# 第二部分 Hos-mips操作系统

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——Hos-mips。

第二部分内容分为三个章节：第一个章节准备Hos-mips操作系统的编译环境，构建（build）能够在MIPSfpga平台上运行的操作系统；第二个章节在Windows环境中搭建Hos-mips操作系统的开发调试环境；第三个章节完成在Hos-mips操作系统中的3个实验，帮助读者理解从用户态进程到内核态例程的调用路径。

## 实验5：Hos-mips操作系统的构建与运行

### 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-mips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。

### 实验内容

Hos-mips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节），并在安装完成后构建并运行Hos-mips操作系统（见5.2.2节和5.2.3节）。

#### 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。如图5‑1下载cygwin的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。

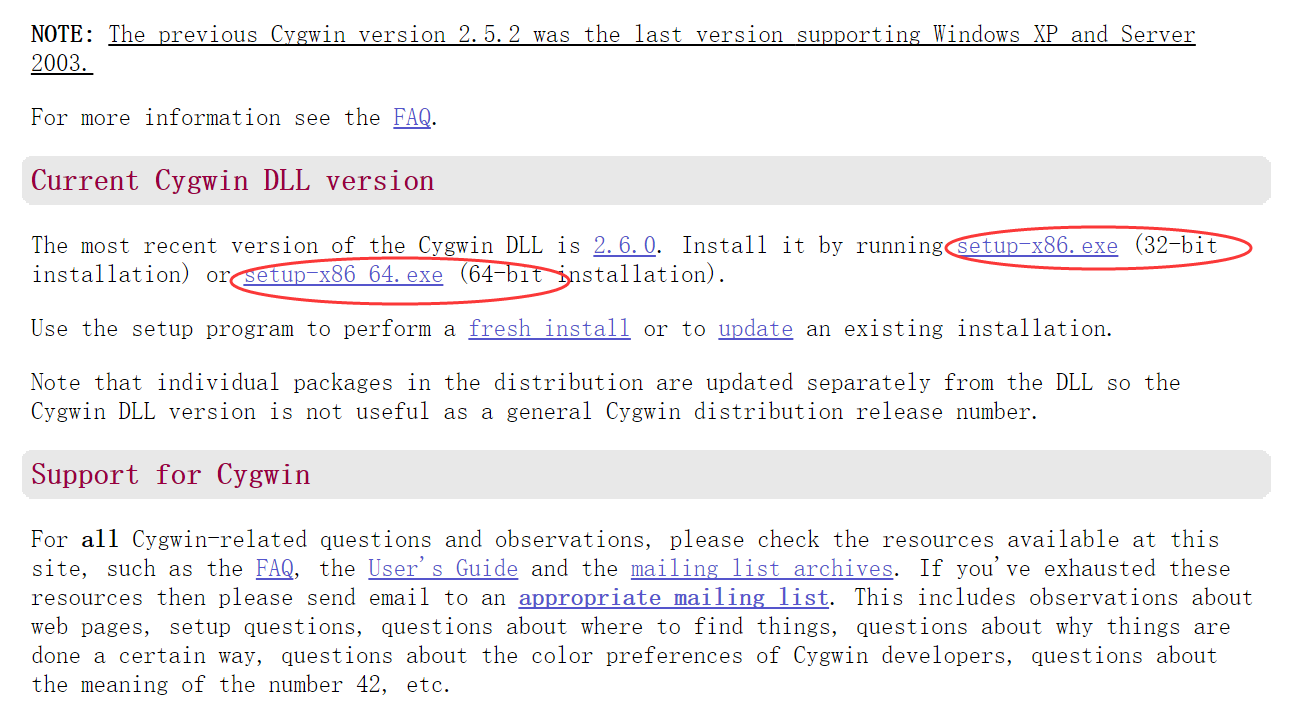


图 5‑1下载cygwin的安装文件

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，如图5‑2运行cygwin的安装文件所示，运行setup-x86\_64.exe：

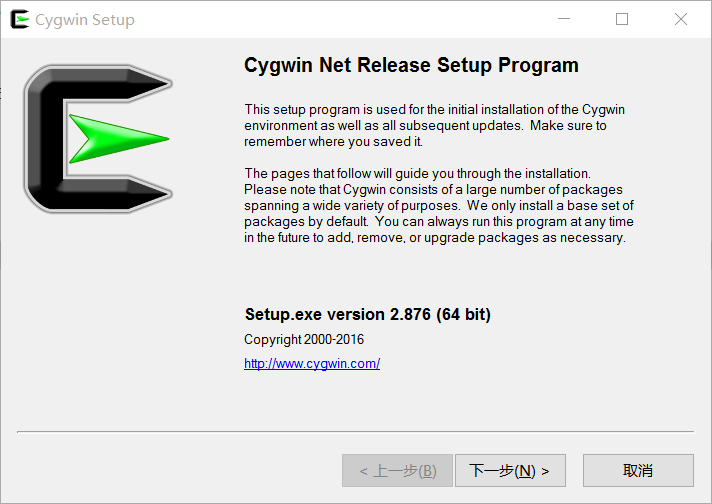


图5‑2运行cygwin的安装文件

如图5‑3点击“下一步”，并选择“Install from Internet”（从网络安装）；

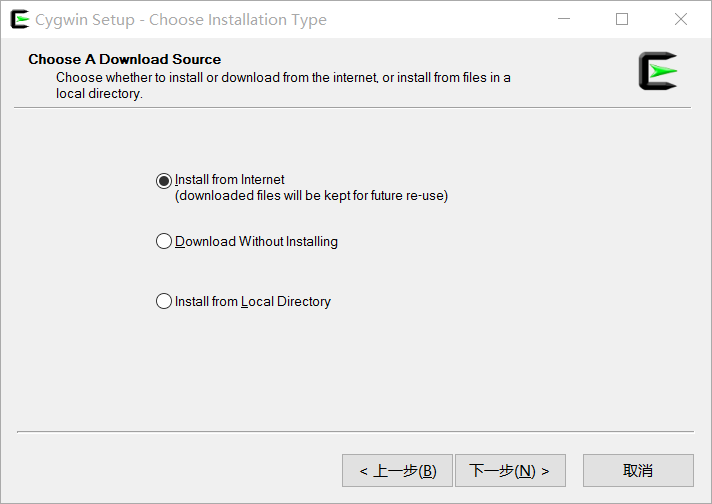


图5‑3选择Cygwin的安装方式

接下来选择安装目录以及安装包的缓存目录：

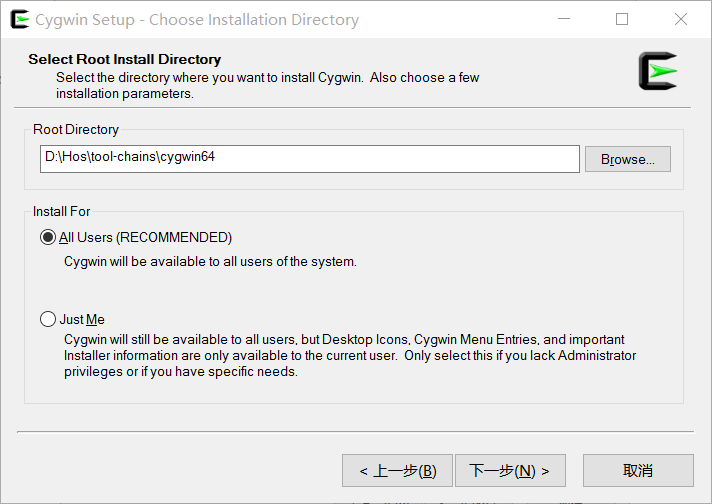
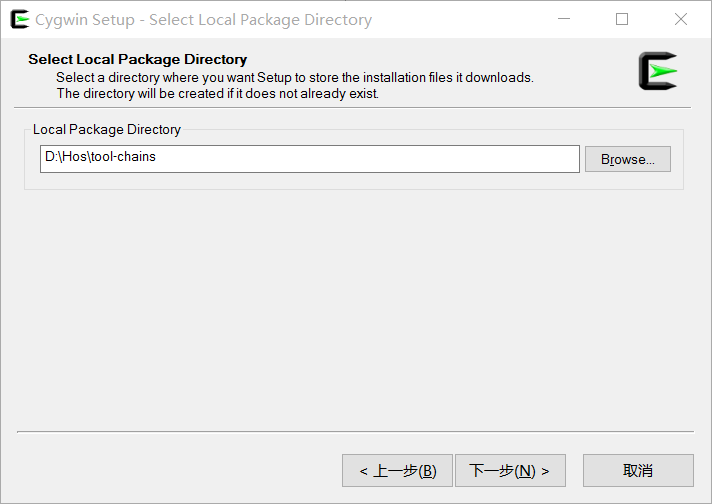
 

图5‑4选择Cygwin的安装路径

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。

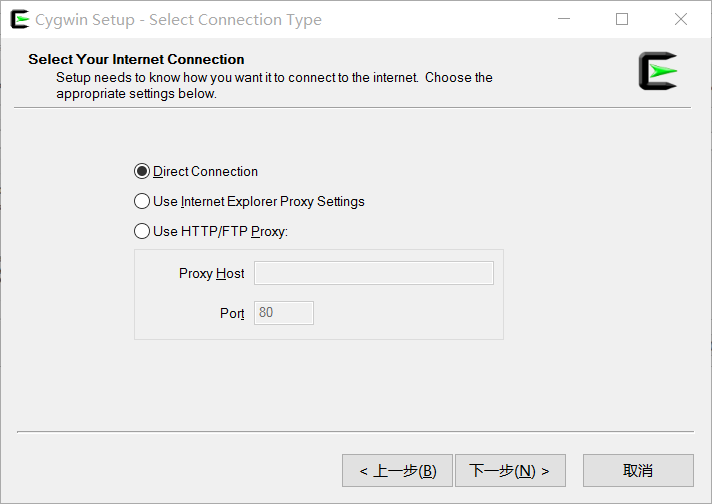
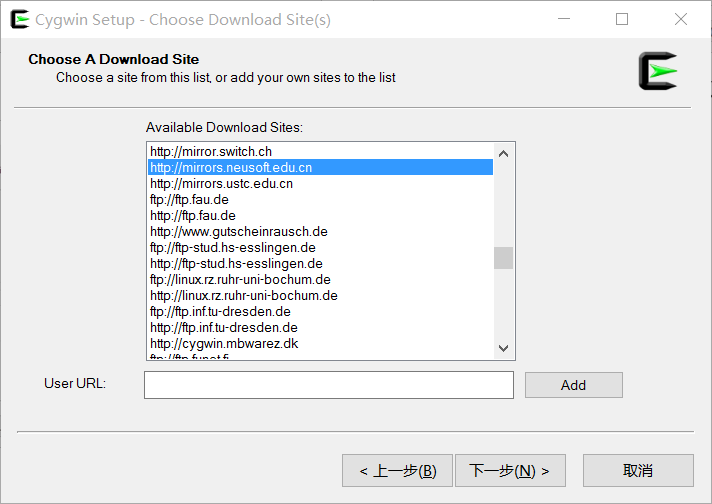
 

图5‑5选择Cygwin的安装源

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，并显示如下图5‑6的界面。

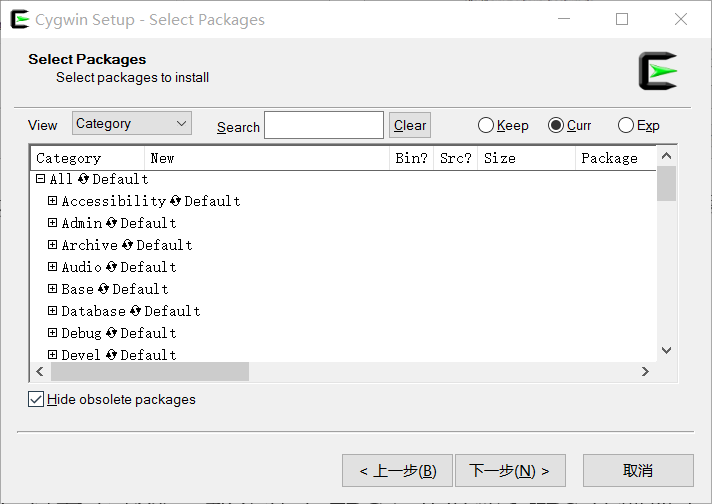


图5‑6 Cygwin的默认配置

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。如图5‑7所示，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”，并在界面中间的安装列表刷新后，点开Devel前的“C:\Users\zhiyuan\AppData\Roaming\Tencent\Users\450259336\QQ\WinTemp\RichOle\G`I%$3HCR6L[NVDOWJOXENM.png”号，并在展开的列表中的“make： The GNU version of the 'make' utility”前的“C:\Users\zhiyuan\AppData\Roaming\Tencent\Users\450259336\QQ\WinTemp\RichOle\D})`{F`4~A[OJZ@QF~X9]AQ.png”上单机，直到显示出即将要安装的make软件工具的版本为止。

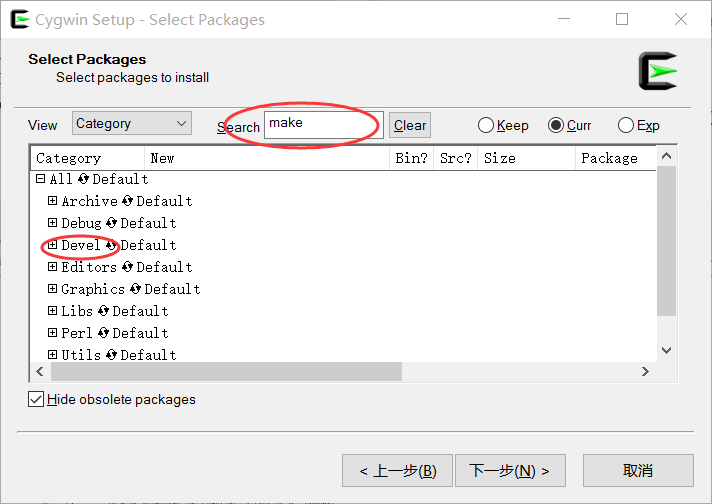
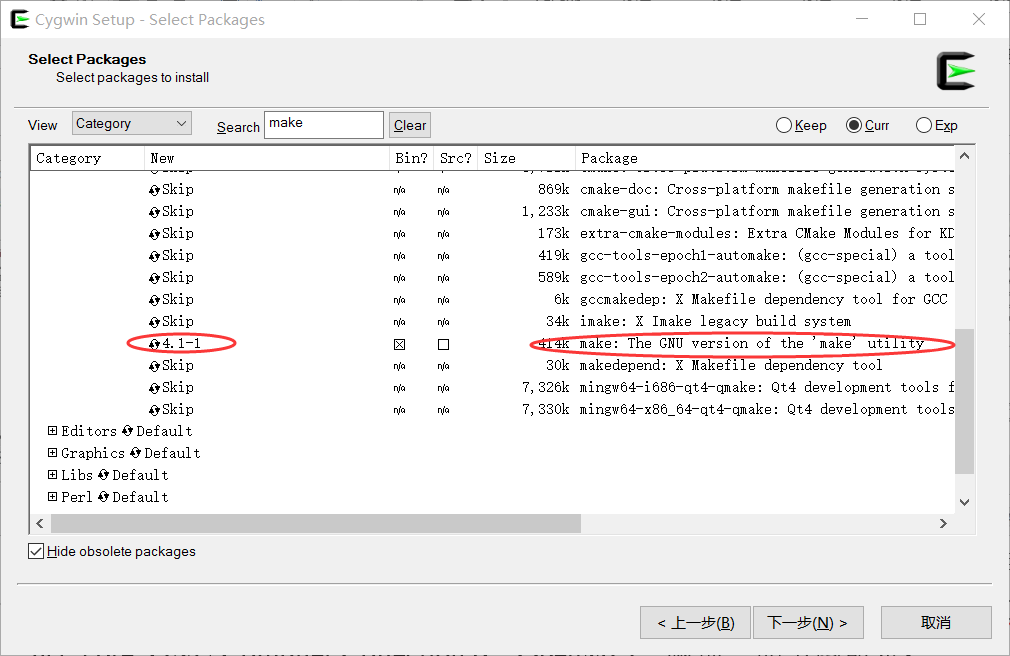
 

图5‑7在Cygwin中安装make

此时，千万不要点击“下一步”，因为我们还要其他软件包需要安装！我们再在Search输入栏中输入“gcc”，并选择其Devel中的“gcc-core： GNU Compiler Collection （C, OpenMP）”和“gcc-g++： GNU Compiler Collection （C++）”两项，如下图5‑8所示：

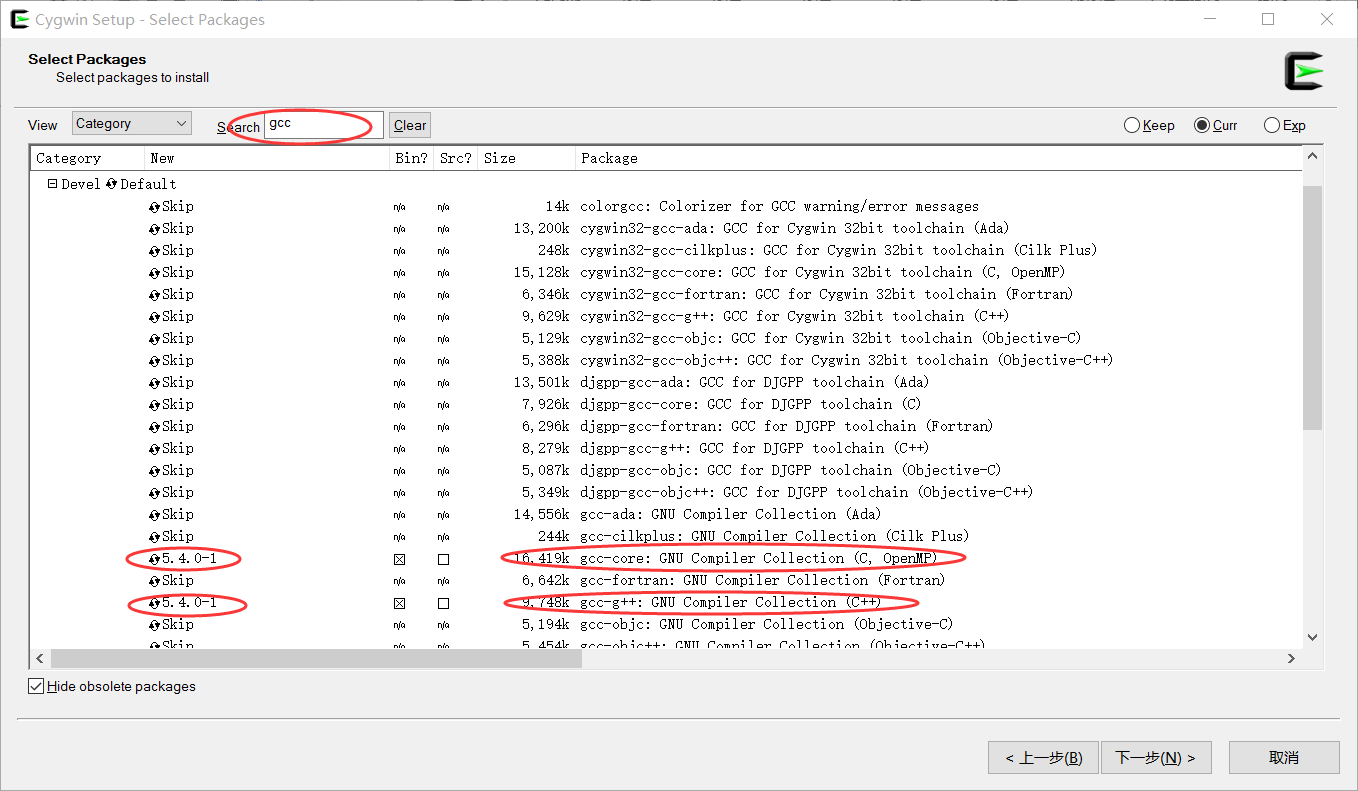


图5‑8在Cygwin中安装gcc

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5‑9所示：

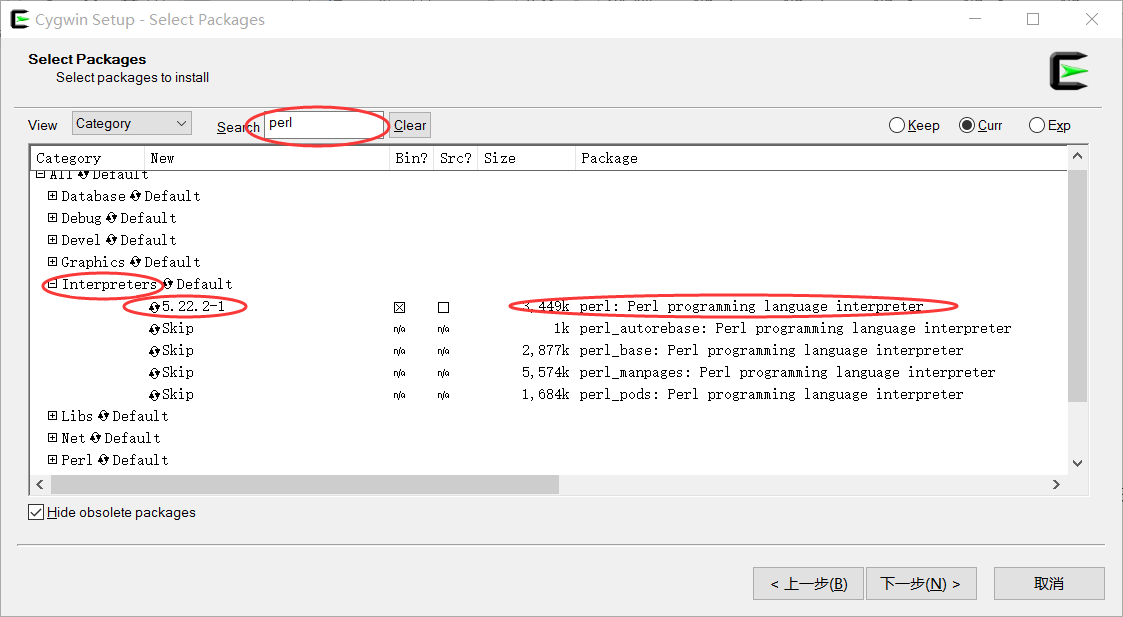


图5‑9在Cygwin中安装perl

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。在安装的最后一个界面上，选中“Create icon on Desktop”，如下图5‑10所示：

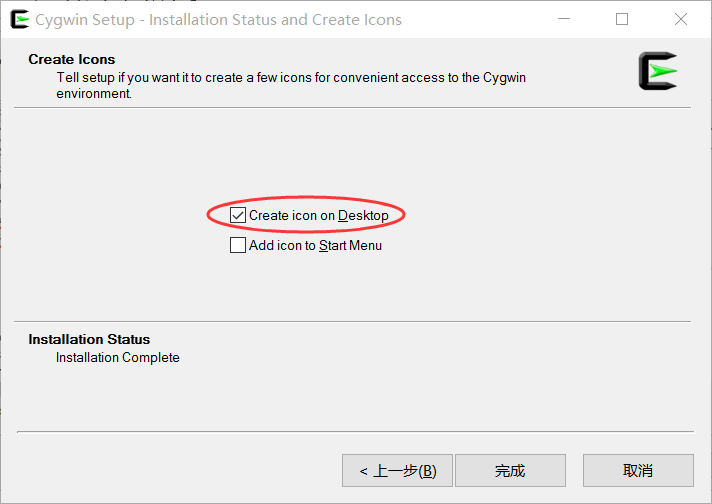


图5‑10完成Cygwin的安装

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5‑11所示：

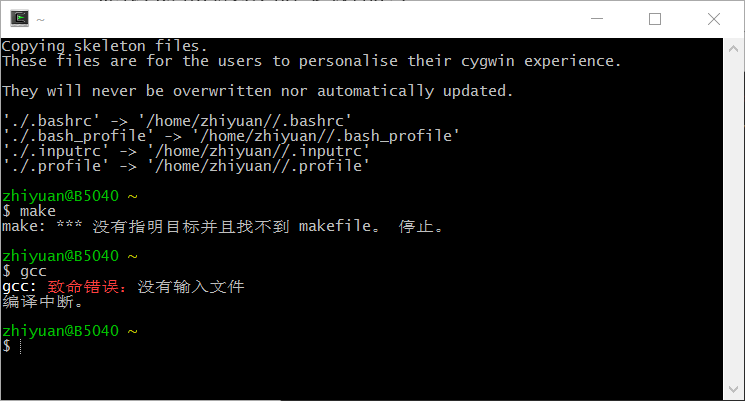


图5‑11运行Cygwin

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]](#endnote-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）。

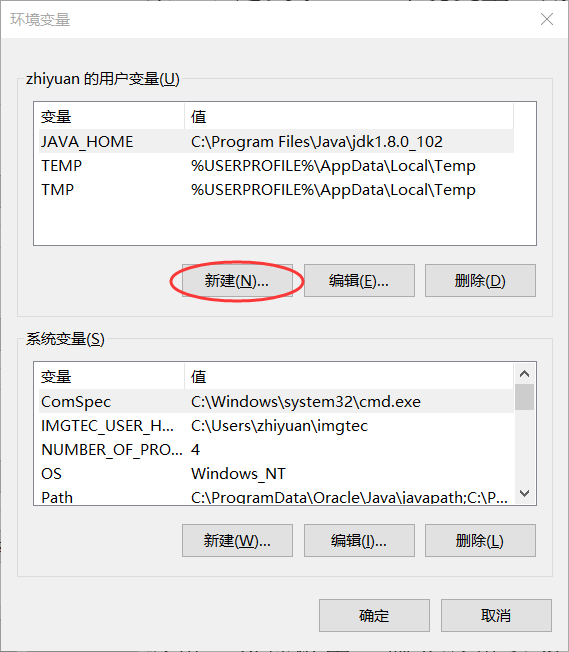
 

图5‑12添加环境变量（Windows10）

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”，如图5‑13所示：

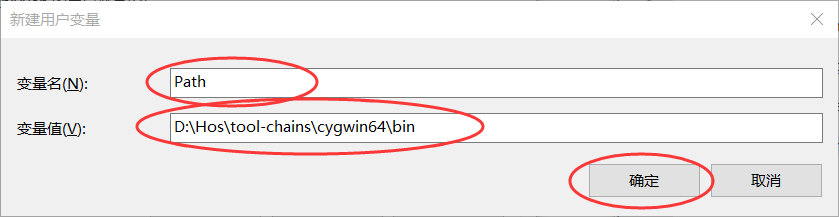


图5‑13环境变量Path

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-mips源码

接下来访问https：//github.com/mrshawcode/hos-mips，并下载Hos-mips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5‑14所示。

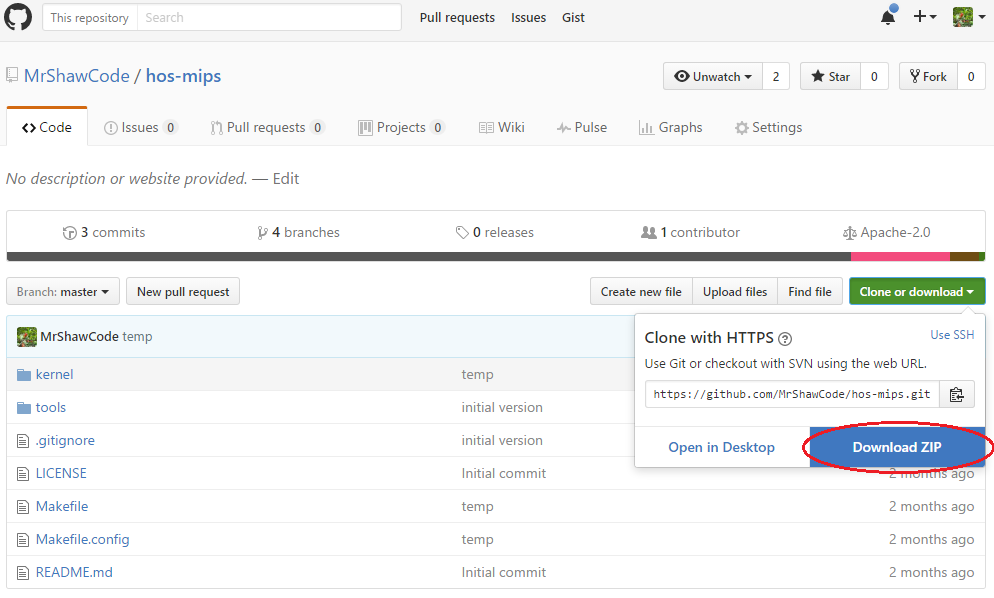


图5‑14下载Hos-mips源代码

现在，我们假设读者已经下载Hos-mips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：

源代码中目录及文件的说明见下表。

表5-1 Hos-mips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-ucore.txt）等。 |
| kernel-ucore目录 | Hos-mips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。

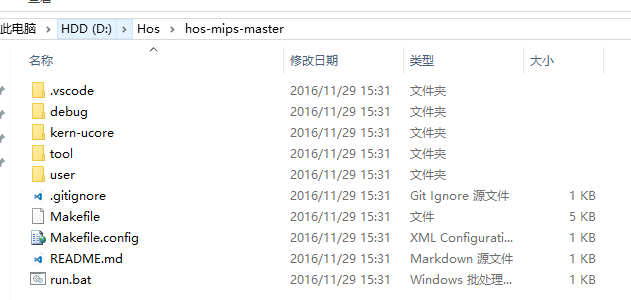


图5‑15解压Hos源代码

#### 构建Hos-mips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 6‑16所示。

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

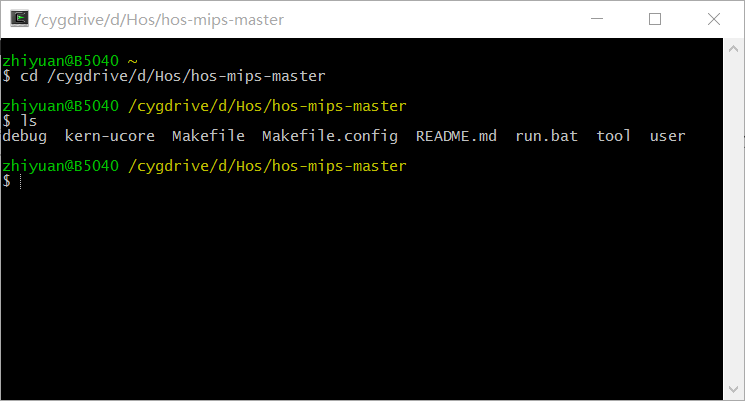


图5‑16启动Cygwin并进入Hos源代码目录

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，会出现如图5‑17（类似的）界面：

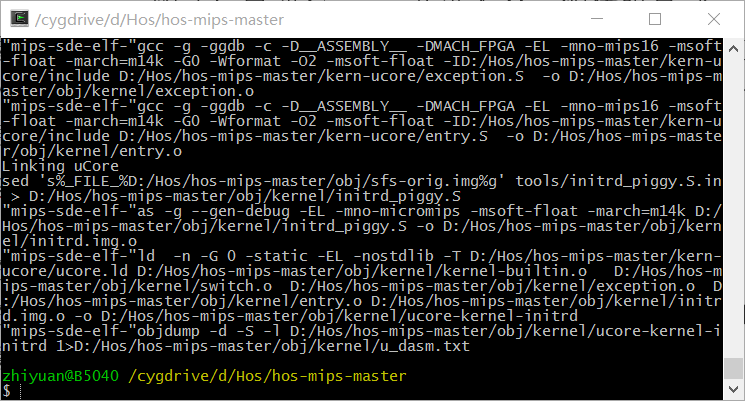


图5‑17 Hos系统的构建过程

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5‑18所示：

C:\Users\zhiyuan\AppData\Roaming\Tencent\Users\450259336\QQ\WinTemp\RichOle\DX`10RFE06DE9AQBL7~B$6M.png

图5‑18 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。

#### 运行Hos-mips

在成功生成Hos-mips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5-21）中的第二行，让Vivado在Hos-mips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5-1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 5‑19 “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5‑20所示的界面。

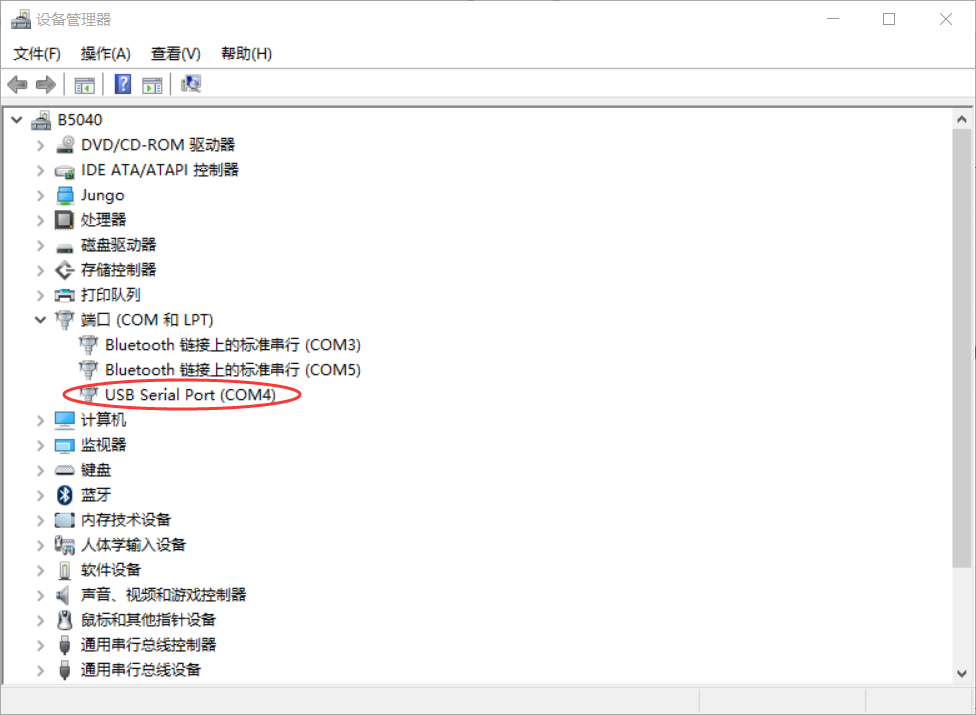


图5‑20查看Nexys4 DDR开发板所对应的COM端口

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-mips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5‑22）。

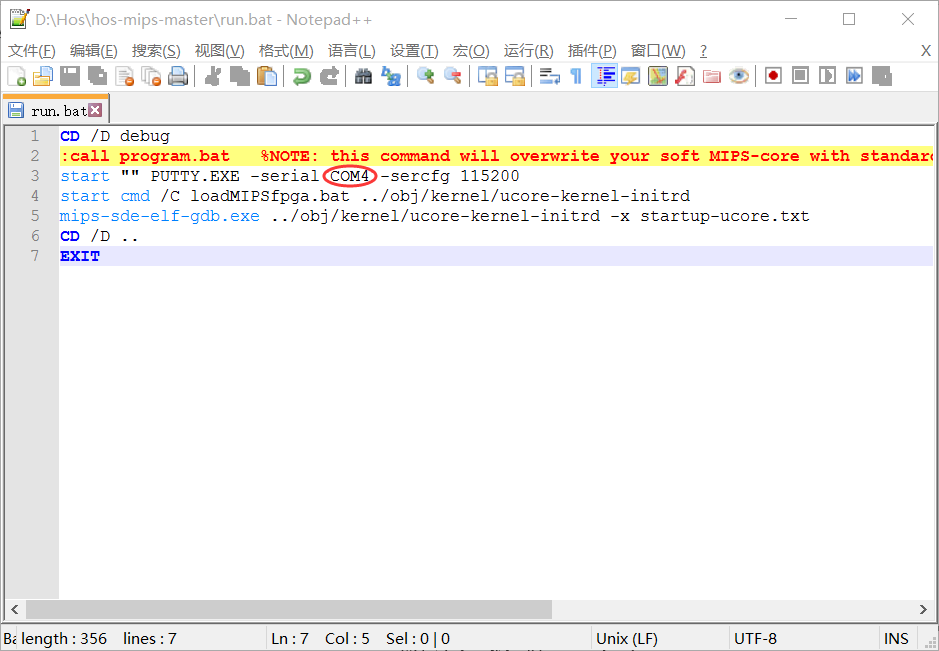


图5‑21 run.bat文件

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-mips操作系统了。执行该命令后，Cygwin中的显示会变成图5‑22所示：

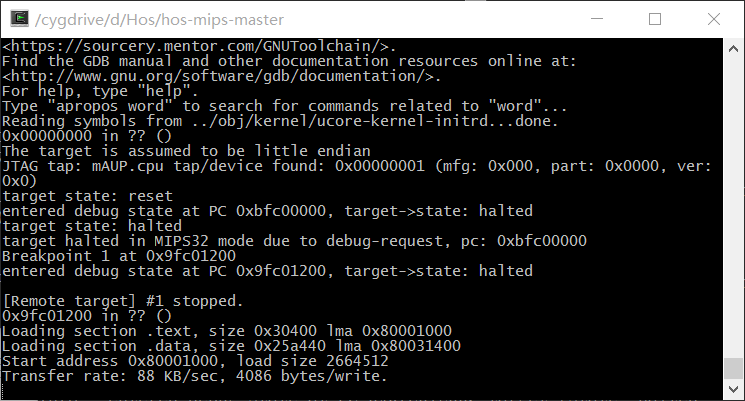


图5‑22 run.bat的执行结果

且会弹出以下两个窗口：

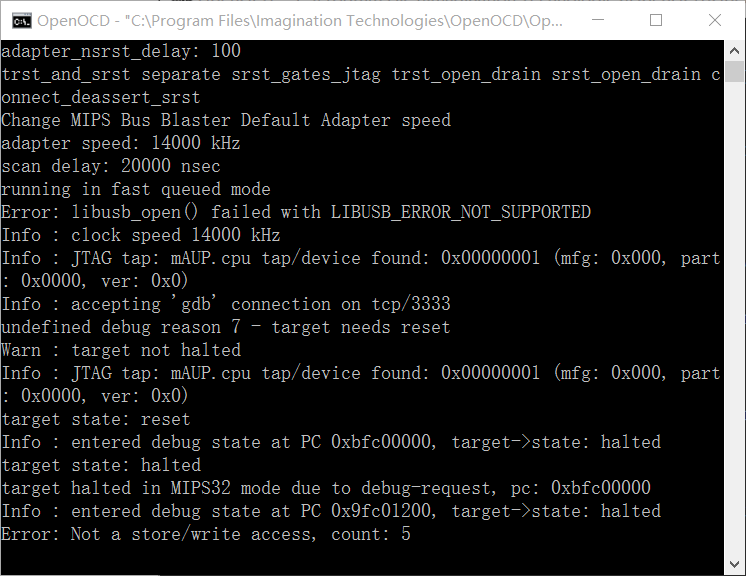
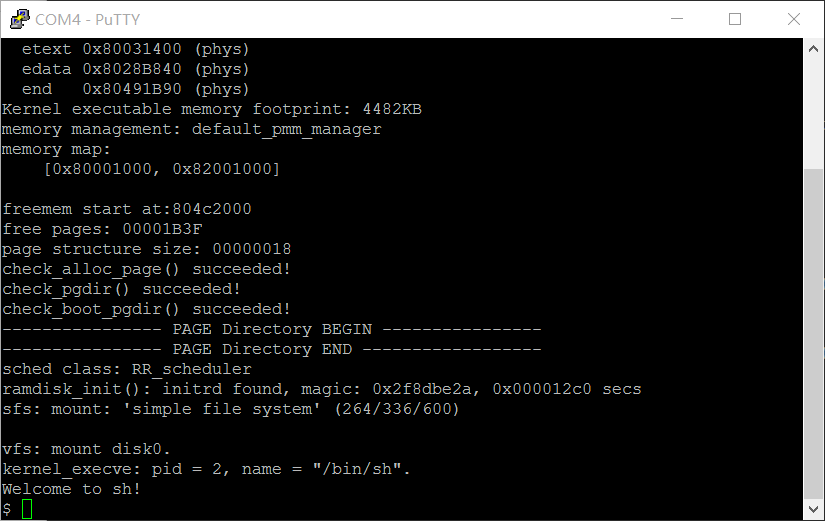
 

图5‑23 OpenOCD和Putty窗口

图5‑23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-mips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5‑21），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5‑23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 6‑23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-mips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-mips时（第7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。

### 实验背景及原理

#### Hos-mips简介

Hos-mips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。

#### 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）。

## 实验6：Hos-mips开发调试环境安装

### 实验目的

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

### 实验内容

#### 安装VSCode

我们选择VSCode （Visual Studio Code）作为Hos-mips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-mips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，如图6‑1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。

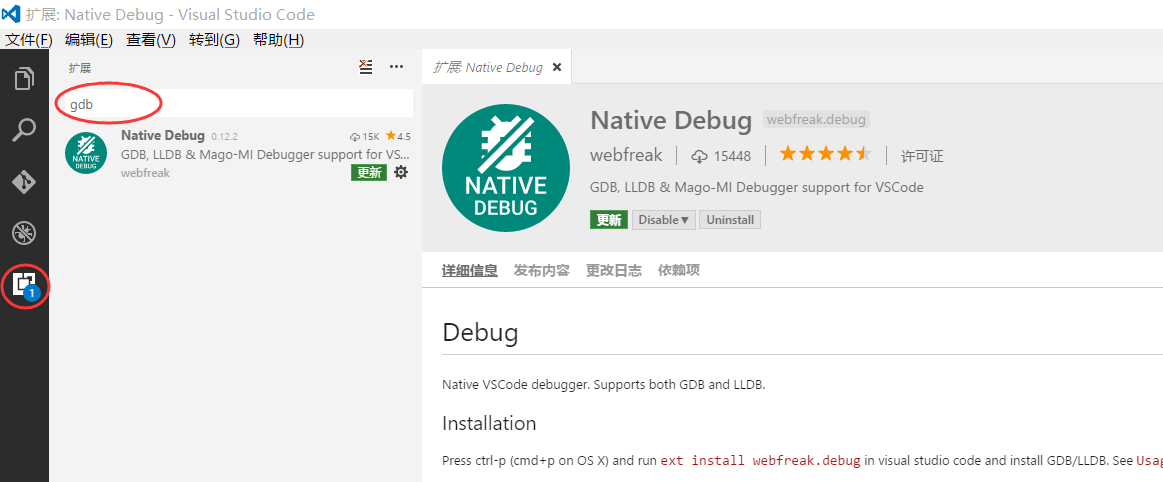


图6‑1在VSCode中安装Native Debug

安装完成后，应重启VSCode以进行之后的实验。

#### 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所示：

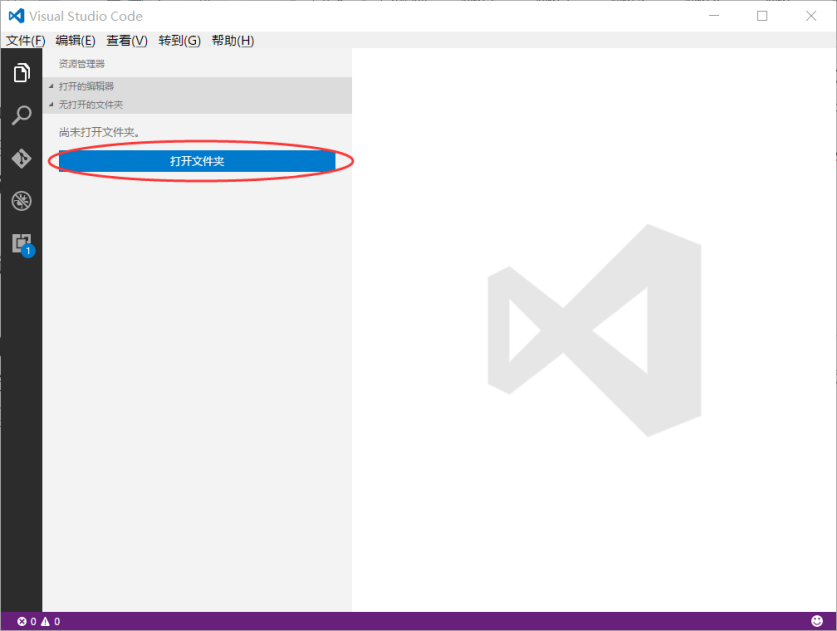


图6‑2用VSCode打开Hos-mips

接下来，在“打开文件夹”窗口中，浏览到Hos-mips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6‑3所示。

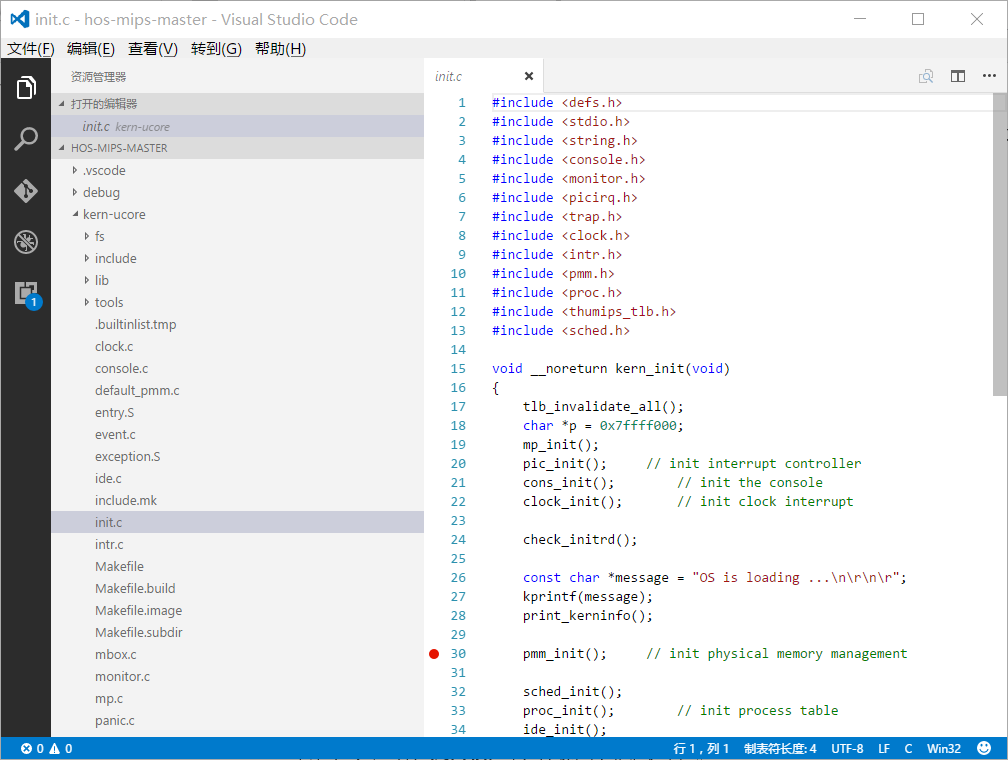


图6‑3用VSCode打开源代码文件

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

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6‑3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-mips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行也是出于演示的目的，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-mips系统，方法见5.2节），该文件的内容如图5‑21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-mips源代码所在的目录（方法见5.2节中描述），来到如图5‑16所示的界面。运行run.bat文件，系统会弹出如图5-23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-mips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-mips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6‑3所示：

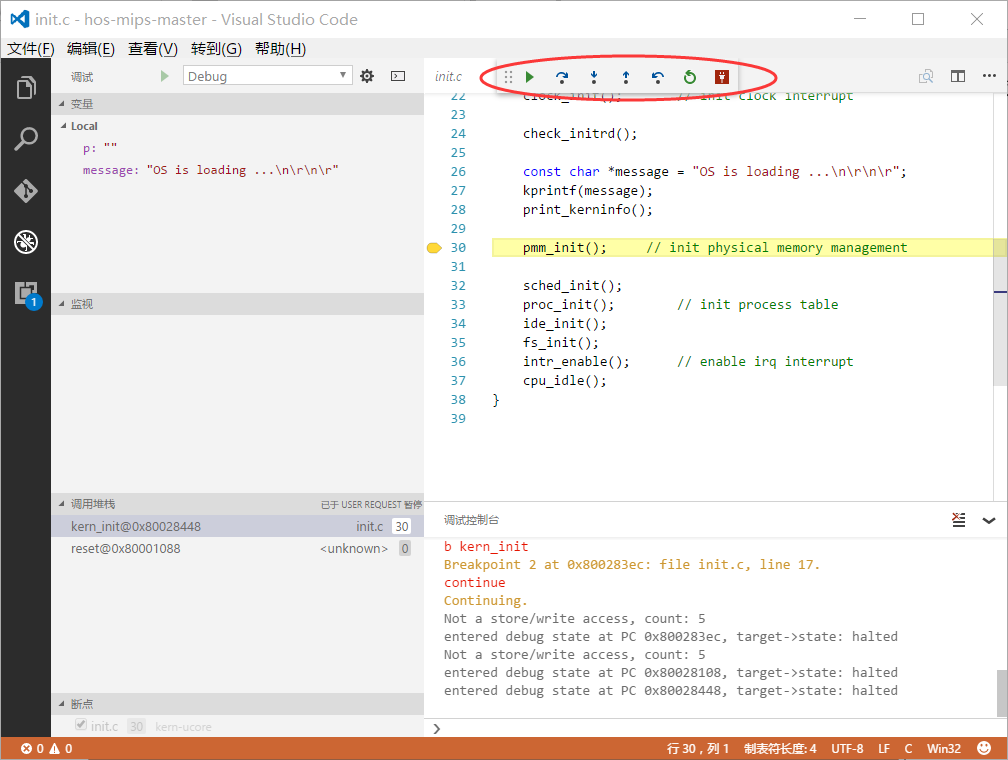


图6‑4 VSCode的调试界面

注意图6‑4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6‑5所示。

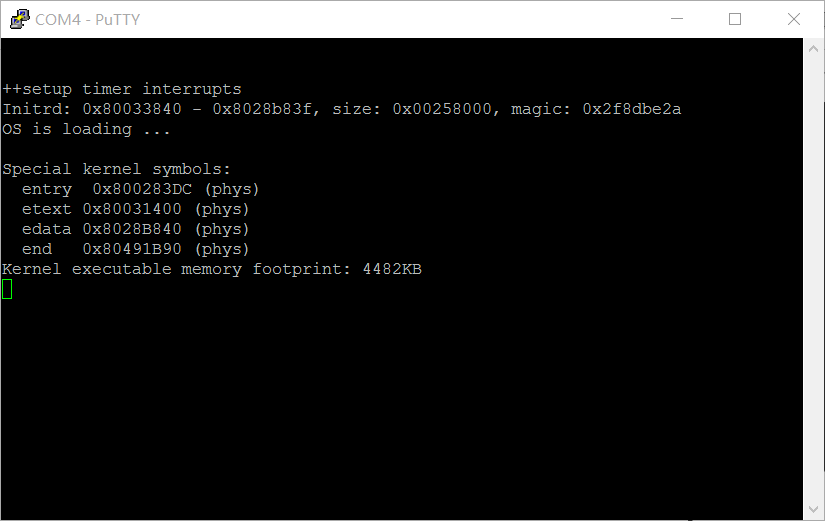


图6‑5调试过程中Putty的输出

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-mips编写程序并碰到Bug时非常有用。

### 实验背景及原理

#### 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-mips的目录结构（如图6‑6所示）：

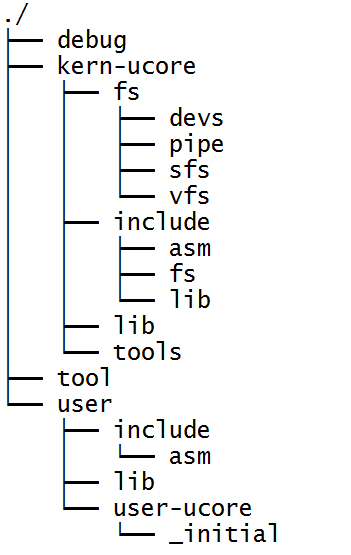


图6‑6 Hos-mips的目录结构

我们在VSCode中打开Hos-mips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-mips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 TMPSFS ：= $（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 $（TMPSFS）

117 @mkdir -p $（TMPSFS）/lib/modules

…

129 @cp -r $（OBJPATH\_ROOT）/user-ucore/bin $（TMPSFS）

130 ifneq （$（UCORE\_TEST）,）

131 @cp -r $（OBJPATH\_ROOT）/user/testbin $（TMPSFS）

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 $（TMPSFS）； \

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/\* $（TMPSFS）； \

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-mips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-mips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/.tmpsfs目录以及obj/.tmpsfs/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/.tmpsfs目录（Makefile的第129行）；

c） 将user/user-ucore/\_initial/hello.txt文件拷贝到obj/.tmpsfs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\*512B=2400KB≈2.4MB；

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-mips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\_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 BUILTINLIST=$（BUILTINLIST）

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，而不会再进行进一步的嵌套。

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-mips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 = . + 0x7ff0； /\* 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 }

……

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。

#### Hos-mips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。

## 实验7. 从应用到内核

### 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。

### 实验内容

#### 在Hos-mips操作系统中添加“hello world!”应用

与标准的Linux不同，Hos-mips并不提供丰富的编辑、编译工具（如vim、gcc等）。为了在Hos-mips中添加新的应用，就必须搞清Hos-mips系统的镜像生成过程。读者可以在本书的6-1节中，找到介绍Hos-mips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-mips的系统镜像中。一个比较简单的做法是将hello world应用视作Hos-mips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：

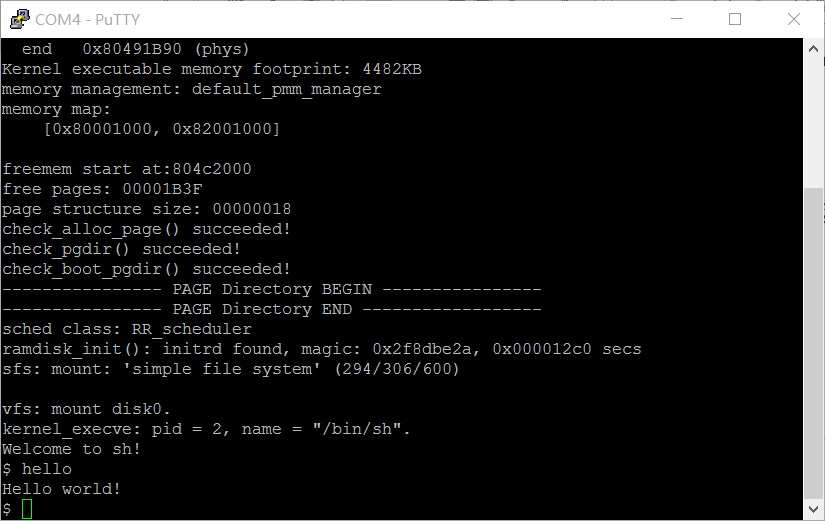


图7‑1 hello world!应用的输出

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-mips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。

#### 在Hos-mips操作系统中添加系统调用

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-mips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-mips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。

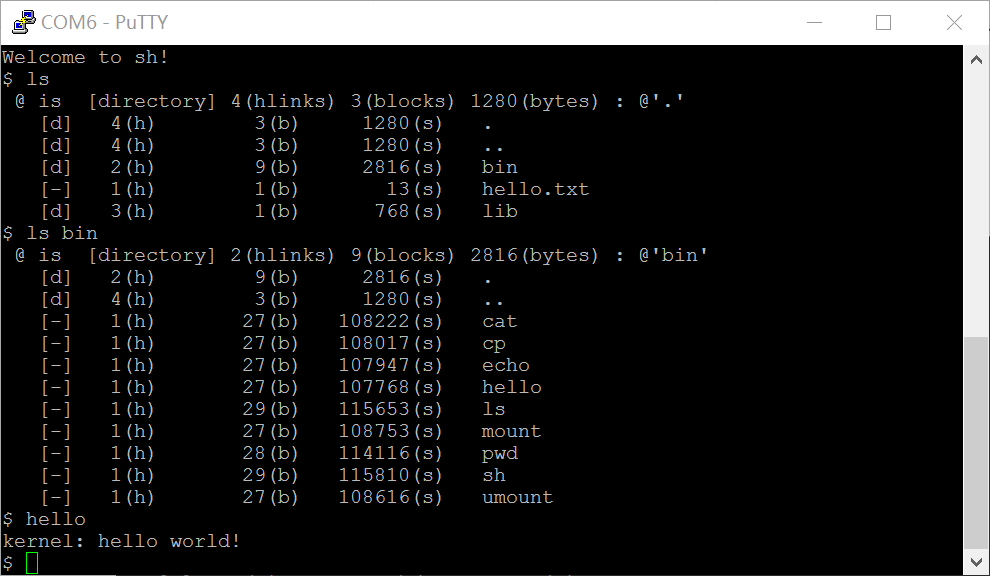


图7‑2 hello world!在操作系统特权态的输出

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7.1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所示。

#### 打印Hos-mips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-mips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7‑3所示：

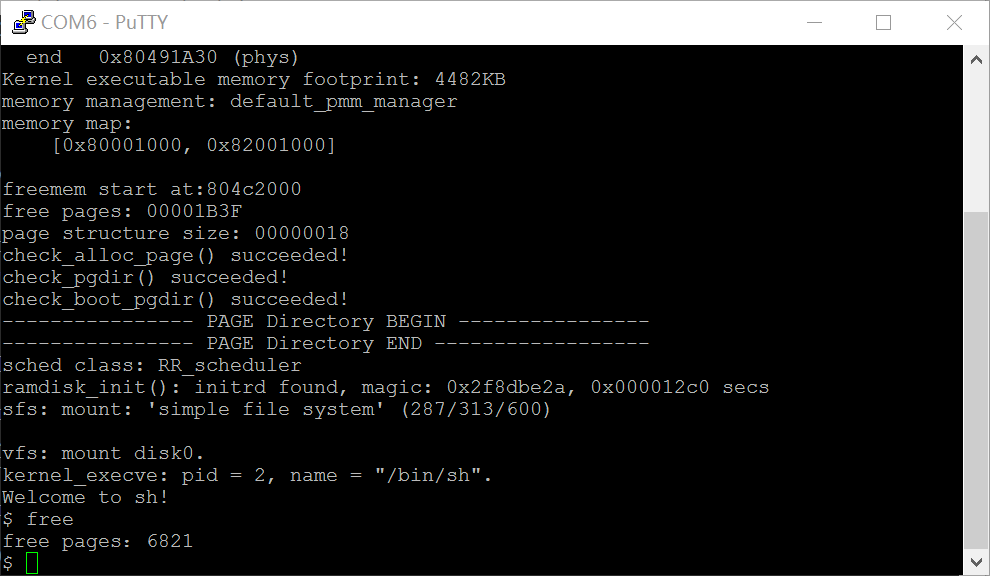


图7‑3显示Hos剩余内存容量

### 实验背景及原理

#### Hos-mips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。

#### 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7‑4 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。

#### Hos-mips的虚地址规划

Hos-mips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 8‑1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, 0x7fffffff] （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：

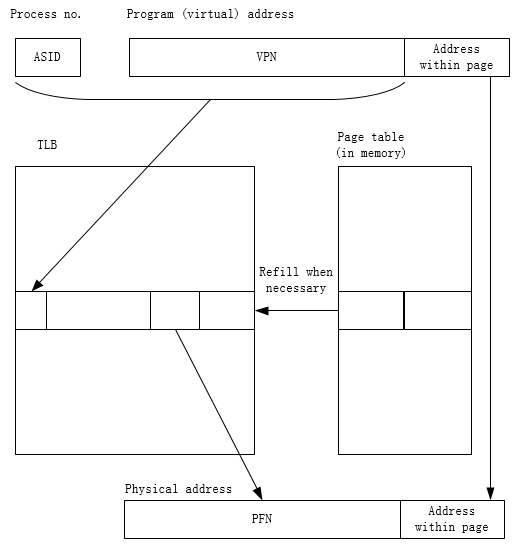


图7‑5硬件-虚实地址转换机制

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：

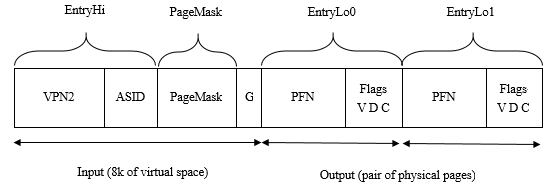


图7‑6 TLB中的一项

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7‑2‑1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  EntryLo1 | 2  3 | VPN所映射到的物理页号，以及对应物理页的存取权限 |
| 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 tlbwr 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  XContext | 4  20 | 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转译纪录的内存拷贝的安排方式，那么紧跟在 TLB 重装入陷入后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

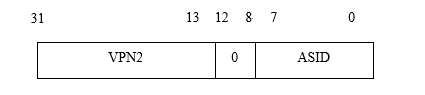


图7‑7 EntryHi寄存器结构

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

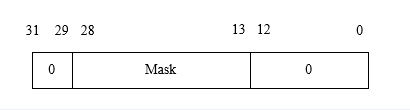


图7‑8 PageMask寄存器结构

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

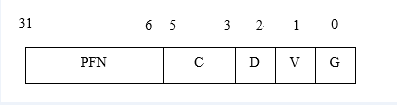


图7‑9 EntryLo寄存器结构

tryHi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。

#### 缺页异常与处理

1.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，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，则系统会讲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。

2.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7‑2所示：

表7‑2‑2MIPS的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hander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7-2。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。

如图7‑10所示，我们用流程图展示中断异常处理的设计流程：

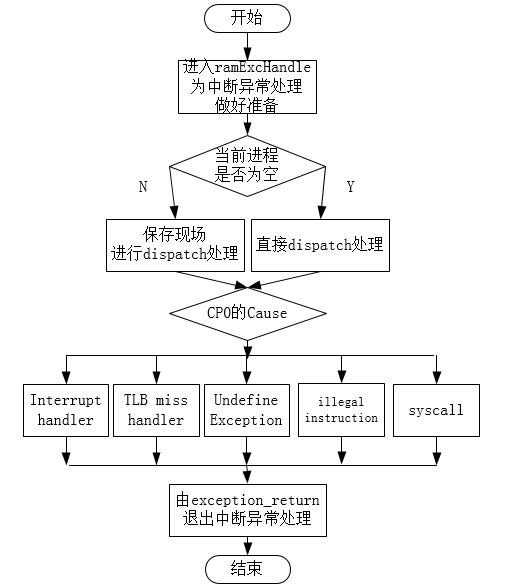


图7‑10中断异常处理流程图

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。

#### 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-mips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 structcure 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？

1. [↑](#endnote-ref-1)