

目录

目录	1
第二部分 操作系统实践	1
第五章 实验 1. Hos 操作系统的构建与运行	2
5.1 实验目的	2
5.2 实验内容	2
5.3 实验背景及原理	13
第六章 实验 2. Hos 开发调试环境安装	14
6.1 实验目的	14
6.2 实验内容	14
6.3 实验背景及原理	17
第七章 实验 3. 从应用到内核	28
7.1 实验目的	28
7.2 实验内容	28
7.3 实验背景及原理	30

第二部分 操作系统实践

通过本书第一部分的学习，读者已经在 NEXYS4 的基础上开发了 MIPSfpga 系统，该系统包含一个标准 MIPS 处理器，以及必要的接口设备。本书的第二部分，将引导读者在第一部分所开发的 MIPSfpga 上，运行一个小型的操作系统。接下来，将讲解如何在该小型操作系统上进行开发、调试，以及开发具有挑战性的任务。

第五章 实验 1. Hos 操作系统的构建与运行

5.1 实验目的

在本实验中，读者将学习在自己的个人电脑上安装构建（build）Hos 操作系统的环境，以及将所生成的镜像下载到 MIPSfpga 开发板、并将其运行起来的方法。我们假设读者的个人电脑上安装的是 Windows 操作系统（Windows7 或 Windows10），对于 MacOS 的用户，可以对以下的安装过程中所介绍的软件，选择对应的 MacOS 上的软件即可。

5.2 实验内容

Hos 操作系统的构建涉及到较多的软件工具，其中包括：Cygwin、交叉编译器（mips-sde）、Putty、Vivado、OpenOCD 等。因为 Vivado、交叉编译器（mips-sde）以及 OpenOCD 已经在第一部分做过介绍，所以在本实验中，读者将安装这些软件（见 5.2.1），并在安装完成后构建并运行 Hos 操作系统（见 5.2.2 和 5.2.3）。

5.2.1 安装开发环境

1) Cygwin 的安装

Cygwin 是一个在 Windows 环境下运行的类 Linux 环境，它能够在 Windows 下提供 Linux 环境以及很多 Linux 工具。在本实验中，我们即将用到 make、gcc、perl 这些基本工具。make 工具用于解析 Hos 的 makefile 文件，gcc 用于编译 Hos 的源代码，perl 用于解释执行 make 过程中 Hos 所带的一些脚本程序。读者可以到以下网站下载 Cygwin 的安装程序：

<http://cygwin.com>

需要指出的是，读者应根据自己的运行环境来选择安装文件。如果运行环境是 32 位的，就应下载 setup-x86.exe；如果运行环境是 64 位的，就需要下载 setup-x86_64.exe。查看自己的个人电脑的运行环境是 32 位还是 64 位的任务相对简单，且能够在互联网上找到大量的介绍，所以我们就不在这里赘述了。

另外，Cygwin 的不同版本也可能会有细微的差距（我们以 Cygwin 的 2.6.0 版本作为讲述蓝本）。但这些细微差距应该不会对我们之后的实验构成太大影响，因为我们只用到了它的几个基本软件包。

NOTE: [The previous Cygwin version 2.5.2 was the last version supporting Windows XP and Server 2003.](#)

For more information see the [FAQ](#).

Current Cygwin DLL version

The most recent version of the Cygwin DLL is [2.6.0](#). Install it by running [setup-x86.exe](#) (32-bit installation) or [setup-x86_64.exe](#) (64-bit installation).

Use the setup program to perform a [fresh install](#) or to [update](#) an existing installation.

Note that individual packages in the distribution are updated separately from the DLL so the Cygwin DLL version is not useful as a general Cygwin distribution release number.

Support for Cygwin

For all Cygwin-related questions and observations, please check the resources available at this site, such as the [FAQ](#), the [User's Guide](#) and the [mailing list archives](#). If you've exhausted these resources then please send email to an [appropriate mailing list](#). This includes observations about web pages, setup questions, questions about where to find things, questions about why things are done a certain way, questions about the color preferences of Cygwin developers, questions about the meaning of the number 42, etc.

图 5.2.1 下载 cygwin 的安装文件

为了叙述方便，我们假设已经将 setup-x86_64.exe 文件放在 D:\Hos\tool-chains 目录中，并希望将 Cygwin 安装在 D:\Hos\tool-chains\cygwin64 目录中。我们这里用到的环境是 64 位的 Windows10 的专业版，其他开发环境（如 Windows7 或者其他 32 位版本）的安装过程类

似。

现在开始安装过程，运行 `setup-x86_64.exe`：

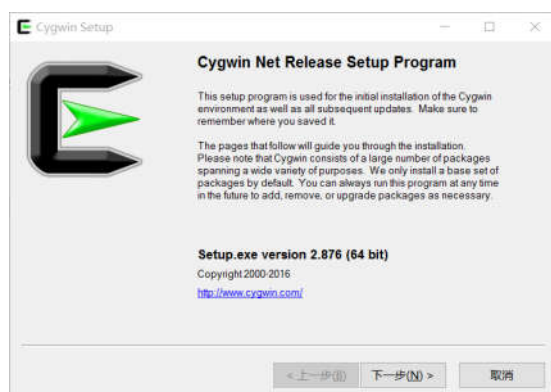


图 5.2.2 运行 cygwin 的安装文件

点击“下一步”，并选择“Install from Internet”（从网络安装）：

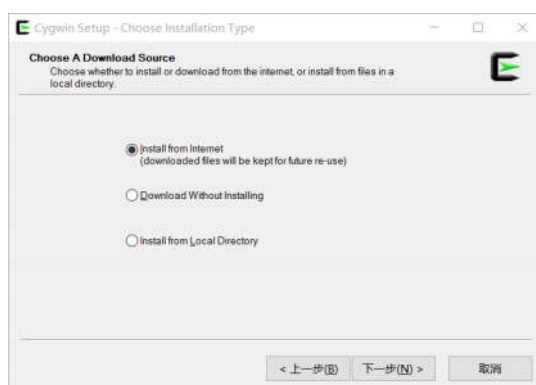


图 5.2.3 选择 Cygwin 的安装方式

接下来选择安装目录以及安装包的缓存目录：

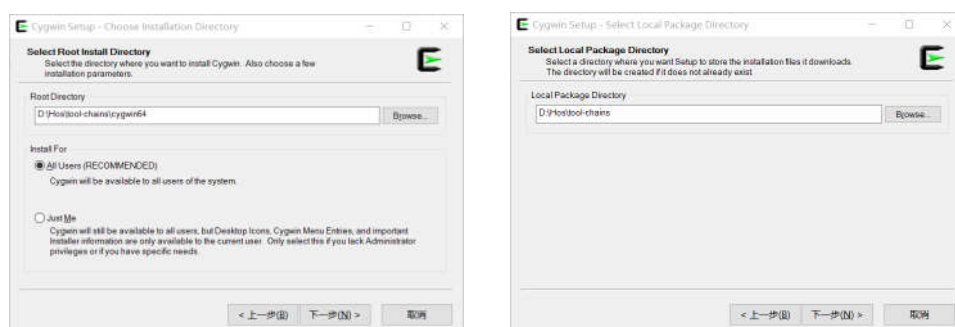


图 5.2.4 选择 Cygwin 的安装路径

再选择安装源，需要注意的是，读者可根据自己的网络连接状况选择是否使用代理以及最近的安装源。这里，我们选择的是位于教育网的镜像 `http://mirrors.neusoft.edu.cn`。

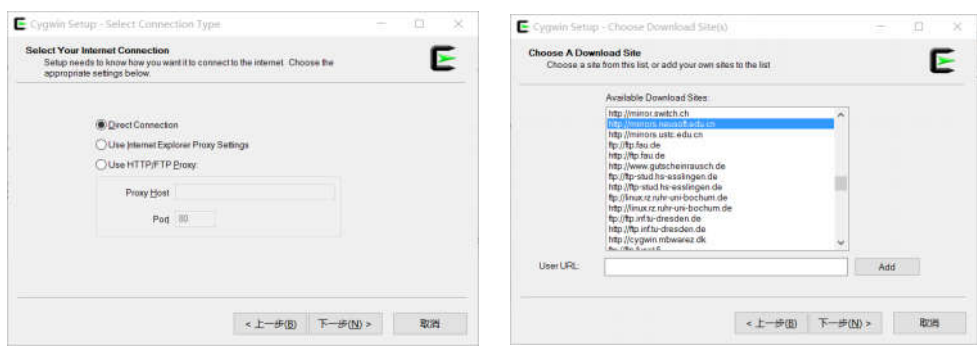


图 5.2.5 选择 Cygwin 的安装源

在点击“下一步”后，Cygwin 的安装文件会从网络上下载基本的安装文件，并显示如下
的界面。

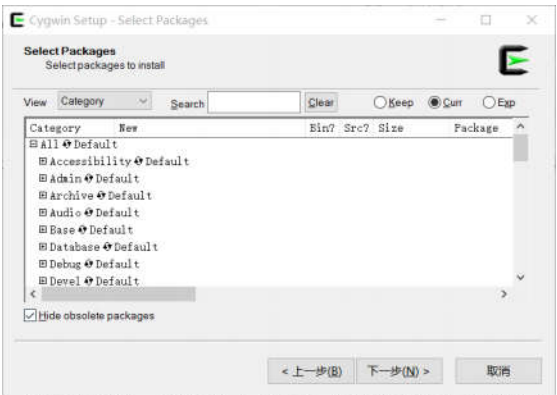


图 5.2.6 Cygwin 的默认配置

实际上，这个这个界面给出的是即将安装到 D:\Hos\tool-chains\cygwin64 目录（注意：目录名中不要出现空格）的 Cygwin 的软件包，且只包含最基本的部分。该默认配置并不包括我们即将要用到的软件工具，所以，我们需要在这里安装额外的软件包。首先是安装 make 软件包，方法是在以上界面的 Search 输入栏中输入“make”，并在界面中间的安裝列表刷新后，点开 Devel 前的“田”号，并在展开的列表中的“make: The GNU version of the 'make' utility”前的“Skip”上单机，直到显示出即将要安装的 make 软件工具的版本为止。

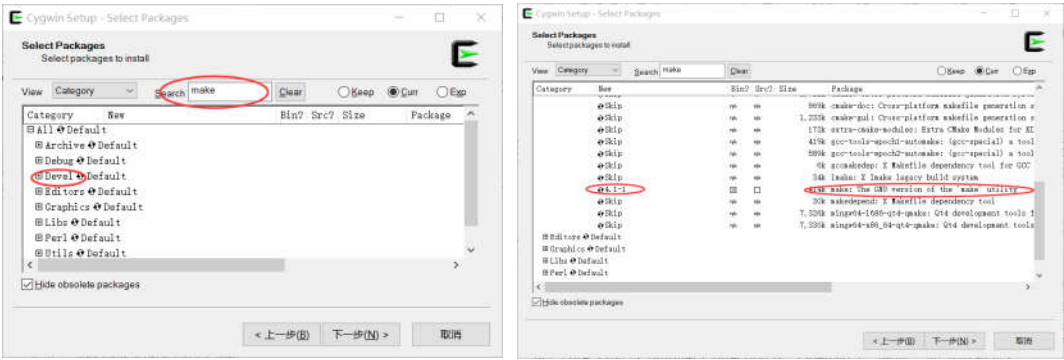
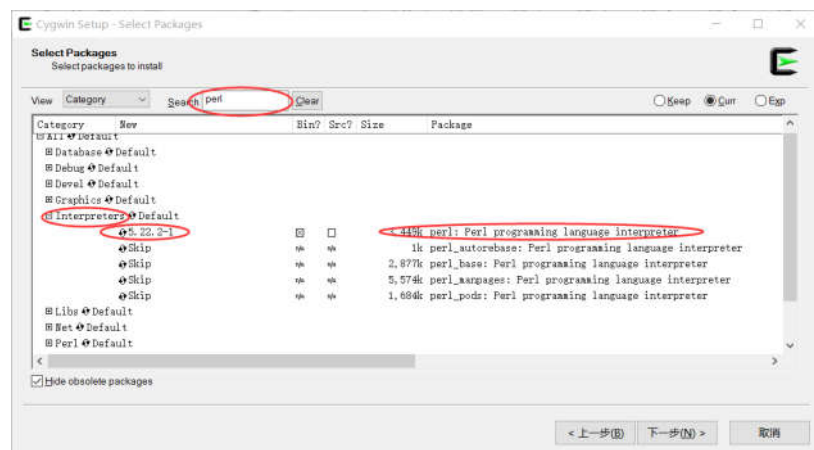


图 5.2.7 在 Cygwin 中安装 make

The screenshot shows the 'Select Packages' window with the 'Search' field set to 'gcc'. The list of packages is filtered to show GCC-related items. The 'gcc-core' package is highlighted in red. The 'gcc-objc' package is also highlighted in red. The 'gcc-objc++' package is highlighted in red. The 'gcc-objc++' package is highlighted in red.

接下来安装 perl，并在输入 perl 后展开“Interpreters”，并选择“perl: Perl programming language interpreter”，如下图所示：



完成以上步骤后，就可以点击“下一步”，并开始真正的下载和安装了。在安装的一个界面上，选中“Create icon on Desktop”，如下图所示：

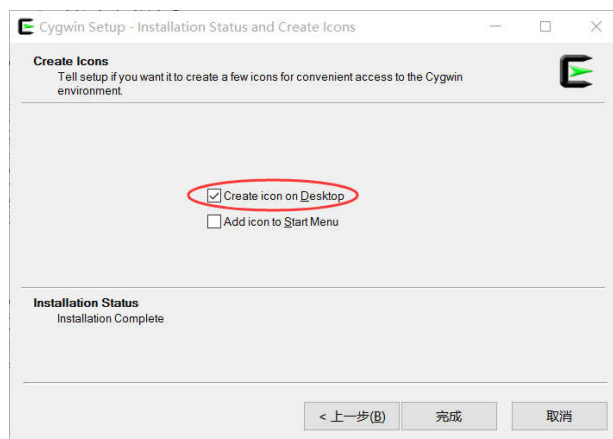


图 5.2.10 完成 Cygwin 的安装

接下来，我们测试一下 Cygwin。方法是双击桌面上的“Cygwin64 Terminal”图标，出来的界面应该如下图所示：

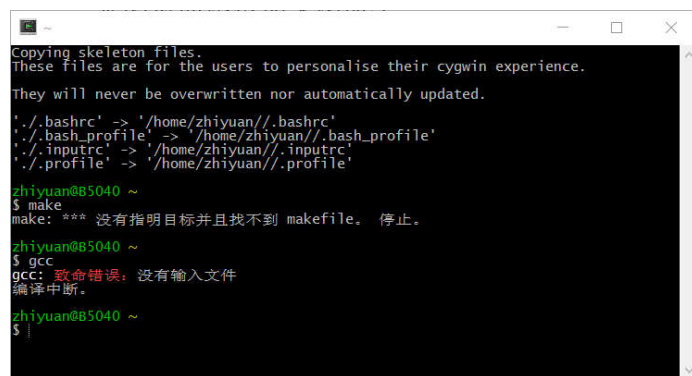


图 5.2.11 运行 Cygwin

在该终端下，我们就可以测试一下之前安装的 make 和 gcc 软件包。因为没有输入文件，这两个命令肯定会报错，但只要不出现“未找到命令”的错误就不会影响之后的实验了。

最后，我们要将 Cygwin 的安装目录下的 bin 子目录（D:\Hos\tool-chains\cygwin64\bin 目录）加入到系统路径中，方法是：“控制面板”→“系统和安全”→“系统”→“高级系统设置”¹，当显示“系统属性”对话框后，点击“环境变量”按钮，并在出现“环境变量对话框”后，选择“新建”按钮（如果已经定义了 Path 环境变量，则可选择“编辑”按钮）。

¹ 因为我们使用的示例系统是 Windows10，它将环境变量分为了系统变量和用户变量，所以我们采用了添加用户变量的方法。而 Windows7 系统只有系统环境变量，所以如果是在 Windows7 下，就需要将 cygwin 的 bin 子目录路径加到已有的 Path 系统环境变量中。方法是将路径添加到 Path 对应的字符串之后，并用分号将新添加的路径和已有的字符串隔开。



图 5.2.12 添加环境变量（Windows10）

在接下来弹出的“新建用户变量”对话框中，输入变量名为“Path”，变量值为“D:\Hos\tool-chains\cygwin64\bin”，如下图所示：



图 5.2.13 环境变量 Path

在输入完成后点击“确定”按钮，就将 Cygwin 加入了我们的开发环境，

2) 下载 Hos 源码

接下来访问 <https://github.com/mrshawcode/hos-mips>，并下载 Hos 源代码。读者可以使用 git 工具来对源代码进行复制（clone 命令）。实际上，这也是较好的方法，因为这样可以跟踪自己对代码所做的所有改动。但对于不熟悉 git 工具的读者，则可以直接下载 zip 包，并在本地进行解压操作，如下图所示。

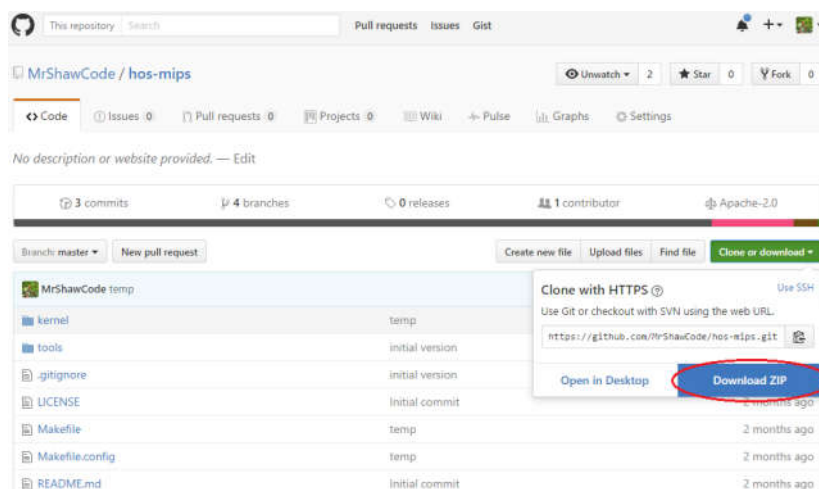


图 5.2.14 下载 Hos 源代码

现在，我们假设读者已经下载 Hos 源代码，并将其解压到 D:\Hos\hos-mips-master\目录（注意：目录名中不要出现空格）下，我们将在该目录下看到以下内容：

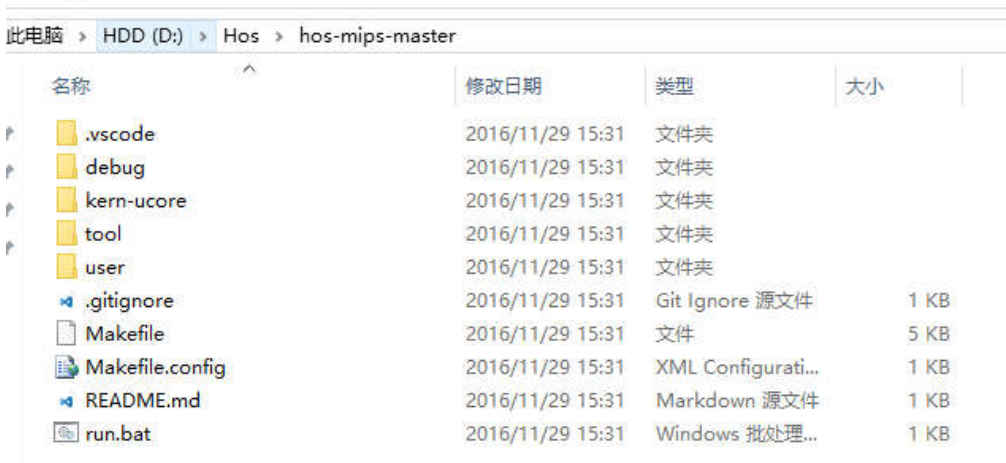


图 5.2.15 解压 Hos 源代码

源代码中目录及文件的说明见下表。

表 5.2.1 Hos 源代码中目录及文件的简单说明

文件/文件夹	说明
.vscode 目录	存放 vscode 的配置文件。当在 vscode 中引入后，该子目录下的 launch.json 和 tasks.json 将生效（将在下一个试验中用到）。
debug 目录	存放用于 Hos 运行的工具程序，例如 JTAG 的启动于配置文件、用于显示 Hos 运行结果的 Putty、以及 mips-sde-elf-gdb 的配置文件（startup-ucoore.txt）等。
kernel-ucoore 目录	Hos 操作系统内核的源代码。
tool 目录	用于生成 sfsimage 镜像的工具。
user 目录	用户态代码。
.gitignore	用于 git 的配置文件（与我们之后的实验无关）。
Makefile	主 make 文件。
Makefile.config	主 make 文件的配置文件，通过该文件可配置交叉编译器 等。
README.md	对于 Hos 编译与使用的简单说明文件。
run.bat	运行文件，在 make 命令执行后，如果成功生成了内核，则可以执行此批处理程序，在 Nexys4 DDR 开发板上运行 Hos。

至此，我们的环境配置就完成了。接下来，我们将构建 Hos 内核，并在本书的第一部分所构造的 MIPS 系统上，运行该操作系统。

5.2.2 构建 Hos 镜像

我们将使用 Cygwin 来构建 Hos 系统。启动 Cygwin，并进入 Hos 源代码所在的目录，如下图所示：

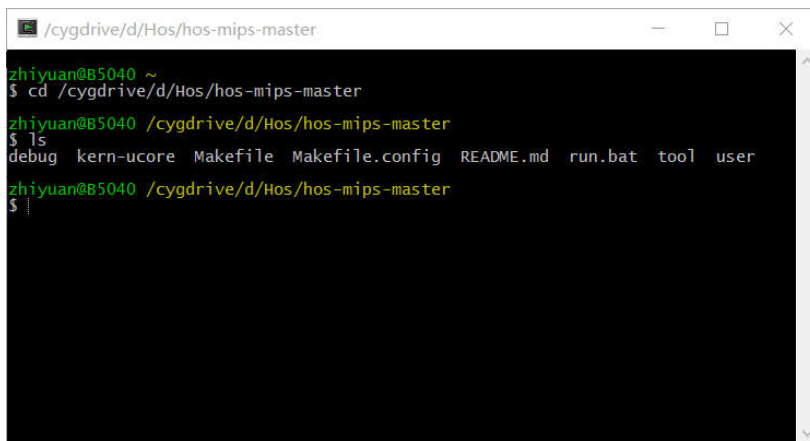


图 5.2.16 启动 Cygwin 并进入 Hos 源代码目录

这里需要注意的是，Cygwin 中使用的路径是 cygpath。我们之前放置 Hos 源代码的目录（也就是 D:\Hos\hos-mips-master\）对应的是 /cygdrive/d/Hos/hos-mips-master，所以转到该目录下的命令是：

```
$ cd /cygdrive/d/Hos/hos-mips-master
```

接下来，输入“make”命令开始构建过程。此时，应确定 Cygwin 以及交叉编译器所在的目录已经在系统路径中了。在构建过程中出现找不到某命令的错误，一般是由于命令所对应的工具不在系统路径中所导致的，这时应检查是否已经正确设置系统路径（如路径中不要出现空格）。

```
$ make
```

构建时间大概会有 1-2 分钟，取决于机器的速度。构建成功后，会出现以下（类似的）界面：

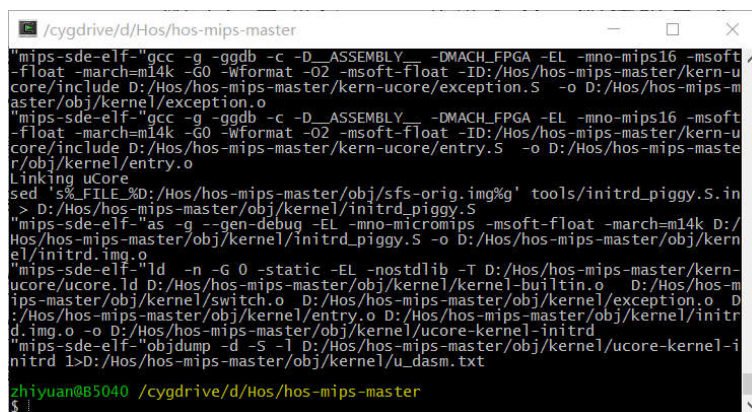


图 5.2.17 Hos 系统的构建过程

为了进一步确保构建过程的正确性，可检查是否正确地生成了 Hos 的系统的镜像，采用以下命令：

```
$ ls ./obj/kernel/ucore-kernel-initrd -alh
```

该命令的输出如下图所示：

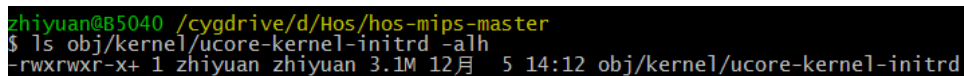


图 5.2.18 Hos 的系统镜像

也就是说，Hos 的系统镜像文件（ucore-kernel-initrd）的大小为 3.1MB 左右。

5.2.3 Hos 的运行

在成功生成 Hos 系统镜像文件后，就可以在本书第一部分所构造的 MIPS 系统上运行该系统了。这里，我们假设读者已经按照本书第一部分的实验，在 Nexys4 DDR 开发板上已经下载了 MIPSfpga 的 bitstream 文件。实际上，读者可以将 vivado 加入到系统路径（Path）中，并打开 run.bat（见图 5.2.21）中的第二行，让 vivado 在 Hos 运行前将标准的 MIPSfpga 所对应的 bitstream 文件下载到开发板中。

但在运行前，需要根据 Nexys4 DDR 开发板所连接的串口端口修改表 6.1 中的 run.bat 文件。方法如下（我们使用 Windows10 系统作为例子，Windows7 系统的过程类似）：

首先，在桌面“我的电脑”图标上单机右键，选择“属性”。在出现的“系统”窗口中，选择左上方的“设备管理器”。如下图所示：

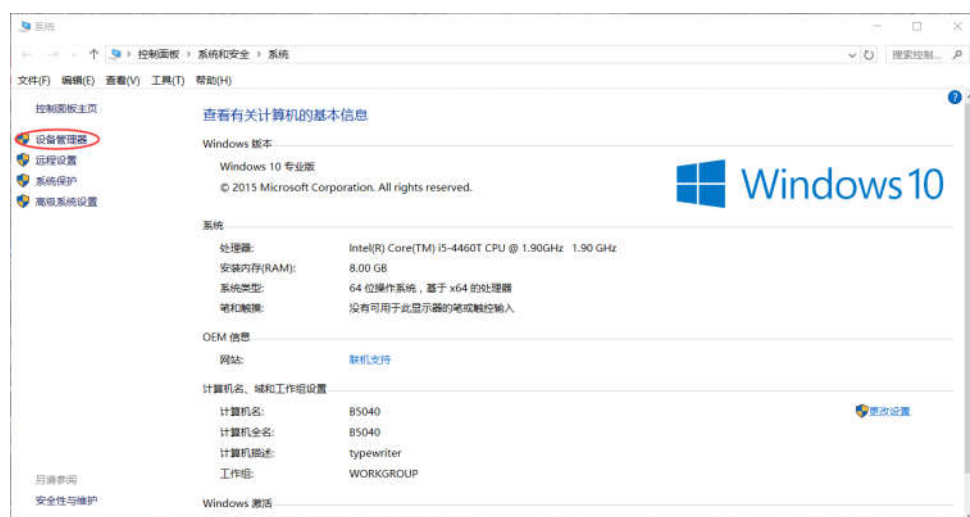


图 5.2.19 “系统”窗口

在接下来出现的“设备管理器”窗口中，展开“端口（COM 和 LPT）”选项，可以看到如图 5.2.20 所示的界面。

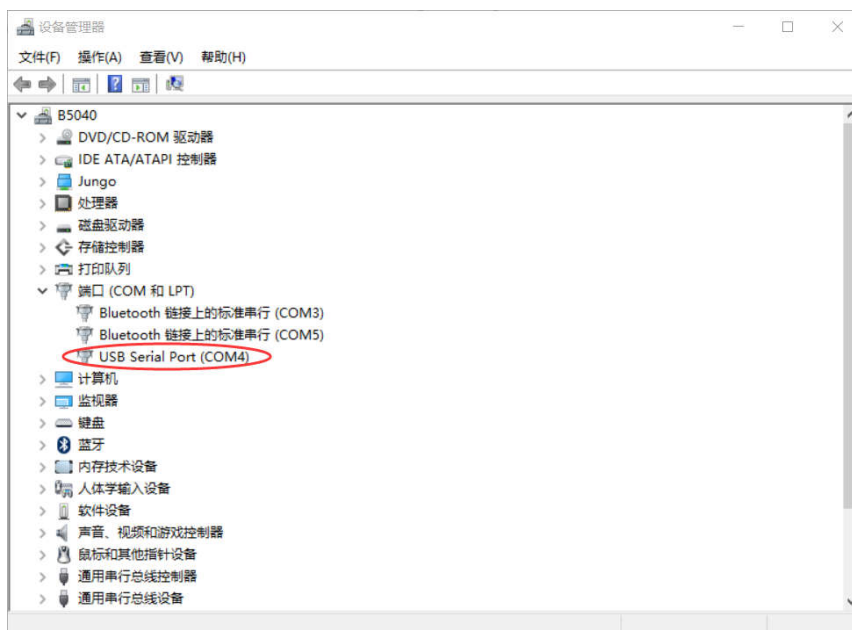


图 5.2.20 查看 Nexys4 DDR 开发板所对应的 COM 端口

该界面红色椭圆标注的部分就告诉我们，Nexys4 DDR 开发板所连接的是第 4 号 COM 端口。接下来，读者就可以采用任意编辑器打开 Hos 源代码目录中的 run.bat 文件，并修改该文件的第三行，写入对应的 COM 端口（见图 5.2.21）。

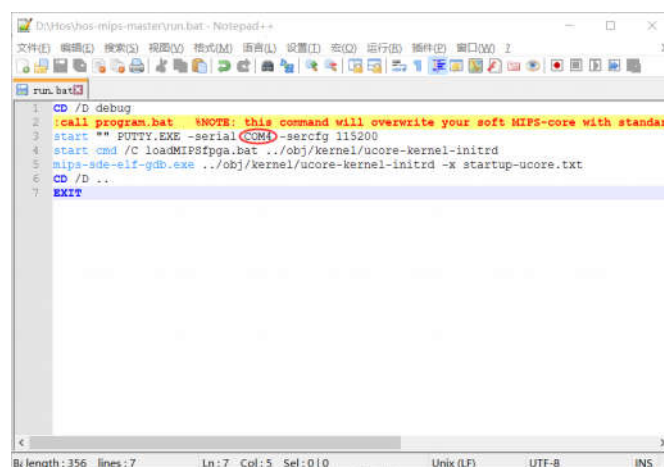


图 5.2.21 run.bat 文件

这里，因为 run.bat 中默认的是 COM4，所以在我所在的 Windows10 系统上就不用修改这个参数。但如果在读者的系统环境中，Nexys4 DDR 开发板所连接的不是 COM4（例如 COM6），那么就需要根据实际的情况，修改 run.bat 中的 COM4（如修改为 COM6）。

在修改了 run.bat 中对应的 COM 端口后，就可以在 Cygwin 中直接执行该批处理命令，以运行 Hos 操作系统了！执行该命令后，Cygwin 中的显示会变成图 5.2.22 所示：

```
cygdrive/d/Hos/hos-mips-master
>https://sourcery.mentor.com/GNUToolchain/>.
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 ../obj/kernel/ucore-kernel-initrd...done.
0x00000000 in ?? ()
The target is assumed to be little endian
JTAG tap: mAUP.cpu tap/device found: 0x00000001 (mfg: 0x000, part: 0x0000, ver:
0x0)
target state: reset
entered debug state at PC 0xbfc00000, target->state: halted
target state: halted
target halted in MIPS32 mode due to debug-request, pc: 0xbfc00000
Breakpoint 1 at 0x9fc01200
entered debug state at PC 0x9fc01200, target->state: halted

[Remote target] #1 stopped.
0x9fc01200 in ?? ()
Loading section .text, size 0x30400 lma 0x80001000
Loading section .data, size 0x25a440 lma 0x80031400
Start address 0x80001000, load size 2664512
Transfer rate: 88 KB/sec, 4086 bytes/write.
```

图 5.2.22 run.bat 的执行结果

且会弹出以下两个窗口：

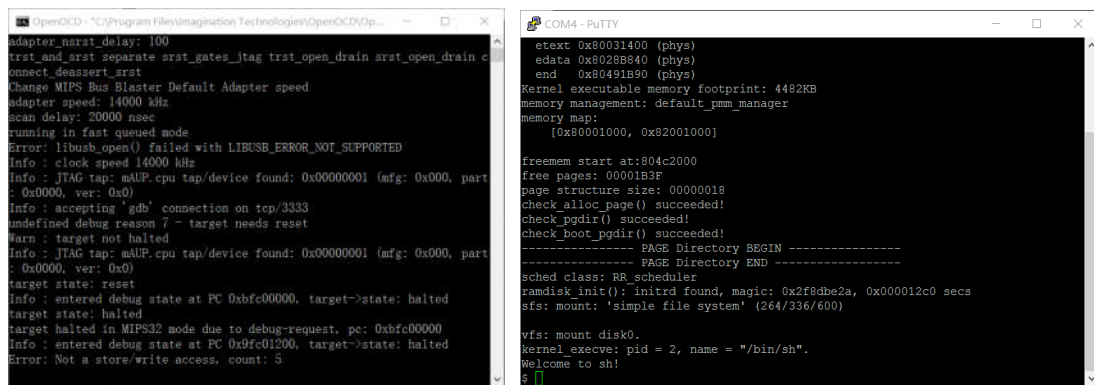


图 5.2.23 OpenOCD 和 Putty 窗口

图 5.2.23 中，左边的是 OpenOCD 的执行结果窗口，其实质是 JTAG 驱动以及 JTAG 中的 GDBServer 的运行结果；右边的是 Putty 的执行结果窗口（我们在 Hos 的源代码中加入了 Putty.exe，读者无需专门安装 Putty），其中的输出表明 Hos 操作系统已经运行起来了，读者可自行在该窗口中输入简单的 UNIX 命令，如“ls”等，观察其执行结果。

实际上，为了理解这些输出，读者可以回到 run.bat 文件（见图 5.2.21），理解其中的动作：

- 该文件的第一行是进入 Hos 源代码中的 debug 目录。
- 第二行是调用 vivado 将 debug 目录中准备的一个标准 MIPS 软核（bitstream 文件

mipsfpga_wrapper.bit）写到 Nexys4 DDR 的 FPGA 中。需要注意的是，这个文件是用于测试 Hos 的，不提供源代码，所以读者还是要按本书中第一部分的内容来构造自己的 MIPS 软核。也正是出于该考虑，这一行中的命令实际上是被注释掉的！对于跳过本书第一部分的读者来说，可以将该行打开，使用标准的 MIPS 软核，但在这样做之前，要确保 vivado 所安装的目录在系统的路径中。

- 第三行是启动 Putty，让它以 115200 的波特率来连接 Nexys4 DDR 开发板，它的执行结果会弹出图 5.2.23 中右边的窗口。

- 第四行是启动 debug 目录中的 loadMIPSfpga.bat 命令，该命令将启动 OpenOCD（也就是 JTAG），并弹出图 5.2.23 中左边的窗口。

- 第五行是启动交叉编译器中带的 mips-sde-elf-gdb.exe，并自动执行 debug 目录中的 startup-ucore.txt 配置脚本，该配置脚本将加载 Hos 系统到 Nexys4 DDR 开发板的内存中，并开始它的执行。需要注意的是，这一行在使用 vscode 调试 Hos 时（第六章的实验）需要被注释掉！因为 vscode 会在调试过程中自行调用 mips-sde-elf-gdb.exe。

5.3 实验背景及原理

5.3.1 Hos 简介

Hos 是基于清华大学的 ucore，开发的适应于 MIPSfpga 平台的一个小型操作系统。对于 ucore 操作系统，读者可访问陈渝老师的 github 主页（https://github.com/chyyuu/ucore_pub），通过阅读陈渝和向勇老师编著的《操作系统实验指导》（2013 年 7 月，清华大学出版社。），并完成里面设计的实验来了解 ucore 操作系统的具体内容。

需要指出的是，ucore 操作系统是面向 x86 体系结构开发的。实际上，细心的读者可以在陈渝老师的 github 主页上找到 ucore 操作系统的多处理器支持版本 ucore-plus（https://github.com/chyyuu/ucore_os_plus），该版本扩展了 ucore，使其能够通过构建选项支持多平台（如 arm、mips）等。Hos 在 ucore-plus 的基础上做了大量的修改，且为了避免和 ucore-plus 可能出现的名字混淆的情况，我们将实验中用到的操作系统起名为 Hos。它可以看成是 ucore 操作系统面向 MIPSfpga 开发板的一个新的分支，而并非另起炉灶。

Hos 与 ucore-plus 存在的不同体现在以下几点：

- 1) Hos 面向的平台是读者在第一阶段开发的 MIPSfpga 平台，所以去掉了 ucore-plus 中的多体系结构支持部分（如 x86、arm 等），使得代码更加简洁。这样做的目标是，帮助读者在其后进行的开发过程中将精力聚焦于 MIPSfpga 平台。

- 2) 虽然 MIPSfpga 平台也是 MIPS 体系结构的一个实例，但它与标准的 MIPS32 平台（也就是 ucore-plus 所考虑的 MIPS 平台）仍然存在着大量的不同，例如其外设、接口，TLB 的设计细节等，Hos 为 MIPSfpga 平台进行了专门的定制。所以，从这个角度来看，Hos 可以视作一个专用（而非通用）的操作系统。

- 3) 由于对 ucore 的 bootloader、内存以及虚存管理等部分进行了大规模的裁剪与简化，使得 Hos 的代码更为精简（总代码量缩减到 2 万行），更加适合初学者。

最后，Hos 对 ucore 的裁剪过程中考虑到了与本书中操作系统实践部分的适应，所以更强调延续性和铺垫性：这一部分的内容在延续读者在本书第一部分所做的 MIPSfpga 处理器工作的同时，也为本书的第三部分中的系统实战打下基础。

需要强调的是，由于写作目的的不同，本书在操作系统实践部分所涵盖的内容与陈渝和向勇老师编著的《操作系统实验指导》中所涵盖的内容只存在较小交集，两者互相独立互为补充。对于本书的读者来说，完成其后的所有实验，并不需要先阅读和掌握《操作系统实验指导》中的所有知识点和完成所有实验。然而，对于希望进一步理解掌握操作系统运作规律细节的读者，我们强烈推荐在完成本书所设计的实验后，继续阅读《操作系统实验指导》，并完成其中所设计的 9 个实验！

5.3.2 相关软件工具

在本实验中，我们用到的主要软件工具是 Cygwin，读者可以在互联网上找到大量的对

Cygwin 的安装、使用进行介绍的文章。

由于篇幅和侧重点方面的考虑，本书只对其基本安装过程，以及被使用到的软件包（如 make、gcc 和 perl）的安装过程进行了简单的介绍。然而，Cygwin 实际上在 Windows 平台上模拟了一个几乎完整的 Linux 环境，能够在该环境中运行的 Linux 命令有很多，有兴趣的读者可以在 Cygwin 中加入更多的软件包（如 git 等）。

第六章 实验 2. Hos 开发调试环境安装

6.1 实验目的

在本实验中，读者将安装 Hos 的开发环境——VSCode，使用 VSCode 打开 Hos 源代码，并在该环境中构建内核、执行并调试。

6.2 实验内容

6.2.1 安装 VSCode

我们选择 VSCode（Visual Studio Code）作为 Hos 的开发和调试环境，这是因为它是我们找到的，在 Windows 平台上跟交叉编译器配合最好的软件工具软件。其次，VSCode 能够在 Windows 中运行，这使得我们可以在 Windows 中开发 Hos 这种类 UNIX 内核！读者需要到以下链接处下载 VSCode：

<https://code.visualstudio.com/Download>

需要注意的是，VSCode 发展出了非常多的版本，本书在写作过程中，我们用到的是它的 1.7.2 版本（安装文件有 32MB），更新的版本应该也是可用的，但读者最好选择安装 Windows 7,8,10 平台下的 32 位 VSCode 版本。

完成安装后打开 VSCode，点击屏幕左侧的“扩展”图标，并在出现中间的扩展窗口后，输入“gdb”，从而安装 Native Debug。

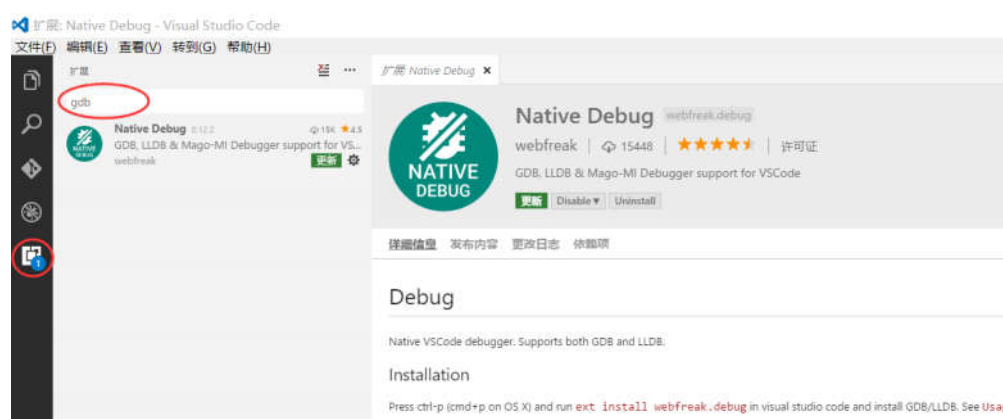


图 6.2.1 在 VSCode 中安装 Native Debug

安装完成后，应重启 VSCode 以进行之后的实验。

6.2.2 使用 VSCode 编辑、构建和调试 Hos

首先打开 VSCode，左键单击左边的“资源管理器”，选择中间出现的“打开文件夹”按钮，如图 6.2.2 所示：

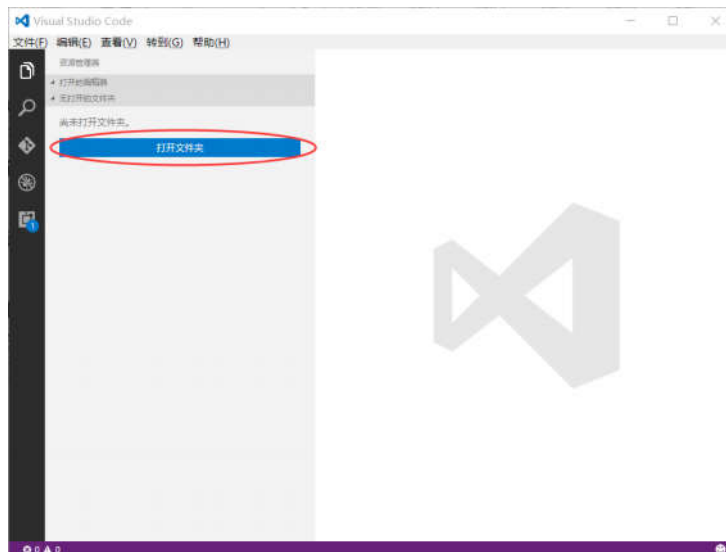


图 6.2.2 用 VSCode 打开 Hos

接下来，在“打开文件夹”窗口中，浏览到 Hos 源代码所在的目录（例如我们用的 D:\Hos\hos-mips-master）。打开该目录后，VSCode 会在中间的“资源管理器”窗口中列出该目录中的所有子目录和文件。浏览“资源管理器”窗口，并打开 kern-ucore 子目录下的 init.c 后，VSCode 的窗口如图 6.2.3 所示。

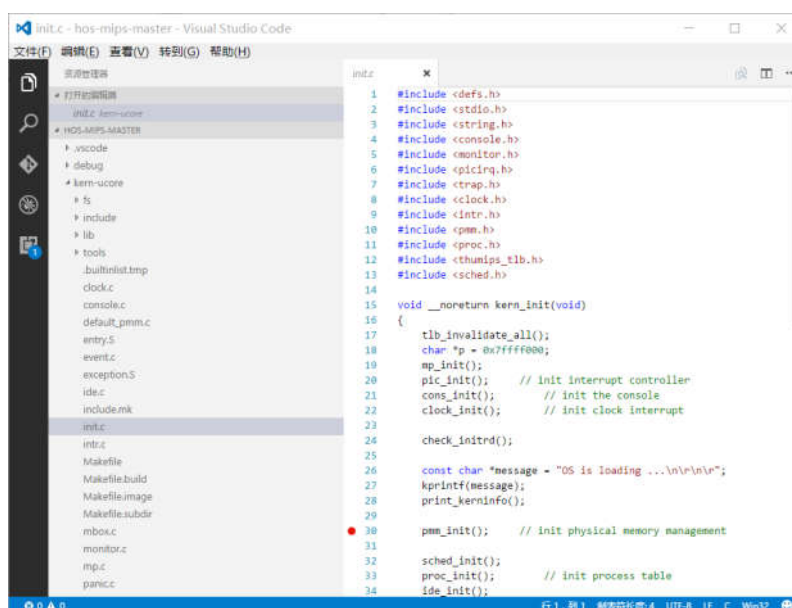


图 6.2.3 用 VSCode 打开源代码文件

现在，读者就可以开始对 Hos 代码进行浏览、编辑了。我们将在实验 3 开始，在 Hos 中加入新的代码。在本章，我们先介绍如何用 VSCode 对 Hos 代码进行调试。在对 Hos 进行调试前，读者需要关闭之前为了运行 Hos 系统所打开的所有窗口（包括 Cygwin, Putty 和 OpenOCD，见 5.2）。

将鼠标移动到“init.c”窗口中程序行号的左边，VSCode 就会显示出一个浅红色的圆点。

例如在图 6.2.3 中，鼠标的位置在第 30 行行号的左边，所以 VSCode 就在改行行号的左边显示了一个浅色的红点。这时，若单击左键，就会在 init.c 的第 30 行设置一个断点。实际上，init.c 中的 kern_init(void)函数就是 Hos 操作系统的入口，读者可以在该入口函数的任意程序行上设置断点。这里选择第 30 行也是处于演示的目的，并无实质调试目标。

再接下来，打开和修改 Hos 源代码根目录中的 run.bat 文件（此时我们假设读者已经正确构建 Hos 系统，方法见 5.2），该文件的内容如图 5.2.21 所示。这次我们需要修改 run.bat 的第 5 行，修改很简单：只需要在该行的最前面加上一个冒号（“:”）即可，也就是将：

```
mips-sde-elf-gdb.exe ../obj/kernel/ucore-kernel-initrd -x startup-ucore.txt
```

改为：

```
: mips-sde-elf-gdb.exe ../obj/kernel/ucore-kernel-initrd -x startup-ucore.txt
```

其目的是调用 run.bat 的时候，关闭 mips-sde-elf-gdb.exe 的执行。这是因为，在调试过程中，我们将会让 VSCode 来调用 mips-sde-elf-gdb.exe 来执行，所以就不需要在命令行执行该程序了。

完成对 run.bat 的修改后，读者可以打开 Cygwin，通过命令进入 Hos 源代码所在的目录（方法见 5.2 中描述），来到如图 5.2.16 所示的界面。运行 run.bat 文件，系统会弹出如图 5.2.23 所示的 OpenOCD 和 Putty 两个窗口，但这时，Putty 窗口并无内容显示（因为 Hos 系统并未开始运行）。

接下来切换到 VSCode，并按下 F5 键进入调试模式。VSCode 在接收到命令后，会打开“调试控制台”子窗口，调用 mips-sde-elf-gdb.exe 将 Hos 镜像加载进开发板内存（这个过程会耗费 1 分钟左右）。在加载完成后 VSCode 将开始 Hos 操作系统的执行，并在我们预设的断点（init.c 的第 30 行）处停下来。此时，VSCode 的界面如图 6.2.4 所示：

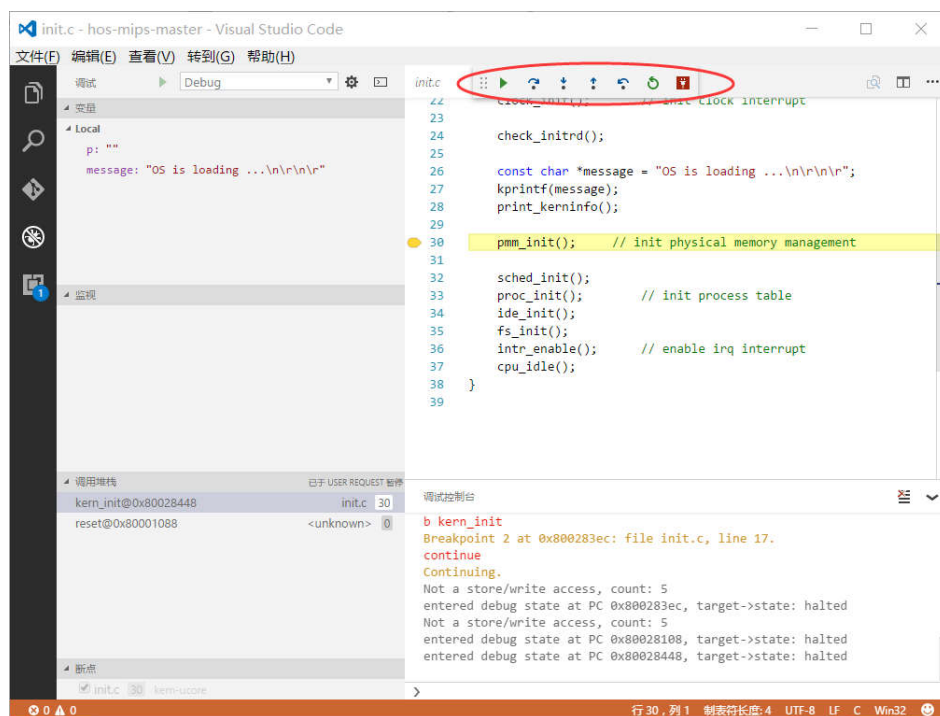
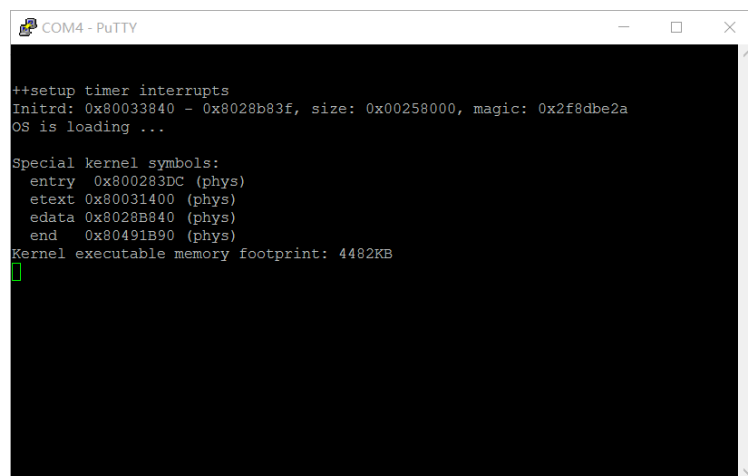


图 6.2.4 VSCode 的调试界面

注意图 6.2.4 中红圈所标注的一组调试按钮，使用这些按钮，读者就能完成常用的调试动作，如继续执行、单步跳过、单步调试、单步跳出、后退等，甚至重启或断开与开发板的连接。

这时，我们可以看到之前弹出的 Putty 窗口也出现了变化，如图 6.2.5 所示。



```
COM4 - PuTTY

++setup timer interrupts
Initrd: 0x80033840 - 0x8028b83f, size: 0x00258000, magic: 0x2f8dbe2a
OS is loading ...

Special kernel symbols:
entry 0x800283DC (phys)
etext 0x80031400 (phys)
edata 0x8028B840 (phys)
end 0x80491B90 (phys)
Kernel executable memory footprint: 4482KB
```

图 6.2.5 调试过程中 Putty 的输出

Putty 中的输出，实际上是因为我们之前让执行中断在 init.c 文件的第 30 行，而该文件中的 kern_init(void)函数在之前已有输出内容。这就意味着，我们可以通过 VSCode 的调试功能并配合 kprintf()函数来共同完成对内核的调试。这些功能，在为 Hos 编写程序并碰到 Bug 时非常有用。

6.3 实验背景及原理

6.3.1 Hos 的构建过程

Hos 的构建过程使用到的重要工具是 Cygwin 中的 make。该工具在被（通过命令行）调用后，会首先寻找当前目录下的 Makefile 文件，解析判断 Makefile 文件中的“伪目标”(Phony)，根据所选择的“伪目标”，判断其依赖关系并执行相应的动作。关于 Makefile 的基础知识，读者可以通过互联网上的资源自行了解，我们推荐读者阅读 CSDN 上的这个帖子：

<http://blog.csdn.net/foryourface/article/details/34058577>

以下，我们仅就 Hos 的构建过程做较为粗略的介绍，感兴趣的读者可以通过修改 Makefile 来体会该构建过程的细节。为了对构建过程有较为直管的认识，我们先来了解 Hos 的目录结构（如图 6.3.1 所示）：

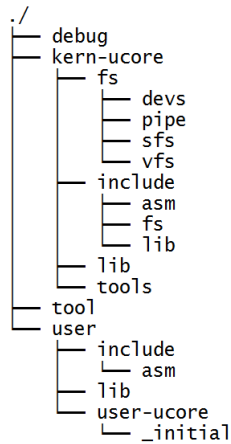


图 6.3.1 Hos 的目录结构

我们在 VSCode 中打开 Hos 根目录下的 Makefile 文件。在该文件的第 73 行，我们看到名字为“all”的伪目标，它代表 make 的缺省目标：

```

72 PHONY+= $(OBJPATH_ROOT)
73 all: sfsimg kernel
74

```

而它有两个较大的依赖目标：sfsimg 和 kernel，也就是说，必须先构建 sfsimg 和 kernel 两个目标才能够最终完成 all 这个伪目标的构建过程。在以下的论述中，我们假定：\$(TOPDIR)=./一直成立，也就是 Hos 所在的顶层目录。

1. sfsimg 的构建

先看 sfsimg，与它相关的定义在 104~117 行：

```

104 ## image
105 SFSIMG_LINK := $(OBJPATH_ROOT)/sfs.img
106 SFSIMG_FILE := $(OBJPATH_ROOT)/sfs-orig.img
107 TMPSPFS := $(OBJPATH_ROOT)/tmpsfs
108 sfsimg: $(SFSIMG_LINK)
109
110 $(SFSIMG_LINK): $(SFSIMG_FILE)
111     @ln -sf sfs-orig.img $@
112
113
114 $(SFSIMG_FILE): $(TOOLS_MKSFS) userlib userapp FORCE | $(OBJPATH_ROOT)
115     @echo Making $@
116     @mkdir -p $(TMPSPFS)
117     @mkdir -p $(TMPSPFS)/lib/modules
...
129     @cp -r $(OBJPATH_ROOT)/user-ucore/bin $(TMPSPFS)
130 ifneq ($(UCORE_TEST),)
131     @cp -r $(OBJPATH_ROOT)/user/testbin $(TMPSPFS)
132 endif
133     @$$(Q)$(MAKE) -f $(TOPDIR)/user/user-ucore/Makefile -C $(TOPDIR)/user/user-ucore
134     @if [ $(ARCH) = "mips" ]; \
135     then \
136         echo "mips"; \
137         cp -r $(TOPDIR)/user/user-ucore/_initial/hello.txt $(TMPSPFS); \
138         rm -f $@; \
139         dd if=/dev/zero of=$@ count=4800; \
140     else \
141         echo -n $(ARCH). " not mips"; \
142         cp -r $(TOPDIR)/user/user-ucore/_initial/* $(TMPSPFS); \

```

```

143             rm -f $@; \
144             dd if=/dev/zero of=$@ bs=256K count=$(UCONFIG_SFS_IMAGE_SIZE); \
145         fi
146         @$(TOOLS_MKSFS) $@ $(TMPSFS)
147         @rm -rf $(TMPSFS)
148
149 endif

```

实际上 sfsimg 的（宏观）构建目标，是生成\$(OBJPATH_ROOT)，也就是./obj 目录（见 Makefile 的第 17 行）下的 sfs.img。为了这个目标，make 工具将依次完成以下工作：

1）创建\$(OBJPATH_ROOT)，也就是./obj 目录；

2）构建生成镜像文件的工具，该工具由\$(TOOLS_MKSFS) 定义，其值为\$(OBJPATH_ROOT)/mksfs，见第 99~102 行：

```

99 TOOLS_MKSFS_DIR := $(TOPDIR)/tool
100 TOOLS_MKSFS := $(OBJPATH_ROOT)/mksfs
101 $(TOOLS_MKSFS): | $(OBJPATH_ROOT)
102     $(Q)$(MAKE) CC=$(HOSTCC) -f $(TOOLS_MKSFS_DIR)/Makefile -C
$(TOOLS_MKSFS_DIR) all

```

这个工具是用 HOSTCC 编译的，实际上也就是我们在第五章安装 Cygwin 时附带安装的 gcc 工具。该工具的源代码在 tool 子目录下，该目录包括一个 Makefile 和一个.c 文件

（mksfs.c）。mksfs 工具的工作原理，是创建一个空白文件，作为一个虚拟的“磁盘分区”。

然后再往其中写入 i 节点等数据结构，从而完成对其“格式化”的过程。需要指出的是，系统

采用了自定义的简单文件系统（simple file system，简称 sfs）来“格式化”虚拟的“磁盘分区”

文件，该文件的格式以及该工具相应的代码，有兴趣的读者可自行阅读。

3）构建用户态库 userlib，见 Makefile 的第 80~81 行：

```

80 userlib: $(OBJPATH_ROOT)
81     $(Q)$(MAKE) -f $(TOPDIR)/user/Makefile -C $(TOPDIR)/user all

```

可以看到，对 userlib 的构建将进入到 Hos 的 user 目录进行，而构建的结果是生成 Hos 的用户态库，也就是 obj/user/ulib.a 文件。

4）构建应用 userapp，见 Makefile 的第 93~94 行：

```

93 userapp: $(OBJPATH_ROOT)
94     $(Q)$(MAKE) -f $(TOPDIR)/user/user-ucore/Makefile -C $(TOPDIR)/user/user-ucore all

```

可以看到，构建过程将进入 user/user-ucore 目录下进行，其目标是构建 obj/user-ucore/bin 下的一系列应用程序。应用程序名字的列表在 Hos 源代码根目录下的 Makefile.config 文件中定义：

```

2 USER_APPLIST:= pwd cat sh ls cp echo mount umount # link mkdir rename unlink lsmod insmod

```

也即是说，包含 pwd、cat、sh、ls、cp、echo、mount、umount 等应用。实际上，这些就是能够在 Hos 中使用的命令的集合，感兴趣的同学可以到 user/user-ucore 目录下阅读这些命令行所对应的源代码。值得注意的是，这些应用的构建用到了在第 3 步中已经构建好的 obj/user/ulib.a 文件。因为 ulib.a 文件中包含一些公用的函数，就使得应用的编程相对简单。

5）在以上步骤都完成后，make 会进行以下动作：

a) 创建 obj/.tmppsfs 目录以及 obj/.tmppsfs/lib/modules 目录（Makefile 的第 116~117 行）；

b) 将 obj/user-ucore/bin 中的文件（带目录）拷贝到 obj/.tmppsfs 目录（Makefile 的第 129 行）；

c) 将 user/user-ucore/_initial/hello.txt 文件拷贝到 obj/.tmppsfs 目录中（Makefile 的第 137 行）；

d) 删除之前的 obj/sfs-orig.img 文件（如果它存在的话），并生成一个空白的 obj/sfs-orig.img 文件（Makefile 的第 138~139 行），其大小为 $4800 \times 512 \text{B} = 2400 \text{KB} \approx 2.4 \text{MB}$ ；

e) 调用之前生成的 mksfs 工具，根据 obj/.tmpsfs 目录中的内容来“格式化”由 obj/sfs-orig.img 文件所模拟的“磁盘分区”（Makefile 的第 146 行）；

f) 删除 obj/.tmpsfs 目录（Makefile 的第 147 行）。

g) 为生成的 obj/sfs-orig.img 文件创建符号链接，也就是生成 obj/sfs.img 文件（Makefile 的第 111 行）。

为了方便读者理解，我们可以把以上步骤的构建过程看作是为计算机“制造”硬盘分区（类似于 Windows 中的 C 盘），并对该分区进行格式化的过程。格式化以后的硬盘分区所用到的文件系统，不是我们耳熟能详的 NTFS、FAT、甚至不是 Linux 中的 EXT 系列文件系统，而是操作系统定义的一个简单文件系统——SFS。

2. kernel 的构建

我们先来阅读 Hos 根目录下的 Makefile 中，对应 kernel 伪目标的构建语句，在该文件的第 77~78 行：

```
77 kernel: $(OBJPATH_ROOT) $(SFSIMG_LINK)
78         $(Q)$(MAKE) -C $(KTREE) -f $(KTREE)/Makefile.build
```

它的含义是转入 kern-ucore 目录（ $KTREE = $(TOPDIR)/kern-ucore$ ，见 Makefile 的第 16 行）下，以 Makefile.build 为主构建文件，开始构建过程。打开 kern-ucore/Makefile.build，并查找它的默认构建伪目标（all），我们看到：

```
35 all: $(KTREE_OBJ_ROOT) $(KERNEL_BUILTIN_O)
36 ifneq ($(UCORE_TEST),)
37         $(Q)touch $(KTREE)/process/proc.c
38 endif
39         $(Q)$(MAKE) KERNEL_BUILTIN=$(KERNEL_BUILTIN_O) -C $(KTREE) -f
$(KTREE)/Makefile.image all
40
41 $(KERNEL_BUILTIN_O): subdir
42         @echo Building uCore Kernel for $(UCONFIG_ARCH)
43         $(Q)$(TARGET_LD) $(TARGET_LDFLAGS) -r -o $@ $(shell xargs < .builtinlist.tmp)
44
45 $(KTREE_OBJ_ROOT):
46         mkdir -p $@
```

我们看到，这里 all 的构建又依赖于两个新的伪目标：\$(KTREE_OBJ_ROOT)、\$(KERNEL_BUILTIN_O)。我们将两个伪目标展开，得到：

```
$(KTREE_OBJ_ROOT) = $(TOPDIR)/obj/kernel
```

```
$(KERNEL_BUILTIN_O) = $(TOPDIR)/obj/kernel/kernel-builtin.o
```

对于第一个伪目标\$(KTREE_OBJ_ROOT)，我们看到，对应的操作非常简单（45-46 行），就是创建 ./obj/kernel 目录而已。然而，第二个伪目标\$(KERNEL_BUILTIN_O)所对应的操作（41-43 行），就比较复杂，它也有自己的依赖伪目标 subdir，所以我们不得不先来看这个 subdir 伪目标所对应的操作：

```
53 subdir: $(KTREE_OBJ_ROOT) $(KCONFIG_AUTOCONFIG) FORCE
54         $(Q)rm -f .builtinlist.tmp
55         $(Q)touch .builtinlist.tmp
56 ifneq ($(UCORE_TEST),)
57         $(Q)touch $(KTREE)/process/proc.c
```



```

58 endif
59 $(Q)$$(MAKE) -f Makefile.subdir OBJPATH=$(KTREE_OBJ_ROOT)
LOCALPATH=$(KTREE) BUILTINLIST=$(KTREE)/builtinlist.tmp

```

这个 `subdir` 伪目标也不简单，它也有 3 个依赖伪目标：`$(KTREE_OBJ_ROOT)`、`$(KCONFIG_AUTOHEADER)`和 `FORCE`，先将宏展开：

```

$(KTREE_OBJ_ROOT) = $(TOPDIR)/obj/kernel
$(KCONFIG_AUTOCONFIG) = $(TOPDIR)/Makefile.config

```

由于 `./obj/kernel` 目录已经创建，而 `Hos` 根目录下的 `./Makefile.config` 早已存在，`FORCE` 无实际动作，所以 `subdir` 的依赖伪目标都已满足，`make` 将继续执行它所规定的动作（54~59）。大体来说有两个动作：创建空白的 `kern-ucore/.builtinlist.tmp` 文件，以及执行 `Makefile.subdir` 中所规定的创建动作，且输入三个环境变量：`OBJPATH`、`LOCALPATH` 以及 `BUILTINLIST`，它们的定义如下：

```

$(OBJPATH)=$(KTREE_OBJ_ROOT)= $(TOPDIR)/obj/kernel
$(LOCALPATH)=$(KTREE)= $(TOPDIR)/kern-ucore
$(BUILTINLIST)= $(KTREE)/.builtinlist.tmp= $(TOPDIR)/kern-ucore/.builtinlist.tmp

```

接下来，打开 `$(TOPDIR)/src/kern-ucore/Makefile.subdir` 文件（以下简称 `Makefile.subdir` 文件）：

```

1 include $(KCONFIG_AUTOCONFIG)
2
3 include Makefile
4
5 DEPS := $(addprefix $(OBJPATH)/, $(obj-y:.o=.d))
6 BUILTIN_O := $(OBJPATH)/builtin.o
7 OBJ_Y := $(addprefix $(OBJPATH)/, $(obj-y))
8
9 all: $(OBJPATH) $(BUILTIN_O) $(dirs-y) FORCE
10 ifneq ($(obj-y),)
11     $(Q)echo $(BUILTIN_O) >> $(BUILTINLIST)
12 endif
13
14 ifneq ($(obj-y),)
15 $(BUILTIN_O): $(OBJ_Y)
16     @echo LD $@
17     $(Q)$$(TARGET_LD) $(TARGET_LDFLAGS) -r -o $@ $(OBJ_Y)
18
19 -include $(DEPS)
20
21 else
22 $(BUILTIN_O):
23     $(Q)touch $@
24 endif
25
26 $(OBJPATH)/%.ko: %.c
27     @echo CC $<
28     $(Q)$$(TARGET_CC) $(TARGET_CFLAGS) -c -o $@ $<
29
30 $(OBJPATH)/%.o: %.c
31     @echo CC $<
32     $(Q)$$(TARGET_CC) $(TARGET_CFLAGS) -c -o $@ $<
33
34 $(OBJPATH)/%.o: %.S
35     @echo CC $<
36     $(Q)$$(TARGET_CC) -D __ASSEMBLY__ $(TARGET_CFLAGS) -c -o $@ $<

```

```

37
38 $(OBJPATH)/%.d: %.c
39     @echo DEP $<
40     @set -e; rm -f $@; \
41         $(TARGET_CC) -MM -MT "$(OBJPATH)/$*.o $@" $(TARGET_CFLAGS) $< > $@;
42
43 $(OBJPATH)/%.d: %.S
44     @echo DEP $<
45     @set -e; rm -f $@; \
46         $(TARGET_CC) -MM -MT "$(OBJPATH)/$*.o $@" $(TARGET_CFLAGS) $< > $@;
47
48 define make-subdir
49 $1: FORCE
50     @echo Enter $(LOCALPATH)/$1
51     -$(Q)mkdir -p $(OBJPATH)/$1
52     +$(Q)$(MAKE) -f $(KTREE)/Makefile.subdir -C $(LOCALPATH)/$1 KTREE=$(KTREE)
OBJPATH=$(OBJPATH)/$1 LOCALPATH=$(LOCALPATH)/$1 BUILTINLIST=$(BUILTINLIST)
53 endef
54
55 $(foreach bdir,$(dirs-y),$(eval $(call make-subdir,$(bdir))))
56
57 PHONY += FORCE
58 FORCE:
59
60 # Declare the contents of the .PHONY variable as phony. We keep that
61 # information in a variable so we can use it in if_changed and friends.
62 .PHONY: $(PHONY)

```

我们发现，这个文件并不长，但是在它的第 1、3 行，分别包含了另外两个文件：
\$(KCONFIG_AUTOCONFIG)以及 Makefile 文件，它们分别对应了\$(TOPDIR)/Makefile.config
和\$(TOPDIR)/kern-ucore/Makefile 这两个文件。其中，前者只是定义了一些构造内核需要的
宏，而\$(TOPDIR)/kern-ucore/Makefile 文件的内容是跟 Makefile.subdir 文件直接相关的，将
其打开，我们看到：

```

1 dirs-y := lib fs
2 obj-y := $(patsubst %.c,%.o,$(wildcard *.c))
3 # obj-y += $(patsubst %.S,%.o,$(wildcard *.S))

```

该文件主要定义了两个变量 dirs-y 和 obj-y，对于当前的 Makefile.subdir 而言，它们的值
为：\$(dirs-y) = lib fs，而\$(obj-y)的值则依赖于当前所处的目录，将目录中所有的.c 文件的扩
展名换成.o，并合并在一起（文件名间加上空格隔开）就是\$(obj-y)的取值了，实际上这也就
是 patsubst 函数的功能。而 Makefile.subdir 而言，\$(OBJ_Y)则是将\$(obj-y)中的所有文件名
加上它的绝对路径，也就是 addprefix 函数的功能。

例如，对于./kern-ucore/fs/pipe 而言（因为.c 文件较少，我们以该目录为例），它的目录
下有 4 个.c 文件，分别是 pipe.c、pipe_inode.c、pipe_root.c 和 pipe_state.c。

则有\$(obj-y)= pipe.o pipe_inode.o pipe_root.o pipe_state.o

而\$(OBJ_Y)的值为：

```

$(OBJ_Y)=
$(TOPDIR)/obj/kernel/fs/pipe/pipe.o $(TOPDIR)/obj/kernel/fs/pipe/pipe.o
$(TOPDIR)/obj/kernel/fs/pipe/pipe_inode.o $(TOPDIR)/obj/kernel/fs/pipe/pipe_root.o
$(TOPDIR)/obj/kernel/fs/pipe/pipe_state.o

```

同时，DEPS 变量也会通过 addprefix 生成，对应到./kern-ucore/fs/pipe 目录，它的取值
为：

```

$(DEPS)= $(TOPDIR)/obj/kernel/fs/pipe/pipe.d $(TOPDIR)/obj/kernel/fs/pipe/pipe_inode.d
$(TOPDIR)/obj/kernel/fs/pipe/pipe_root.d $(TOPDIR)/obj/kernel/fs/pipe/pipe_state.d

```

对于.d文件，它们的生成规则在 Makefile.subdir 文件的 43~46 行，通过 TARGET_CC 变量所定义的工具（也就是 mips-sde-elf-gcc）生成，它的作用是找到.c文件的所有依赖文件，如.h文件等。

了解了这四个重要变量（dirs-y、obj-y、OBJ_Y 和 DEPS）的取值，以及 patsubst 以及 addprefix 函数的功能后，我们接着看 Makefile.subdir 中的 all 伪目标，它有 4 个依赖伪目标：\$(OBJPATH)、\$(BUILTIN_O)、\$(dirs-y)、FORCE。其中 FORCE 伪目标并无直接的动作（第 58 行），\$(OBJPATH)（=\$(TOPDIR)/obj/kernel）已经创建。

\$(BUILTIN_O)的定义在 Makefile.subdir 文件的第 6 行：

\$(BUILTIN_O) = \$(OBJPATH)/builtin.o = \$(TOPDIR)/obj/kernel/builtin.o

也就是它的目标是生成 \$(TOPDIR)/obj/kernel/builtin.o 文件，它对应的动作在 Makefile.subdir 文件的 14~24 行定义。

对于当前目录./kern-ucore 而言，由于存在大量.c文件，如 clock.c、ide.c、monitor.c 等，所以\$(obj-y)和\$(OBJ_Y)并不为空，Makefile.subdir 文件的 15~17 行生效。而\$(BUILTIN_O)伪目标的构建又依赖于\$(OBJ_Y)，也就是说，必须要等\$(OBJ_Y)所定义的所有.o文件生成后，才会最终通过第 17 行的 ld 命令将这些.o文件链接到一个集成文件 builtin.o 中。而.d文件，也就是.c文件的所有依赖文件这时会被加进来（第 19 行），而.c文件会通过 Makefile.subdir 文件的 32 行通过交叉编译器所编译，从而生成所有\$(BUILTIN_O)所依赖的.o文件，而最终生成\$(BUILTIN_O)所定义的 builtin.o 文件。这样，对于当前目录./kern-ucore 而言，它所对应的\$(TOPDIR)/obj/kernel/builtin.o 就生成了。

是不是到这里万事大吉了呢？实际上，还远没有结束。因为对于 Makefile.subdir 而言，它的最终伪目标（all），还有一个依赖目标，那就是\$(dirs-y)！在之前的叙述中，我们知道对于./kern-ucore 目录而言，它的\$(dirs-y) = lib fs，所以最终目标 all 还有两个“隐性”的依赖目标，就是 lib 和 fs！

而我们在 Makefile.subdir 中并未找到 lib 和 fs 这两个伪目标的定义，那么它们是在哪定义的呢？让我们查看 Makefile.subdir 的第 48~55 行，特别是第 55 行：

```
55 $(foreach bdir,$(dirs-y),$(eval $(call make-subdir,$(bdir))))
```

它的作用是，对于\$(dirs-y)中的每一个项目，调用在 48~53 行定义的 make-subdir 函数进行处理，\$(dirs-y)中的每一个项目将作为参数传递给 make-subdir 函数，也就是该函数的\$1。现在我们试图把 lib 作为参数带入该函数并展开，我们会得到以下函数：

```
49 lib: FORCE
50     @echo Enter $(LOCALPATH)/lib
51     -$(Q)mkdir -p $(OBJPATH)/lib
52     +$(Q)$(MAKE) -f $(KTREE)/Makefile.subdir -C $(LOCALPATH)/lib KTREE=$(KTREE)
OBJPATH=$(OBJPATH)/lib LOCALPATH=$(LOCALPATH)/lib BUILTLIST=$(BUILTLIST)
```

这样，我们就得到了 lib 伪目标所对应的构建动作了！同理，我们会得到 fs 参数带入后所定义的伪目标构建动作。我们先来分析以上 lib 所对应的构建动作：我们看到有两个动作，其一是建立\$(OBJPATH)/lib 目录，也就是\$(TOPDIR)/obj/kernel/lib 目录；另一个动作是进入\$(LOCALPATH)/lib 目录（make 命令的 -C 参数，\$(LOCALPATH)/lib 展开后得到./kern-ucore/lib），并仍然执行 Makefile.subdir 构建脚本。也就是说，流程不变，但构建的目录换成了./kern-ucore/lib，且传入的参数 LOCALPATH 也换成了./kern-ucore/lib。这就意味着，Makefile.subdir 的构建过程是嵌套进行的！

进入./kern-ucore/lib，我们发现该目录下的 Makefile 文件非常简单：

```
1 obj-y := $(patsubst %.c,%.o,$(wildcard *.c))
```

它只规定了 obj-y，它等于./kern-ucore/lib 下所有.c文件的扩展名换成.o之后的字符串。而未规定 dirs-y，也就是 dirs-y 的值为空。这样，在./kern-ucore/lib 下进行的构建动作只会生

成最终的 builtin.o 文件，也就是 ./obj/kernel/lib/builtin.o，而不会进行进一步的嵌套。

但对于 ./kern-ucore/fs 目录，则不同，我们查看该目录下的 Makefile 文件：

```
1 dirs-y := devs pipe vfs
2 dirs-$(UCONFIG_HAVE_SFS) += sfs
3
4 obj-y := file.o fs.o iobuf.o sysfile.o
```

对 dirs-y 的定义，意味着在 ./kern-ucore/fs 目录下进行的构建过程将嵌套调用 4 次 Makefile.subdir 构建脚本，它们分别对应 ./kern-ucore/fs/devs、./kern-ucore/fs/pipe、./kern-ucore/fs/vfs 和 ./kern-ucore/fs/sfs。

对 Makefile.subdir 脚本所对应的构建过程进行总结，我们发现，它对应的动作发生在以下 7 个目录中：

- 1) ./kern-ucore
- 2) ./kern-ucore/lib
- 3) ./kern-ucore/fs
- 4) ./kern-ucore/fs/devs
- 5) ./kern-ucore/fs/pipe
- 6) ./kern-ucore/fs/vfs
- 7) ./kern-ucore/fs/sfs

构建过程分别进入这些目录，并依次编译它们中的 .c 文件，生成以下 builtin.o 文件：

```
./obj/kernel/builtin.o
./obj/kernel/lib/builtin.o
./obj/kernel/fs/builtin.o
./obj/kernel/fs/devs/builtin.o
./obj/kernel/fs/pipe/builtin.o
./obj/kernel/fs/sfs/builtin.o
./obj/kernel/fs/vfs/builtin.o
```

最终，回到 ./kern-ucore 目录下的 Makefile.build 文件（Makefile.subdir 的上一级），执行其中的第 43 行，生成最终的 \$(KTREE_OBJ_ROOT)/kernel-builtin.o，也就是 ./obj/kernel/kernel-builtin.o 文件。

但由于 Makefile.build 文件中的 \$(KERNEL_BUILTIN_O) 伪目标只是最终目标 all 的一个依赖目标，所以将继续 Makefile.build 文件中第 39 行所规定的构建动作：

```
35 all: $(KTREE_OBJ_ROOT) $(KERNEL_BUILTIN_O)
36 ifneq ($(UCORE_TEST),)
37     $(Q)touch $(KTREE)/process/proc.c
38 endif
39     $(Q)$(MAKE) KERNEL_BUILTIN=$(KERNEL_BUILTIN_O) -C $(KTREE) -f
$(KTREE)/Makefile.image all
```

也就是进入 ./kern-ucore 目录，并执行 Makefile.image 脚本。这样，我们就进入了 Hos 内核构建的第三个阶段。

3. image 的生成

我们先来查看 Makefile.image 脚本的内容：

```
1 ifneq ($(MAKECMDGOALS),clean)
2 include $(KCONFIG_AUTOCONFIG)
3 endif
4
5 ARCH_DIR := $(KTREE)
6
```

```

7 KERNEL_ELF := $(KTREE_OBJ_ROOT)/ucore-kernel-initrd
8 LINK_FILE   := $(KTREE)/ucore.ld
9
10 ROOTFS_IMG   := $(OBJPATH_ROOT)/sfs-orig.img
11
12 SRC_DIR      := $(ARCH_DIR)/include
13 ASMSRC       := $(wildcard $(KTREE)/*.S)
14 MIPS_S_OBJ   := $(patsubst $(ARCH_DIR)/%.S, $(KTREE_OBJ_ROOT)/%.o, $(ASMSRC))
15 INCLUDES     := $(addprefix -I,$(SRC_DIR))
16
17 MK_DIR:
18     mkdir -p $(KTREE_OBJ_ROOT)
19     mkdir -p $(KTREE_OBJ_ROOT)/init
20     mkdir -p $(KTREE_OBJ_ROOT)/trap
21     mkdir -p $(KTREE_OBJ_ROOT)/process
22     mkdir -p $(KTREE_OBJ_ROOT)/module
23
24 ifeq ($(ON_FPGA), y)
25 MACH_DEF := -DMACH_FPGA
26 else
27 MACH_DEF := -DMACH_QEMU
28 endif
29
30 $(MIPS_S_OBJ): $(KTREE_OBJ_ROOT)/%.o: $(ARCH_DIR)/%.S
31     $(TARGET_CC) -g -ggdb -c -D_ASSEMBLY_ $(MACH_DEF) -EL -mno-mips16
-msoft-float -march=m14k -G0 -Wformat -O0 -msoft-float $(INCLUDES) $< -o $@
32
33 $(KERNEL_ELF): $(LINK_FILE) $(KERNEL_BUILTIN) $(RAMDISK_OBJ) $(MIPS_S_OBJ)
34     @echo Linking uCore
35     sed 's%_FILE_%$(ROOTFS_IMG)%g' tools/initrd_piggy.S.in >
$(KTREE_OBJ_ROOT)/initrd_piggy.S
36     $(CROSS_COMPILE)as -g --gen-debug -EL -mno-micromips -msoft-float -march=m14k
$(KTREE_OBJ_ROOT)/initrd_piggy.S -o $(KTREE_OBJ_ROOT)/initrd.img.o
37     $(Q)$(TARGET_LD) $(TARGET_LDFLAGS) -T $(LINK_FILE) $(KERNEL_BUILTIN)
$(RAMDISK_OBJ) $(MIPS_S_OBJ) $(KTREE_OBJ_ROOT)/initrd.img .o -o $@
38     $(CROSS_COMPILE)objdump -d -S -l $@ 1>$(KTREE_OBJ_ROOT)/u_dasm.txt
39
40 $(BOOTSECT): $(OBJPATH_ROOT)
41     $(Q)$(MAKE) -C $(BLTREE) -f $(BLTREE)/Makefile all
42
43 .PHONY: all clean FORCE
44 all: $(KERNEL_ELF)
45
46 FORCE:
47
48 clean:
49     rm -f $(KERNEL_ELF)

```

我们先理清这个构建脚本的最终伪目标 all 的依赖路径，用箭头来表示依赖关系，如有 A→B,C 就标识 A 依赖于 B 和 C：

all → \$(KERNEL_ELF) → \$(LINK_FILE)、\$(KERNEL_BUILTIN)、\$(RAMDISK_OBJ)、\$(MIPS_S_OBJ)

逐次查看被依赖的伪目标，我们发现：

1) \$(LINK_FILE) = \$(KTREE)/ucore.ld = ./kern-ucore/ucore.ld

2) \$(KERNEL_BUILTIN) 是在 Makefile.image 脚本被调用时，从 Makefile.build 中传递过来的参数，它等于 \$(KTREE_OBJ_ROOT)/kernel-builtin.o，也就

是./obj/kernel/kernel-builtin.o。

3) \$(RAMDISK_OBJ)并无实际定义

4) \$(MIPS_S_OBJ)实际上是先查找./kern-ucore 下所有的.S 汇编文件，将扩展名变为.o 并将其路径替换为./obj/kernel 后的结果。因为./kern-ucore 下有 3 个.S 文件，它们分别是 entry.S、exception.S 和 switch.S，替换后，\$(MIPS_S_OBJ)的值为：

\$(MIPS_S_OBJ) = ./obj/kernel/switch.o ./obj/kernel/exception.o ./obj/kernel/entry.o

这样，Makefile.image 最终的构建伪目标所依赖的伪目标就只有对\$(MIPS_S_OBJ)中所规定的.o 文件的编译（Makefile.image 的第 31 行）。编译完成后，将进行第 35~38 行的动作。第 35 行的动作是根据./kern-ucore/tools/initrd_piggy.S.in 的内容，替换（sed 命令）其中部分内容，将结果输出到\$(KTREE_OBJ_ROOT)/initrd_piggy.S，也就是./obj/kernel/initrd_piggy.S 中。./kern-ucore/tools/initrd_piggy.S.in 中的内容为：

```
1 .section .data
2 .align 4 # which either means 4 or 2**4 depending on arch!
3
4 .global _initrd_begin
5 .type _initrd_begin, @object
6 _initrd_begin:
7 .incbin "_FILE_"
8
9 .align 4
10 .global _initrd_end
11 _initrd_end:
```

结果替换后，输出文件./obj/kernel/initrd_piggy.S 的内容为：

```
1 .section .data
2 .align 4 # which either means 4 or 2**4 depending on arch!
3
4 .global _initrd_begin
5 .type _initrd_begin, @object
6 _initrd_begin:
7 .incbin "D:/Hos/hos-mips/obj/sfs-orig.img"
8
9 .align 4
10 .global _initrd_end
11 _initrd_end:
```

对比两个文件，我们发现 initrd_piggy.S.in 文件中的 _FILE_ 字符串被替换为 D:/Hos/hos-mips/obj/sfs-orig.img，也就是 Hos 构建过程中的第一步所生成的虚拟“磁盘镜像”文件！initrd_piggy.S 文件的作用，是定义最终生成的 ELF 文件中的数据段（.data），把第一步生成的“磁盘镜像”文件整体放到数据段的 _initrd_begin 和 _initrd_end 两个符号之间。

接下来，我们回到 Makefile.image 脚本的 36 行，该行的动作是调用交叉编译器的汇编命令，生成\$(KTREE_OBJ_ROOT)/initrd.img.o，也就是./obj/kernel/initrd.img.o。该文件是按照 initrd_piggy.S 文件所生成的 ELF 文件，只包含数据段，且将“磁盘镜像”文件“嵌”到该数据段中。我们再来到 Makefile.image 脚本的 37 行，将该行的宏定义替换后得到以下命令行：

```
"mips-sde-elf-"ld -n -G 0 -static -EL -nostdlib
-T ./kern-ucore/ucore.ld ./obj/kernel/kernel-builtin.o ./obj/kernel/switch.o ./obj/kernel/exception.o ./obj/kernel/entry.o ./obj/kernel/initrd.img.o -o ./obj/kernel/ucore-kernel-initrd
```

该命令实际上是通过交叉编译器所提供的链接程序（ld）根据./kern-ucore/ucore.ld 模版，生成 ELF 文件。构造该 ELF 文件的输入有：构建第二步生成的内核镜像

(`./obj/kernel/kernel-builtin.o`) ; 根据 `.S` 编译生成的 `.o` 文件 (`./obj/kernel/switch.o`、`./obj/kernel/exception.o` 和 `./obj/kernel/entry.o`)；以及刚刚生成的包含第一步构建过程所得到的“磁盘镜像”的 ELF 文件 (`./obj/kernel/initrd.img.o`)。其中模版文件

(`./kern-ucore/ucore.ld`) 中重要的部分有：

```

1 OUTPUT_FORMAT(elf32-tradlittlemips)
2 OUTPUT_ARCH(mips:isa32)
3 ENTRY(kernel_entry)
4
5 SECTIONS
6 {
7     . = 0x80001000;
8     .text :
9     {
10        . = ALIGN(4);
11        wrs_kernel_text_start = .; _wrs_kernel_text_start = .;
12        *(startup)
13        *(.text)
14        *(.text.*)
15        *(.gnu.linkonce.t*)
16        *(.mips16.fn.*)
17        *(.mips16.call.*) /* for MIPS */
18        *(.rodata) *(.rodata.*) *(.gnu.linkonce.r*) *(.rodata1)
19        . = ALIGN(4096);
20        *(.ramexv)
21    }
22    . = ALIGN(16);
23    wrs_kernel_text_end = .; _wrs_kernel_text_end = .;
24    etext = .; _etext = .;
25    .data ALIGN(4) : AT(etext)
26    {
27        wrs_kernel_data_start = .; _wrs_kernel_data_start = .;
28        *(.data)
29        *(.data.*)
30        *(.gnu.linkonce.d*)
31        *(.data1)
32        *(.eh_frame)
33        *(.gcc_except_table)
34        . = ALIGN(8);
35        _gp = . + 0x7fff0; /* set gp for MIPS startup code */
36        /* got*, dynamic, sdata*, lit[48], and sbss should follow _gp */
37        *(.got.plt)
38        *(.got)
39        *(.dynamic)
40        *(.got2)
41        *(.sdata) *(.sdata.*) *(.lit8) *(.lit4)
42        . = ALIGN(16);
43    }
44    .....

```

可以看到,该 ELF 文件的目标平台是 MIPS,其入口是 `kernel_entry`(见 `kern-ucore/init.c`),起始虚地址为 `0x80001000`。ELF 的第一个段是代码段 (`.text`),接下来才是数据段 (`.data`)。

经过链接,构建过程中生成的所有“成果”都融到了 `./obj/kernel` 目录下,文件名为 `ucore-kernel-initrd` 的文件中,在本书中,我们称该文件为 Hos 的操作系统镜像文件。

最后, `Makefile.image` 脚本在第 38 行,将通过交叉编译器的 `objdump` 将 ELF 中的符号

反编译输出到./obj/kernel/u_dasm.txt 中。

6.3.2 Hos 的载入和调试

在我们的实验中，镜像文件（ucore-kernel-initrd）的是通过交叉编译器所提供的 gdb 工具，装载到 MIPSfpga 开发板上的，其装载过程分为代码段、数据段的装载。由于这个原因，Hos 与传统的操作系统不同，它没有 bootloader（也就是加载操作系统到内存的）部分。之所以可以通过 gdb 加载整个镜像文件，是因为 J-TAG 的作用类似于 gdb-server。需要说明的是，J-TAG 的 gdb-server 对应的代码自动地加载到虚地址 0x80000000 — 0x80001000 之间，这也是为什么 Hos 的起始虚地址是 0x80001000，而不是 0x80000000（ucore 以及 ucore-plus）的原因。

其调试过程是通过在主机上的 gdb、VSCode，以及 J-TAG 的配合，在给定的虚地址（VSCode 中设置的断点）暂停操作系统的执行而实现的。

为了更好地对加载和调试过程进行理解，我们建议读者先对 ELF 文件的格式，和 gdb 工具的使用进行了解，清华大学出版社出版的陈渝和向勇老师编著的《操作系统实验指导》对这两个知识点进行了较好的讲解，在此不再赘述。

第七章 实验 3. 从应用到内核

7.1 实验目的

回顾操作系统特权级知识，掌握 Hos 中从应用层到系统内核层的完整调用路径，在 Hos 内核中添加系统调用，使得应用层能够触发操作系统特权级的动作。

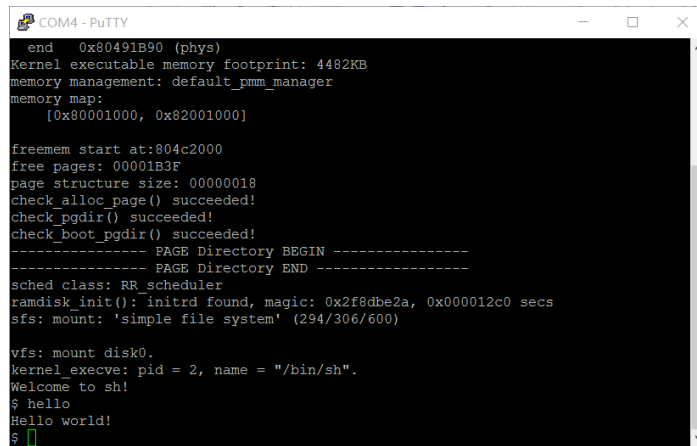
7.2 实验内容

7.2.1 在 Hos 操作系统中添加应用，打印“hello world!”。

与标准的 Linux 不同，Hos 并不提供丰富的编辑、编译工具（如 vim、gcc 等）。为了在 Hos 中添加新的应用，就必须搞清 Hos 系统的镜像生成过程。读者可以在本书的 6.3.1 章节中，找到介绍 Hos 系统镜像的生成过程的知识。

应用的开发应该在读者的开发主机上用 VSCode 来完成，用交叉编译器进行编译，并需要将编译生成的二进制代码（ELF 文件）“加入”到 Hos 的系统镜像中。一个比较简单的做法是将 hello world 应用视作 Hos 中的一个命令（如 ls，cd 等），因为它与其他命令一样，处于操作系统的应用层。当然，读者也能够找到其他的办法来将自己编写的应用加入到系统镜像中，作为 sfs 文件系统中的一个普通文件存在。

在完成应用的编写、编译链接，并加入到系统镜像后，依照本书前两章介绍的方法，重新启动系统。在 Putty 终端中，切换到新编写的应用所在的目录，并执行自己编写的应用，读者将看到如下界面：



```
end 0x80491B90 (phys)
Kernel executable memory footprint: 4482KB
memory management: default_pmm_manager
memory map:
[0x80001000, 0x82001000]

freemem start at:804c2000
free pages: 00001B3F
page structure size: 00000018
check alloc_page() succeeded!
check pgdir() succeeded!
check boot_pgdir() succeeded!
----- PAGE Directory BEGIN -----
----- PAGE Directory END -----
sched class: RR_scheduler
ramdisk init(): initrd found, magic: 0x2f8dbe2a, 0x000012c0 secs
sfs: mount: 'simple file system' (294/306/600)

vfs: mount disk0.
kernel_execve: pid = 2, name = "/bin/sh".
Welcome to sh!
$ hello
Hello world!
$
```

图 7.2.1 hello world!应用的输出

这里,需要提示的是,Hos 的用户态 lib 只提供了 fprintf 编程接口,没有我们熟悉的 printf。为了实现 hello world!字符串的打印,读者可以在自己的程序中定义如下的宏:

```
#define printf(...) fprintf(1, __VA_ARGS__)
```

其中, fprintf 的第一个参数 (1) 表示将输出定向到“标准输出”文件,也就是 Hos 的终端了。这样,读者就能够像在其他标准 Linux 中那样,“愉快”地使用 printf 了。

另外,重新编译、构建 Hos 操作系统镜像,有两个方法:

1. 切换到 cygwin, 使用 make 命令行构造。
2. 在 vscode 中通过快捷键 Ctrl + shift + b 来构建。

7.2.2 在 Hos 操作系统中添加系统调用, 实现应用层对新添系统调用的触发

出于安全性方面的考虑,今天的操作系统在 CPU 硬件(我们在第一部分设计的 MIPSfpga 就提供这样的支持)的配合下,将软件的执行环境分为两个特权态:用户态和核心态。对应的,CPU 硬件在这两个状态下分别处于最低特权和最高特权两个状态。

一般来说,在用户态运行的程序,对于计算机资源利用的特权级也最低。只能执行一些非特权指令,如算数逻辑指令和对指定地址区间的访存指令。而在核心态运行的程序,则能够使用计算机系统的所有资源。在 Hos 操作系统中,应用(如我们在上一个实验中写的打印 hello world!的程序)就是在用户态下运行,而操作系统内核则在核心态下运行。

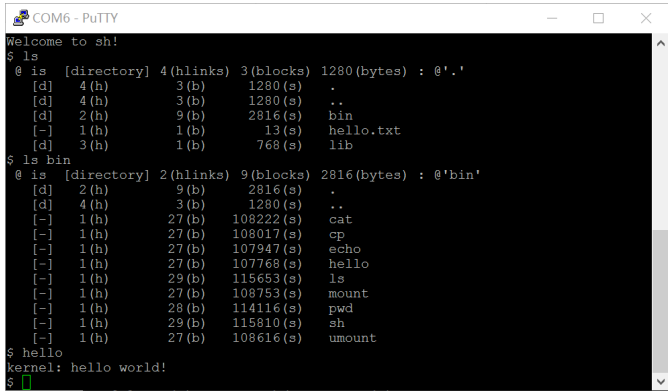
这样的设计带来一个问题:如果应用态的程序希望执行特权态的动作(例如控制 I/O 设备,我们将在本书的第三部分中碰到这方面的问题),应该如何实现呢?一个最简单和直管的方法是通过系统调用来实现。当然,万事没有绝对,我们在本书的第三部分将为了提高系统的性能而设计其他的方法。在本部分,我们将完成从应用到系统调用的全路径实现。

本实验的第一步,要求读者在 Hos 操作系统内核中新添一个系统调用。该系统调用的动作非常简单:在内核态打印“kernel: hello world!”字符串。提示:这个过程,涉及到的文件有 \kern-ucore\syscall.c 文件。

接下来,就要考虑如何在应用层启动程序,来触发我们新添的系统调用了。提示:这个过程涉及到将新添的系统调用加入到整个 Hos 体系中,读者可以参考其他系统调用的实现方法。具体来说,需要修改以下文件: kern-ucore\include\lib\unistd.h、user\include\unistd.h,以及在 user\include\syscall.h 声明一个可以由应用层代码调用的接口。

最后,读者可以在 7.2.1 节所述的实验代码的基础上,调用 user\include\syscall.h 中声明

的接口，从而完成对新添系统调用的触发。实验完成后，Hos 的执行效果如图 7.2.2 所示。



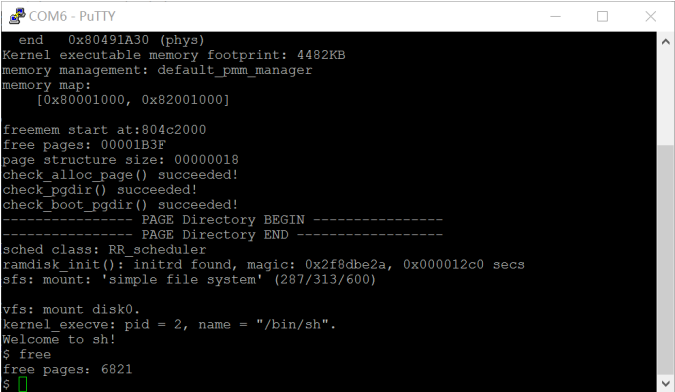
```
COM6 - PuTTY
Welcome to sh!
$ ls
@ is [directory] 4(hlinks) 3(blocks) 1280(bytes) : @'.'
[d] 4(h) 3(b) 1280(s) .
[d] 4(h) 3(b) 1280(s) ..
[d] 2(h) 9(b) 2816(s) bin
[-] 1(h) 1(b) 13(s) hello.txt
[d] 3(h) 1(b) 768(s) lib
$ ls bin
@ is [directory] 2(hlinks) 9(blocks) 2816(bytes) : @'bin'
[d] 2(h) 9(b) 2816(s) .
[d] 4(h) 3(b) 1280(s) ..
[-] 1(h) 27(b) 108222(s) cat
[-] 1(h) 27(b) 108017(s) cp
[-] 1(h) 27(b) 107947(s) echo
[-] 1(h) 27(b) 107768(s) hello
[-] 1(h) 29(b) 115653(s) ls
[-] 1(h) 27(b) 108753(s) mount
[-] 1(h) 28(b) 114116(s) pwd
[-] 1(h) 29(b) 115810(s) sh
[-] 1(h) 27(b) 108616(s) umount
$ hello
kernel: hello world!
$
```

图 7.2.2 hello world!在操作系统特权态的输出

7.2.3 打印 Hos 操作系统的空闲页面数量

通过 7.2.2 节中的实验，我们掌握了在应用层编写程序实现特权动作的方法。在本节，我们将再次应用这个方法，在应用层开发程序来实现特权动作：统计系统中剩余的空闲页面的数量。这个类似于 Linux 下的 free 命令，总是能显示出系统的剩余内存量。

该实验可以借鉴本章前两个实验的内容，对内存剩余物理页面数量的获取，可以参考 ./kern-ucore/pmm.c 中 Hos 对物理内存管理的机制。实验最后生成的系统，其运行结果如图 7.2.3 所示：



```
COM6 - PuTTY
end 0x80491a30 (phys)
Kernel executable memory footprint: 4482KB
memory management: default_pmm_manager
memory map:
[0x80001000, 0x82001000]

freemem start at:804c2000
free pages: 00001B3F
page structure size: 00000018
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
----- PAGE Directory BEGIN -----
----- PAGE Directory END -----
sched class: RR_scheduler
ramdisk_init(): initrd found, magic: 0x2f8dbe2a, 0x000012c0 secs
sfs: mount: 'simple file system' (287/313/600)

vfs: mount disk0.
kernel_execve: pid = 2, name = "/bin/sh".
Welcome to sh!
$ free
free pages: 6821
$
```

图 7.2.3 显示 Hos 剩余内存容量

7.3 实验背景及原理

7.3.1 Hos 操作系统的特权态

由于本书所指导的课程是面向高年级本科生的，相信本书的读者已经完成了《操作系统原理》的学习。在《操作系统原理》课程中，读者已经接触过操作系统的特权态概念。然而，在本书中，我们要强调的是：操作系统特权态的实现是有“物质基础”的。操作系统提供特权态支持的前提，是运行该操作系统的物理处理器必须支持特权级的区分。

在 MIPS 处理器，也就是 Hos 所面对的处理器，是通过 CP0 协处理器。实际上 CP0 是 MIPS 系统中一个非常重要的寄存器，起到控制 CPU 的作用。MMU、异常处理、乘除法等功能，都依赖于协处理器 CP0 来实现，它也是打开 MIPS 特权级模式的大门。

为了实现未开放的特权动作（Hos 中没有 Linux 中的 /proc 文件系统），所以只能通过应

用程序——>用户态函数库——>内核这个流程来进行开发。

7.3.2 MIPS 的内存映射

在 32 位 MIPS 体系结构下，最多可寻址 4GB 地址空间，对这 4GB 空间的分配见下图：

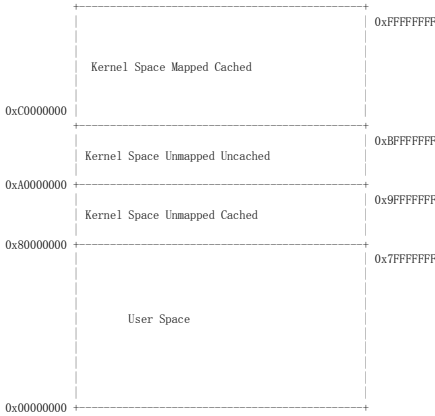


图 7.3.1 MIPS 系统的虚拟地址分配

上图是 MIPS 处理器的逻辑寻址空间分布图。我们看到：

1. 从 0x00000000 到 0x7FFFFFFF 。2GB 以下的地址空间为 User Space，可以在用户态下访问。当然，在内核态下也是可以访问的。程序在访问 User Space 的内存时，会通过 MMU 的 TLB，映射到实际的物理地址上。也就是说，这一段逻辑地址空间和物理地址空间的对应关系，是由 MMU 中的 TLB 表项决定的。

2. 从 0x80000000 到 0xFFFFFFFF。这一段为 Kernel Space，仅限于内核态访问。如果在用户态下试图访问这一段内存，将会引发系统的一个 Exception。MIPS 的 Kernel Space 又可以划分为三部分：

① 从 0xC0000000 到 0xFFFFFFFF。是通过 MMU 映射（通过 TLB 进行地址转换）到物理地址的 1GB 空间地址范围。这 1GB 空间可以用来访问实际的 DRAM 内存，可以为操作系统的内核所用。

② 从 0x80000000 到 0x9FFFFFFF。这一段的特点是 Kernel Space Unmapped Cached，也就是说，对它的访问可以借助 Cache 的帮助，而得到性能上的提高。一般地，这段内存空间用于内核代码段，或者内核中的堆栈。

③ 从 0xA0000000 到 0xBFFFFFFF。这一段的特点是 Kernel Space Unmapped Uncached，也就是说，对它的访问是直接定向到存储器的，无法借助 Cache 的帮助。这样做的好处在于无需考虑 Cache 一致性的问题，所以更加符合对硬件 I/O 寄存器的操纵。例如，MIPS 的程序上电启动地址 0xBFC00000，也落在这段地址空间内。——上电时，MMU 和 Cache 均未初始化，因此，只有这段地址空间可以正常读取并处理。

需要指出的是，对②和③中的两段逻辑地址（从 0x80000000 到 0xBFFFFFFF）的访问，都是直接映射到物理内存，且这个过程无需进行 MMU 映射，也就是说无须通过 TLB。

7.3.3 Hos 的虚地址规划

Hos 中的虚地址规划是：内核虚地址起点为 0x80000000，而应用程序的虚地址起点为 0x10000000。对应图 7.3.1 中所示的 MIPS 系统的虚拟地址分配，Hos 为各个区段的逻辑地址定义了它们各自的区段，以方便管理：

1> [0x00000000, 0x7ffffff] (0~2G-1) KUSEG

这些地址是用户态可用的地址。这些地址通常使用 MMU 进行地址转换。换句话说，除非 MMU 的机制被建立好，这 2G 地址是不可以进行使用的。在本次系统中，用户程序的起始地址皆为 0x10000000。

2> [0x80000000, 0x9fffffff] (2G~2.5G-1) KSEG0

这些地址简单的通过映射即可找到对应的物理地址。方式为把最高位清零，然后把它们映射到物理地址低段 512M[0x00000000, 0x1FFFFFFF]。因为这种映射是很简单的。但是几乎全部的对这段地址的存取都会通过快速缓存 (cache)。因此在 cache 设置好之前，不能随便使用这段地址。通常一个没有 MMU 的系统会使用这段地址作为其绝大多数程序和数据的存放位置。对于有 MMU 的系统，操作系统核心会存放在这个区域。

3> [0xa0000000, 0xbfffffff] (2.5G~3G-1) KSEG1

这些地址通过把最高 3 位清零的方法来映射到相应的物理地址上，与 kseg0 映射的物理地址一样。但 kseg1 是非 cache 存取的。kseg1 是唯一的在系统重启时能正常工作的地址空间。这也是为什么重新启动时的入口向量是 0xbfc00000。这个向量相应的物理地址是 0x1fc00000。将使用这段地址空间去存取你的初始化 ROM。大多数人在这段空间使用 I/O 寄存器。

4> [0xc0000000, 0xffffffff] (3G~4G-1) KSEG2

这段地址空间只能在核心态下使用并且要经过 MMU 的转换。在 MMU 设置好之前，不能存取这段区域。除非你在写一个真正的操作系统，一般来说你不需要使用这段地址空间。

通过各段部分的介绍，我们知道 KSEG0 和 KSEG1 是可以不依赖 MMU，直接进行地址转换访问物理地址的。这也解释了为什么我们的 kernel 是从 0x80000000 开始的（即位于 KSEG0 这一区间），且大部分外设的统一编址都位于 KSEG1 这一区间，如串口、键盘、VGA 等等。而对于 KUSEG 这一段地址，都是留给用户的应用程序使用的空间。

在了解了前面的四段地址的基础上，我们再对 MIPS 的内存转换机制进行详细说明，过程如下图所示：

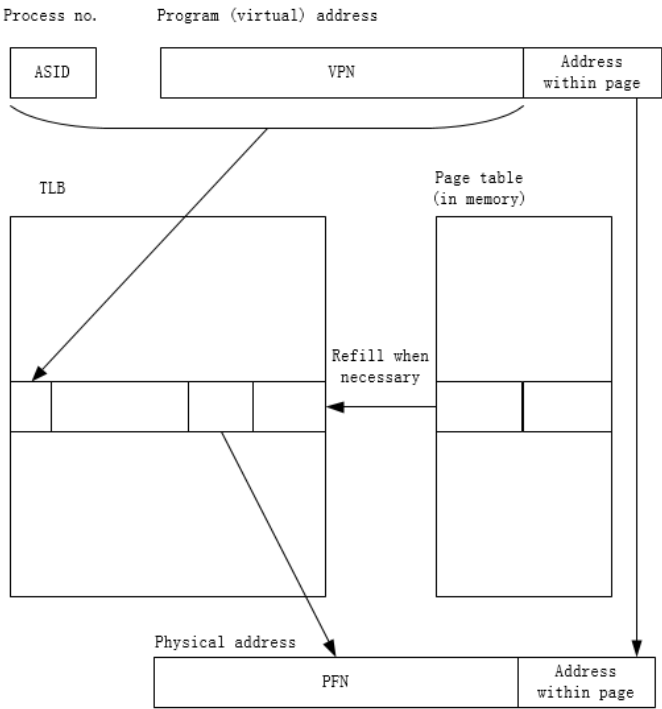


图 7.3.2 硬件-虚实地址转换机制

硬件的处理过程大致上是这样的：

1. 一个虚地址被分割为两部分，低半部分地址位（通常为 12 位）不经转译直接通过，因此转译结果总是落在一个页内（通常 4KB 大小）。因此这部分被称作页内偏移。

2. 高半部分地址位，也就是 VPN（Virtual Page Number, 虚拟页号），VPN 由页目录和页表号组成，然后会在前面拼接上当前运行进程的 ASID（Address Space ID, 地址空间标示符）以形成一个独一无二的页首地址。因此，我们也不必担心两个不同的进程的同一个虚拟地址会访问到同一个物理地址。

3. 我们在 TLB（Translation Lookaside Buffer, 快表）中查找是否有一个本页的转译项在里面。如果存在，那么我们将得到对应的物理地址的高位，最终得到可用地址。TLB 是一个做特殊用途的存储器件，可以运用各种有效的方法来匹配地址。

这里的重点则是 TLB。TLB 是将程序的虚拟地址转换成访问存储器中的物理地址的硬件。我们早就知道，MIPS 的 CPU 的地址转换单位为页，页内偏移是可以直接从虚拟地址传递到物理地址，而虚拟页号和物理页号则需要通过查找页表来实现，在 MIPS 中，页表会被缓存到 TLB 当中，即 CPU 是通过 TLB 将虚拟页号翻译成物理页号的。

TLB 中的每一项含有一个页的虚拟页号 VPN 和一个物理页号 PFN（Page Frame Number, 页帧号）。当程序给出一个虚拟地址，该地址的 VPN 会一一和 TLB 中的每一个 TLB 进行比较，如果匹配成功就给出对应的 PFN，并返回一组标志位让操作系统来确定某一页为只读或者是否进行高速缓存。

其结构如下图所示：

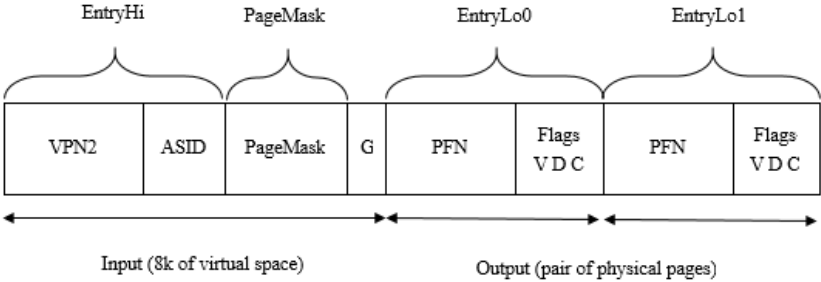


图 7.3.3 TLB 中的一项

上图为 TLB 中的一个数据项，其可以容纳一对相邻的虚拟页面和对应的两个单独的物理地址。同时，我们在图中标出了加载和读取 TLB 表项时使用的 CP0 寄存器的名称。

为了顺利完成页式地址转换的过程，CP0 中有几个寄存器会辅助 TLB 进行地址转换，下面列表说明：

表 7.3.1 用于虚实地址转换的 CP0 寄存器

寄存器助记符	CP0 寄存器号	功能
EntryHi	10	具备 VPN 和 ASID
EntryLo0	2	VPN 所映射到的物理页号, 以及对应物理页的存取权限
EntryLo1	3	
PageMask	5	用来创建能映射超过 4KB 的页的入口
Index	0	决定相应指令要读写的 TLB 表项
Random	1	这个伪随机值（实际上是一个自由计数的计数器）用来让 tlbnwr 写入新的 TLB 入口到一个随机选择的位置。为那些使用随机替换的软件在陷入 TLB 重装入异常时的处理节省了时间
Context	4	这些是很有用的寄存器，用来加速 TLB 重装入的过程。它们的高位可读写，低位从不可转译的 VPN 中得来。寄存器的域这样布置使得如果您使用了合适的内存

XContext	20	转译纪录的内存拷贝的安排方式，那么紧跟在 TLB 重装陷入后 Context 会装有一个指向用来映射触发异常地址的页表纪录的指针。XContext 在处理超过 32 位有效地址时做了相同的工作
----------	----	--

这里我们就 MIPS32 架构详细介绍一下几个常用寄存器，并就 QEMU 模拟器和 NEXYS4 开发板之间的不同之处进行分析。

1. EntryHi

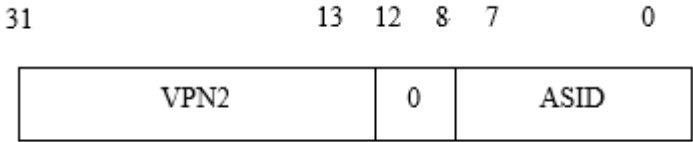


图 7.3.4 EntryHi 寄存器结构

VPN2 是除去页内偏移部分的虚拟地址，一共有 20 位。ASID 是被用来保存操作系统当前的地址空间的标识。但是系统发生异常时，该数据不会发生改变。因为处理完异常后，执行的进程仍然可以对应该 ASID。总的来说，这个寄存器就可以定位到某个进程的虚拟页号了。

2. PageMask

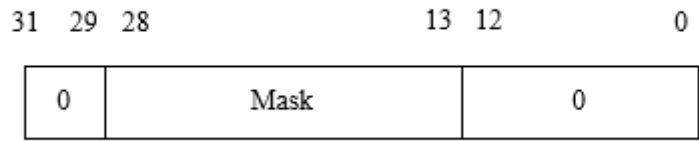


图 7.3.5 PageMask 寄存器结构

PageMask 是用来设置 TLB，让它可以映射更大的页。这个寄存器在 QEMU 模拟器中存在，但是在 NEXYS4 开发板中没有进行预留。但是这个寄存器的功能主要是设置映射的页面大小，而我们常用的页面大小是 4K，因此该寄存器的存在与否并不影响我们的操作系统的运行。

3. EntryLo

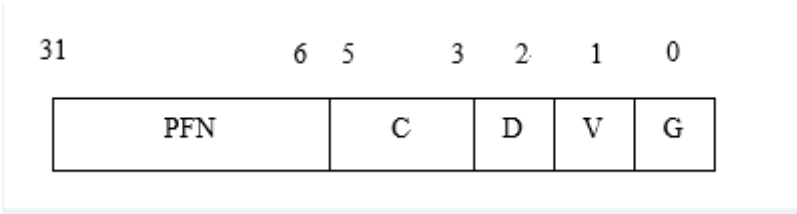


图 7.3.6 EntryLo 寄存器结构

PFN 是和 EntryHi 对应的地址转换项的物理地址的高位部分。C 是可以用来设置多处理器情况下的 cache 一致性算法，D 则代表常见脏位，V 代表该项是否有效，G 代表 global，提供了一种所有进程共享的地址空间。

总结一下，系统将用户进程的 ASID 和虚拟地址的 VPN 传给 CP0 的 EntryHi，然后硬件就会自动从 TLB 中查找到对应的 EntryLo，从而得到物理地址的 PFN 和相应物理地址的读写权限和高速缓存设置等信息，从而用户可以获取进程中虚拟地址指向的指令或数据。

这里需要提到在 `tlb_refill` 的处理中, `tlb_refill` 这个函数的功能就是将存在于内存中的数据填充到 TLB 中, 从而 CPU 可能正常在 TLB 中使用虚拟地址匹配到对应的物理地址。但是, 事实上 QEMU 模拟器和 NEXYS4 真实硬件上存在不同之处。我们知道在标准 CP0 当中存在一个 Random 寄存器。它的功能是生成一个这个伪随机值, 然后执行一个 `tlbwr` 指令, 该指令的功能是选择一个新的 TLB 入口到一个随机选择的位置, 然后把数据写入到那个位置。这些机能为那些使用随机替换的软件在陷入 TLB 缺失异常后, 重装入 TLB 的处理节省了时间, 从而整体提升了效率。但是, 我们之前说过, 为了提高 CPU 的频率, 我们的 NEXYS4 硬件中不再包含 Random 寄存器和 `tlbwr` 这条指令。因此, 我们不能使用这种随机写入页表项的方式来刷新我们的 TLB 硬件。但是作为代替, 我们可以通过 Index 寄存器和 `tlbwi` 对 TLB 进行 FIFO 刷新方式, 来代替之前的刷新方式。

7.3.4 缺页异常与处理

为什么会产生缺页异常

我们知道操作系统和用户程序都存放在磁盘之中, 而不是内存, 其原因有很多, 如:

1. 使用易失性存储介质作为内存, 断电时导致存放的用户数据丢失
2. 本应存放在磁盘上的用户程序、数据内容远远超过内存的大小。

但是, 为了保证程序的高效运行, 又不能让 CPU 直接去磁盘寻找数据。因此, 我们需要一种方案来主动将数据从磁盘载入内存之中, 这就是缺页异常的来源。

在 MIPS 中 CPU 访问内存的一般过程为, CP0 中的一些寄存器会先帮助 CPU 访问 TLB (快表) 中的指定项内容, 然后再根据对应的 TLB 表项找到物理地址, 然后再通过物理地址进行访问。如果 TLB 中没有数据, 则会发生 TLB 缺失异常, 然后我们根据之前中断异常处理的设计流程知道, 此时会有对应的 TLB 缺失的 `handler` 函数处理 TLB 缺失异常, 然后处理完异常后的程序即可满足 CPU 在 TLB 表项中访问到物理地址。此时我们处理 TLB 缺失异常的过程有两种情况:

1. 需要访问的地址存在于在内存中, 但是其对应的内容不在 TLB 的每一项中。
2. 需要访问的地址根本就不在内存中, 此时该数据一定不会出现在 TLB 的任意一项中。

对于第一种情况, 我们只需要根据出现异常出现的地址, 系统会通过 `tlb_refill` 函数缺失的页面地址信息重新载入到 TLB 当中。

而第二种则是先将数据从磁盘载入到内存中, 然后再使用 `tlb_refill` 进行数据。在这种情况下, 系统首先需要获取出现异常出现的地址以及其相关权限信息, 如果因为权限信息表明该改地址属于异常访问的地址时, 系统将会返回一个错误码。如果权限信息正常, 则系统会将磁盘中的数据载入到新分配的页面当中, 并将异常地址信息载入到 TLB 中。

中断和异常处理

中断和异常是操作系统中极为重要的一部分。一个系统想要与外部进行有效通讯, 通常需要中断处理来达到目的; 类似地, 如果它想长时间的有效执行, 必须具备处理异常的能力。每次在进入中断异常处理前, 系统都必须首先保护现场。因为在中断异常处理结束后, 系统是需要重新回到尚未执行完的任务中去。这一部分中, 主要是完成两个部分: 一个部分是保存现场和状态, 主要是 CPU 寄存器和 CP0 寄存器; 另一个是将中断向量号、异常处理号与对应的处理函数对应起来。

我们通过设计在硬件中可以允许发生的异常, 归纳如下表 7.3.2 所示:

表 7.3.2 中断和异常处理号对应表

符号	编号	功能
EX_IRQ	0	Interrupt

EX_MOD	1	TLB Modify (write to read-only page)
EX_TLBL	2	TLB miss on load
EX_TLBS	3	TLB miss on store
EX_ADEL	4	Address error on load
EX_ADES	5	Address error on store
EX_SYS	8	Syscall
EX_BP	9	Breakpoint
EX_RI	10	Reserved (illegal) instruction
EX_CPU	11	Coprocessor unusable

相比通过设置中断门和异常门来实现系统对中断和异常的响应的方式，MIPS 主要是通过 CP0 给出的 CPU 的状态，然后按照上面的表格通过软件的方式进行处理。MIPS 的中断和异常的实现主要依赖于之前提到过的 CP0 的 Cause 和 EPC 来实现。这里，通过结合之前的 CP0 寄存器表格和异常处理号对应表，我们可以轻松的完成这一部分的内容。

事实上，在本系统中，所有的外部中断和异常都被使用一个 `handler` 函数进行处理，从而整个中断作为一个分支被纳入到异常处理分发中。因此，这里我们提到的异常包括以下几个方面：

1. 外部事件。在有外部事件时，中断被用来引起 CPU 的注意。中断是唯一由 CPU 执行以外的事件引起的异常。当前只能通过 `pic_disable` 来使得中断变得无效。这里将存在的外部事件有键盘、串口和时钟等基本设备。
2. 内存翻译异常。当 CPU 没有根据虚拟地址找到与之对应的物理地址时，或者当程序试图写一个有写保护的页面时，会发生此种异常。
3. 程序或硬件检查出的错误。包括 CPU 执行到非法指令,在不正确的用户权限下执行的指令，地址对齐出错等等。
4. 系统调用和陷入。某些指令只是用来产生异常。它们提供了一种进入操作系统的安全机制。

关于异常处理的内容可以参考前面的异常处理号对应表 7.3.2。与我们常看到 CISC 硬件或微指令把 CPU 分派到不同的入口地址不同，MIPS 的协处理器 CP0 会记住产生异常的原因，之后再内核中通过软件的方式派发到对应的异常处理函数进行解决。

下面，我们用流程图展示中断异常处理的设计流程：

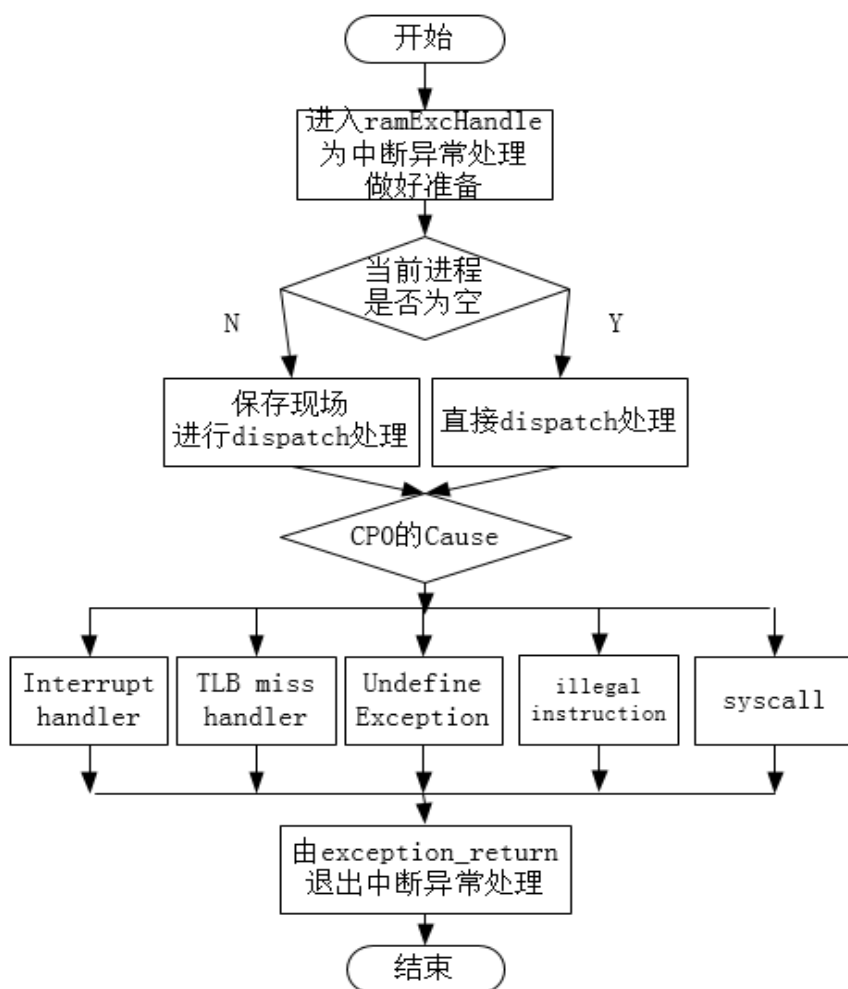


图 7.3.7 中断异常处理流程图

整个中断异常处理流程如下：

1. 发生中断或异常，首先进入通用处理入口 `ramExcHandle`，这里的操作保存当前进程现场，获取 CP0 寄存器信息等等，为中断异常处理做好准备。
2. 根据 CP0 的 Cause 寄存器来决定中断处理的函数或者异常处理的函数，大体上可以划分为 5 类：中断处理、TLB 缺失处理、非法指令处理、系统调用、未定义异常。
3. 根据各自的原因进行中断和异常的处理，如外部中断则根据中断向量号来找到对应的处理函数；系统调用则根据系统调用号找到对应的内核函数进行处理；TLB 缺失则调用相应平台的缺页填充机制或者将磁盘数据搬运到内存中。
4. 处理完中断异常后，统一进入 `exception_return` 过程，这一步将恢复之前保护的线程，修改 CP0 的异常状态，并继续执行程序。

7.3.5 以页为单位管理物理内存

在获得可用物理内存范围后，系统需要建立相应的数据结构来管理以物理页（按 4KB 对齐，且大小为 4KB 的物理内存单元）为最小单位的整个物理内存，以配合后续涉及的分页管理机制。每个物理页可以用一个 Page 数据结构来表示。由于一个物理页需要占用一个 Page 结构的内存空间，Page 结构在设计时须尽可能小，以减少对内存的占用。Page 数据结构的定义在 `kern-ucore/include/memlayout.h` 中，以页为单位的物理内存分配管理的实现在 `kern-ucore/default_pmm.c` 中。

为了与以后的分页机制配合，我们首先需要建立对整个计算机的每一个物理页的属性用

结构 Page 来表示，它包含了映射此物理页的虚拟页个数，描述物理页属性的 flags 和双向链接各个 Page 结构的 page_link 双向链表。

```
54 struct Page {
55     atomic_t ref;           // page frame's reference counter
56     uint32_t flags;        // array of flags that describe the status of the page frame
57     unsigned int property;  // used in buddy system, stores the order (the X in 2^X) of the
                             // continuous memory block
58     int zone_num;          // used in buddy system, the No. of zone which the page belongs to
59     list_entry_t page_link; // free list link
60 };
```

这里看看 Page 数据结构的各个成员变量有何具体含义。ref 表示这样页被页表的引用记数。如果这个页被页表引用了，即在某页表中有一个页表项设置了一个虚拟页到这个 Page 管理的物理页的映射关系，就会把 Page 的 ref 加 1；反之，若页表项取消，即映射关系解除，就会把 Page 的 ref 减 1。flags 表示此物理页的状态标记，进一步查看 kern-ucore/include/memlayout.h 中的定义，可以看到：

```
62 /* Flags describing the status of a page frame */
63 #define PG_reserved      0    // the page descriptor is reserved for kernel or unusable
64 #define PG_property      1    // the member 'property' is valid
65 #define PG_dirty         3    // the page has been modified
66 #define PG_active        5    // the page is in the active page list
```

这表示 flags 目前用到了两个 bit 表示页目前具有的两种属性，bit 0 表示此页是否被保留（reserved），如果是被保留的页，则 bit 0 会设置为 1，且不能放到空闲页链表中，即这样的页不是空闲页，不能动态分配与释放。比如目前内核代码占用的空间就属于这样“被保留”的页。在本实验中，bit 1 表示此页是否是 free 的，如果设置为 1，表示这页是 free 的，可以被分配；如果设置为 0，表示这页已经被分配出去了，不能被再二次分配。

在本实验中，Page 数据结构的成员变量 property 用来记录某连续内存空闲块的大小（即地址连续的空闲页的个数）。这里需要注意的是用到此成员变量的这个 Page 比较特殊，是这个连续内存空闲块地址最小的一页（即头一页，Head Page）。连续内存空闲块利用这个页的成员变量 property 来记录在此块内的空闲页的个数。这里去的名字 property 也不是很直观，原因与上面类似，在不同的页分配算法中，property 有不同的含义。

Page 数据结构的成员变量 page_link 是便于把多个连续内存空闲块链接在一起的双向链表指针。这里需要注意的是用到此成员变量的这个 Page 比较特殊，是这个连续内存空闲块地址最小的一页（即头一页，Head Page）。连续内存空闲块利用这个页的成员变量 page_link 来链接比它地址小和大的其他连续内存空闲块。

在初始情况下，也许这个物理内存的空闲物理页是连续的，这样就形成了一个大的连续内存空闲块。但随着物理页的分配与释放，这个大的连续内存空闲块会分裂为一系列地址不连续的多个小的不连续的内存空闲块，且每个连续内存空闲块内部的物理页是连续的。那么为了有效地管理这些小连续内存空闲块。所有的连续内存空闲块可用一个双向链表管理起来，便于分配和释放，为此定义了一个 free_area_t 数据结构，包含了一个 list_entry 结构的双向链表指针和记录当前空闲页的个数的无符号整型变量 nr_free。其中的链表指针指向了空闲的物理页。

```
85 /* free_area_t - maintains a doubly linked list to record free (unused) pages */
86 typedef struct {
87     list_entry_t free_list; // the list header
```

```
88         unsigned int nr_free;    // # of free pages in this free list
```

```
89 } free_area_t;
```

有了这两个数据结构，Hos 就可以管理起来整个以页为单位的物理内存空间。关于内存分配的操作系统原理方面的知识有很多，但在本实验中只实现了最简单的内存页分配算法。相应的实现在 default_pmm.c 中的 default_alloc_pages 函数和 default_free_pages 函数，相关实现很简单，这里就不具体分析了，直接看源码，应该很好理解。

其实实验二在内存分配和释放方面最主要的作用是建立了一个物理内存页管理器框架，这实际上是一个函数指针列表，定义如下：

```
19 struct pmm_manager {
20     const char *name;           // XXX_pmm_manager's name
21     void (*init) (void);        // initialize internal description&management data structure
22     // (free block list, number of free block) of XXX_pmm_manager
23     void (*init_memmap) (struct Page * base, size_t n);    // setup description&management
data structure according to
24     // the initial free physical memory space
25     struct Page *(*alloc_pages) (size_t n); // allocate >=n pages, depend on the allocation
algorithm
26     void (*free_pages) (struct Page * base, size_t n);    // free >=n pages with "base" addr of
Page descriptor structures(memla    yout.h)
27     size_t(*nr_free_pages) (void); // return the number of free pages
28     void (*check) (void);    // check the correctness of XXX_pmm_manager
29 };
```

对于这个内存管理器框架而言，其重点是实现 init_memmap/ alloc_pages/ free_pages 这三个函数。这里，读者应该知道如何获得系统当前的剩余内存容量了把？