编译（将高级语言程序转化为汇编语言文件）两步，**直接从汇编语言文件开始**。路线 A 展示过程是一个一般的编译过程，那么交叉汇编和这个过程的唯一差别就是如 路线 B 所示，使用交叉编译工具链，即 `mips-linux-gnu-xx` 工具。这样生成的二进制文件，其指令集是目标平台的（本实验中为 MIPS），可以在模拟了 MIPS 指令集的 QEMU 模拟器上运行。

顺便看一下上个问题中所提到的利用 PCSpim 进行模拟运行的过程，根据其官方描述和项目开源代码来看，其执行原理是直接将文本形式的汇编语言文件进行解释并模拟其操作，整个过程中并没有直接执行过机器码，这个过程和 python 等解释性语言类似，但是与汇编语言程序的真实执行情况不相同，因此本实验中采取交叉编译的方案，利用 QEMU 模拟器真实模拟代码的运行过程。

Q. 交叉编译出来的 hello 文件和采用一般编译工具链得到的 hello 有何不同?

A. Linux 环境下的二进制代码文件和可执行文件均采用 ELF 格式，文件格式和指令集是没有关系的，Linux 内核都会使用相同的机制加载和运行 ELF 文件，有兴趣的同学可以使用交叉反汇编器 `mips-linux-gnu-objdump` 或交叉 ELF 文件解析工具 `mips-linux-gnu-readelf` 查看最终交叉编译得到的二进制文件，看看和一般的 x86 二进制可执行程序有什么相同点、相似点和不同点。
