

# 目 录

第 1 章 引言 .....	1
1.1 背景介绍 .....	1
1.2 主要工作 .....	1
1.3 下文组织 .....	2
第 2 章 开发基础 .....	4
2.1 先前工作 .....	4
2.2 硬件平台 .....	5
2.2.1 总体介绍 .....	5
2.2.2 DM9000A 以太网接口芯片 .....	6
2.2.3 DM9000A 芯片引脚功能详述 .....	6
2.2.4 DM9000A 芯片读写数据时序 .....	7
2.3 网络功能 .....	8
2.3.1 网卡使用流程 .....	8
2.3.2 网络协议栈 .....	10
2.4 文件系统 .....	10
2.4.1 Simple File System 总述 .....	10
2.4.2 索引节点 .....	11
2.4.3 文件数据 .....	11
第 3 章 网络功能 .....	13
3.1 层次结构 .....	13
3.2 硬件层 .....	13
3.2.1 中断信号 .....	14
3.2.2 数据读写 .....	14
3.3 驱动层 .....	15
3.3.1 寄存器访问 .....	15

3.3.2	网络芯片初始化 .....	15
3.3.3	接收数据 .....	16
3.3.4	发送数据 .....	17
3.4	基础协议 .....	18
3.4.1	ARP .....	18
3.4.2	IP .....	19
3.4.3	ICMP .....	19
3.5	TCP .....	19
3.6	网络功能工作流程 .....	20
3.7	网络功能实现细节 .....	21
3.7.1	多数据包处理 .....	21
3.7.2	TCP 状态转移 .....	22
3.7.3	连接建立 .....	22
3.7.4	接收数据 .....	22
3.7.5	发送数据 .....	23
3.7.6	标志位及状态码 .....	23
3.7.7	多 TCP 连接支持 .....	25
<b>第 4 章</b>	<b>文件系统功能拓展 .....</b>	<b>26</b>
4.1	ucore OS 的文件系统现状 .....	26
4.2	文件系统抽象层 .....	27
4.3	Simple FS 文件系统层 .....	27
4.4	具体实现 .....	29
4.4.1	新建文件流程 .....	29
4.4.2	功能函数 .....	29
<b>第 5 章</b>	<b>实验结果 .....</b>	<b>31</b>
5.1	功能测试 .....	31
5.2	测试结果 .....	32
5.2.1	文件传输测试 .....	32

5.2.2 多 TCP 连接测试结果 .....	33
第 6 章 结论 .....	34
6.1 内容总结 .....	34
6.2 不足与拓展 .....	34
6.3 结语 .....	35
插图索引 .....	36
表格索引 .....	37
参考文献 .....	38
致 谢 .....	39
声 明 .....	40
附录 A 外文资料的调研阅读报告或书面翻译 .....	41

# 第 1 章 引言

## 1.1 背景介绍

在计算机科学的教学中，实验既可以作为教学的辅助，又能让学生综合运用学到的知识进行练习，起着多种多样的作用，构成课程安排中重要的一环。<sup>[1]</sup> 实验的进行依赖于相应的实验平台，而受院系自身的课程安排影响，实验平台又有着多种不同的来源。在一些情况下，使用市场上已有的软硬件平台就可以满足课程的要求；但对于一些课程而言，市场上已有的软硬件平台不足以满足院系对课程的实验要求，因此自行开发也是实验平台的一个重要来源。<sup>[2]</sup> 在清华大学计算机系目前的课程安排中，针对一些比较复杂的实验自行开发了一系列实验平台，以更为灵活地设计相关的实验与教学。其中比较重要的包括 THINPAD 教学实验平台与 ucore OS。

THINPAD 教学实验平台<sup>[3]</sup> 采用 Xilinx 公司的 FPGA 芯片作为硬件核心，使用 SRAM 静态存储器作为内存，并且配备了多种外设接口或设备，比如串口、USB、VGA、RJ45、FLASH、网卡芯片，构成了一个完善的计算机硬件平台。THINPAD 既支持针对计算机结构进行功能、系统级别的设计开发，也能用于定制功能的硬件之上的软件开发。

ucore OS 以 MIT CSAIL PDOS 课题组开发的 xv6<sup>[4]</sup> 为基础进行开发，借鉴了哈佛大学开发的 OS161 教学操作系统以及 Linux-2.4 内核的相关代码。在这些项目的基础上，课程团队不断针对操作系统课程的教学需要进行功能的完善与代码结构的优化，从而为实验性质的功能实现、功能拓展提供了基础。另外，ucore OS 还开发了 32 位 MIPS 架构的版本，并且已经可以在 THINPAD 平台上运行。

## 1.2 主要工作

我们以目前已有的 THINPAD 上基于 FPGA 设计的硬件环境与 ucore OS 为基础，实现了对网络功能的支持以及对文件系统新建文件功能的拓展。

在网络功能的支持方面，我们实现了对 THINPAD 上的 DM9000A 以太网接

口芯片的硬件支持与软件支持，并且以此为基础实现了网络层与传输层的一系列协议，最终实现了基于 TCP/IP 协议栈、通过有线网络进行的实验板与 PC 端的连接与数据传输。与之前的尝试相比，我们真正地将网络功能集成到操作系统中，对用户端提供 POSIX socket API 的简化版接口，使得可以用与广泛使用的 POSIX socket API 相似的方法来使用网络功能。在本项目中，我们采用中断处理的方式处理网络数据包的接收，并且使用缓冲区处理发送、接受数据，从而可以无阻塞地发送数据，并且可以在调用接受数据函数之前收到的数据不会丢失；我们还在操作系统中进行了多连接同时存在的支持，实现了多个 TCP 连接同时存在、同时工作。

在文件系统的拓展方面，我们对 ucore OS 当前的文件系统进行了研究，参考 PC 端生成文件系统镜像以及内存中加载文件系统的步骤，实现了新建文件的功能。同时，结合网络功能，我们实现了一个简单的文件传输程序，可以将 32 位 MIPS 的可执行程序从 PC 端传输到 THINPAD 上的 ucore OS 中，并成功运行。

### 1.3 下文组织

第二章介绍了本次工作的开发基础。首先，对之前已有的相关开发、尝试进行了简要介绍，说明了本次工作之前已有的尝试，并且指出了哪些之前已有的工作被用到我们的实现中。然后，介绍了 THINPAD 硬件平台的具体内容，着重介绍了 DM9000A 网络芯片的使用步骤与相关参数，这也是硬件、软件方面实现支持的基础。之后，对需要支持的网络协议进行了简要介绍，并且说明会如何简化协议的实现，从而将功能支持集中在一个比较具体但又足够常用的使用环境。最后，介绍了 ucore OS 使用的文件系统 Simple File System 的数据保存格式。

第三章介绍了在 THINPAD 与 ucore OS 这一软硬件平台上，对网络功能的支持。首先，对网络功能的支持进行了层次划分，将这部分工作分为四个部分。然后，按照之前划分的层次，逐个介绍各层的实现方式。最后，对网络功能的实现中遇到的一些具体问题与解决方案进行了介绍。

第四章介绍了对 ucore OS 的文件系统的功能拓展。首先，对 ucore OS 的文件系统的现有功能进行了介绍。然后，按照文件系统上的虚拟文件系统与 Simple File System 的划分，对创建文件的流程进行了介绍。最后，介绍了我们的实现中的一些细节。

第五章给出了功能测试的流程与结果，验证了我们成功地实现了网络功能与文件系统功能拓展。

最后一章对我们的工作进行了总结，并对其中有待改进、拓展的地方进行了说明。

## 第 2 章 开发基础

### 2.1 先前工作

以 THINPAD 与 ucore OS 为基础，此前已经有许多功能拓展，接下来会介绍与本次项目有关的功能拓展。

通过使用硬件描述语言对 FPGA 进行编程，可以在 THINPAD 上实现定制功能的硬件器件。目前，已经有基于 32 位 MIPS 架构的流水线 CPU 实现<sup>①</sup>，并且在硬件上实现了 TLB、部分 CP0 协处理器、中断信号、外设读写等功能，可以支持 32 位 MIPS 的大多数指令，允许软件实现虚拟内存管理，允许软件端实现时钟中断、I/O 中断信号、异常处理，并且已经可以支持串口、FLASH、VGA 等外设。同时，这一实现中也提供了一个适配这个硬件平台的 ucore OS，可以使用 Sourcery CodeBench Lite 2012.03-64 for MIPS ELF<sup>②</sup>工具链编译操作系统与应用系统。我们的工作主要基于这一软硬件平台进行开发，对于这一实现中与我们的工作无关的内容不会作太多介绍，如需了解请查看相关文档。

另外，还有以上述 CPU 实现为基础的实现<sup>③</sup>，增加了对网络芯片的读写支持与中断信号支持，但这一实现对芯片时序的支持不完善。

在网络功能方面，此前已经有多个相似的实现<sup>④⑤</sup>，已经可以让网络芯片正常工作。但在这些实现中，实际的收发数据的过程都是在用户程序中实现，这无论是从安全性还是从设计的合理性上都有所不足，而且与 PC 端编程的接口使用方式差别很大。另外，这些实现接收数据包采用循环查询的方式，仅能在主动查询的时候接收数据，没有对 TCP 数据包的 ACK 进行处理，也没有加入多连接的支持，使得发送、接收数据的使用方式受限。除此之外，之前的实现采用回调函数的方式实现接收数据，对在 ucore OS 目前的使用环境及类似使用环境下最常用的基于同步 I/O 的应用程序的编写极为不利。

我们的实现使用了之前实现中对 DM9000A 芯片的硬件支持的大部分代码，

---

① <https://git.net9.org/armcpu-devteam/armcpu/tree/master>

② <https://sourcery.mentor.com/GNUToolchain/release2189>

③ <https://github.com/cjld/armcpu>

④ <https://github.com/blahgeek/MadeAComputerIn20Days>

⑤ <https://github.com/cjld/armcpu>

进行了一定改进；使用了之前实现中对 DM9000A 芯片的软件支持的部分代码，主要是芯片的初始化与常量定义部分的代码，收、发数据包的较低抽象层的代码根据我们的需要进行了一定的修改，使用中断接收数据的部分完全重写；网络层协议根据需要进行了一定修改；传输层的 TPC 协议使用了少量已有的功能代码，从功能设计到具体实现完全由我们完成。

在文件系统方面，目前已经有对新建文件功能的尝试，该实现在我们使用的软硬件平台上会引入错误，运行后有一定概率会使操作系统崩溃。我们参考了之前的实现方式，使用相似的思路实现了我们的工作流程。

## 2.2 硬件平台

### 2.2.1 总体介绍

本节对 THINPAD 上的设备进行简单介绍。我们使用的 THINPAD 实验板为 32 位版本，核心芯片为一块 Xilinx Spartan6 xc6slx100 FPGA。THINPAD 的元件清单如表 2.1 所示。

表 2.1 THINPAD 的元件清单

元件	数量与参数
Xilinx Spartan6 xc6slx100 FPGA	1 块
CPLD	1 块
RAM	1 块, 32bit 字长, 共 8M
FLASH	1 块, 16bit 字长, 共 8M
串口	2 个
USB 串口	1 个
ps/2 接口	1 个
DM9000A 以太网接口芯片	1 块
10/100M RJ45 插座	1 个
VGA 接口	1 个
数码管	2 个
自复位开关	4 个
LED 灯	16 个
拨码开关	16 个



我们的工作中，需要开发的主要是 DM9000A 芯片，RJ45 插座由以太网芯片处理。

### 2.2.2 DM9000A 以太网接口芯片

DM9000A 芯片是一块 48 引脚的芯片，在 THINPAD 实验平台上，DM9000A 与 FPGA 直接相连的引脚有 23 个，其他引脚或是与电源、其他支持器件有关，用于支持 DM9000A 芯片工作，或是没有被连接到 FPGA，不提供该功能。FPGA 可以使用的 23 个引脚，包括 16 个数据引脚，CMD 引脚、CS 引脚、INT 引脚、IOR 引脚、IOW 引脚、RESET 引脚、时钟引脚各一个，其中时钟引脚需要接收 25M 时钟信号，RESET 引脚用于硬件重置，都是外设比较常见的引脚，本文不进行特别说明。接下来对 FPGA 会用到的其他引脚的功能进行说明，下一小节如未特加说明，以 DM9000A 一端来阐述数据传输，比如“输出”指的是 DM9000A 发出信号。另外，由于开发的角度更偏向软件端，本文的大部分内容会从软件的角度来看待数据流动，与下一小节的情况相反。

### 2.2.3 DM9000A 芯片引脚功能详述

INT 引脚在收到新数据包后会输出高电平，在收到消除中断信号的操作后会输出低电平。这个信号可以用于产生中断，使操作系统触发对网络数据包的接收处理。需要注意的是，INT 引脚只是在收到新数据包时输出高电平，而不是在缓冲区内有没有处理的数据时输出高电平，因此如果消除中断信号之前没有处理所有收到的数据包，中断信号无法帮助操作系统处理剩余的数据包。这使得基于中断的方法与基于循环查询的方法需要使用不同的方法来处理数据。

CS 引脚用于选片，在我们的实现中不会用到 EEPROM，因此一直输入低电平即可。

CMD 引脚用于选择指令类型。输入高电平表示选择数据端口，输入低电平表示选择地址端口。下一节会对这个引脚的使用进行具体介绍。

IOR 引脚、IOW 引脚都是在输入低电平时使能读、写操作。下一小节以及下一节会对这两个引脚的使用进行具体介绍。

16 个 DATA 引脚用于读写数据，外界设备从 DM9000A 读数据时应发送高阻信号，向 DM9000A 写数据时应发送数据内容。

功能上来看，DM9000A 具备 16 位模式与 8 位模式两种工作状态。但实际

上 THINPAD 将 DM9000A 芯片的工作状态固定在 16 位模式，没有提供选择的机会。

2.2.4 DM9000A 芯片读写数据时序

根据 DM9000A 数据手册<sup>[5]①</sup>，DM9000A 芯片读、写操作的时序如图 2.1 与图 2.2 所示。

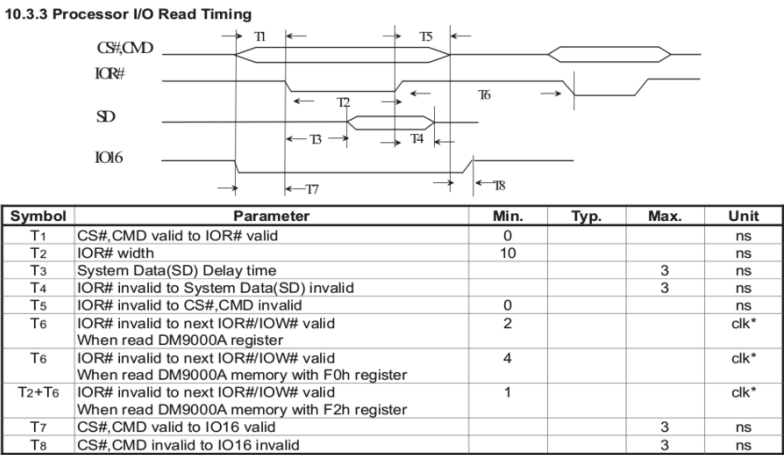


图 2.1 DM9000A I/O 读时序<sup>[5]</sup>

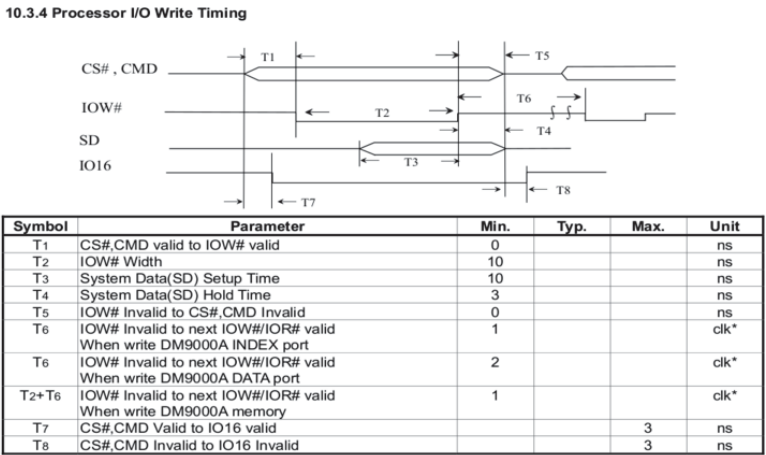


图 2.2 DM9000A I/O 写时序<sup>[5]</sup>

① <http://www.davicom.com.tw/userfile/24247/DM9000A-DS-F01-101906.pdf>

## 2.3 网络功能

### 2.3.1 网卡使用流程

DM9000A 使用手册<sup>[6]①</sup>给出了 DM9000A 芯片的使用方式，比较重要的有网卡芯片的初始化，以及通过网卡芯片接受、发送数据包。接下来会对软件端需要操作的流程进行介绍。

需要注意的是，这里的发送、接收数据包操作的是以太网帧。发送数据时，DM9000A 芯片会在使用者的设置下为以太网帧生成报头，写入的数据作为以太网帧的负载被传输；而接收数据时，读到的数据部分是包含了完整报头的以太网帧，因此需要软件端处理以太网帧的报头。

#### 2.3.1.1 读写寄存器

DM9000A 芯片在硬件的层次，对用户仅暴露两个寄存器，即地址 (DATA) 寄存器与数据 (INDEX) 寄存器，通过 CMD 引脚进行选择。其他寄存器不会直接向用户暴露，而要通过 DATA 与 INDEX 两个寄存器间接进行读写。需要对一个内部寄存器进行读或写的操作时，首先要将内部寄存器的地址写入 INDEX 寄存器，然后对 DATA 寄存器进行需要进行的读写操作，即可对内部寄存器进行读或写。具体的实现方式可以参考 3.3 中的具体实现。

对 DM9000A 的使用大多需要使用内部寄存器，接受、发送数据包时会有部分操作仅使用 DATA 寄存器。

除了 DM9000A 的寄存器之外，IEEE802.3 还定义了 32 个 PHY 寄存器，对于 PHY 寄存器的读写可以参考使用手册 5.4 节的说明。

#### 2.3.1.2 网卡初始化

使用网卡之前需要对网卡进行初始化，具体步骤可以参考使用手册 5.2 节的流程。初始化过程主要工作是对网卡芯片进行启动，重置，设置 MAC 地址，清除之前积累的状态、数据，使能中断信号、接受数据、校验和自动计算等功能。

其中可能需要修改的是设置 MAC 地址。由于 MAC 地址在数据链路层作为设备的标识信息使用，DM9000A 芯片不应与使用环境的其他设备有相同的 MAC 地址。

---

① <http://www.cs.columbia.edu/~sedwards/classes/2011/4840/Davicom-DM9000A-Application-Notes.pdf>

2.3.1.3 接受数据包

**接收策略** DM9000A 使用手册建议了两种接受数据包的策略<sup>[6]</sup>。

第一种策略是不断查询接收数据区，如果收到一个数据包就进行处理，否则继续查询。这种方法从实现 ethernet 功能的角度看比较简单，但不符合操作系统使用外设的常见方法，而且在应用程序使用时会有诸多不便。

第二种策略是利用 DM9000A 芯片提供的中断信号，通过操作系统的中断机制处理数据包的接收。在收到中断信号后，从网络芯片读取刚收到的数据包并进行处理，然后将 DM9000A 的 ISR 寄存器置为 1，消除中断信号。需要指出的是，第二种方案应当将数据缓冲区完全读取，并依次处理收到的所有数据包，否则可能导致一些数据包堆积在数据缓冲区但没有中断信号，不能及时处理。

**接收数据包简介** DM9000A 芯片从 RJ45 接口收到的数据会保存在接收缓冲区中，以供使用者在需要时读取。读取数据时，首先要对 MRCMDX 寄存器进行一次虚假读取，然后读取 MRCMDX1 寄存器，如果其低 8 位为 0x01，表示缓冲区有还未读取的数据。然后，可以读取 MRCMD 寄存器来读取缓冲区中的数据，此处可以连续读取，即只将 INDEX 寄存器写为 MRCMD 寄存器的地址一次，然后不断从 DATA 寄存器读取数据，直到读取一个数据包的数据。需要注意的是，DM9000A 芯片的寄存器是 16 位的，每次读取数据会读到两个字节。

缓冲区结构图如图 2.3 所示。

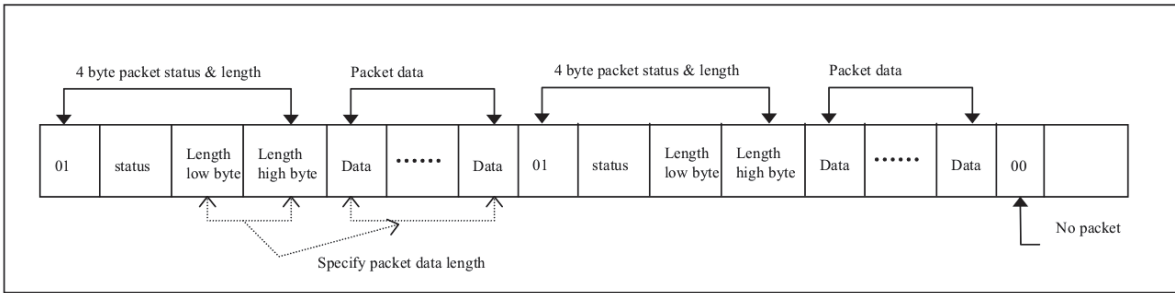


图 2.3 DM9000A 接收缓冲区<sup>[6]</sup>

2.3.1.4 发送数据包

发送数据包之前，首先对 MWCMDX 寄存器写入 0，以使网络芯片准备进行发送。然后，将 INDEX 寄存器写为 MWCMD 的地址，然后将以太帧的内容每次

两个字节地写入 DATA 寄存器。接下来，将以太帧长度写入 TXPLH 与 TXPLL 寄存器。最后，将 TCR 写为 1 即可发送数据。

### 2.3.2 网络协议栈

我们使用 TCP 来实现网络功能的数据传输，为此，我们实现了四个协议的简化版，包括 IP、ARP、ICMP、TCP。协议的选择主要基于让 TCP 可以工作的需要。ARP 用于 THINPAD 上的 ucore OS 与 PC 端的相互识别与 IP 确认，是 IP 工作的基础；网络层协议 IP 是 ICMP 与 TCP 的基础，通过 IP 地址决定数据的路由目的地；ICMP 用于 TCP/IP 中控制信息的传输，在 TCP 工作时一定会通过 ICMP 交换一些控制信息，因此必须实现；传输层协议 TCP 是我们实现数据传输的基础。

对这些协议的具体介绍，以及报头格式，可以参考任意网络方面的书籍或资料。本文内如无必要不会专门说明。

## 2.4 文件系统

ucore OS 使用的文件系统基于 Simple File System 实现。在这一节，我们会介绍 Simple File System 的使用方式、数据组织方式、数据存储格式。本节仅对与我们的工作关系较大的部分进行介绍，具体细节请参考 uCore OS 实验指导书<sup>[7]①</sup>。

### 2.4.1 Simple File System 总述

Simple File System 在 ucore OS 的文件系统中属于较底层的抽象层，包括了操作 Simple File System 的文件系统层与内存、磁盘中保存数据的 SFS 数据格式。在被用户直接调用时，要先经过文件系统抽象层，将不同具体文件系统抽象为统一的接口，因此，对 ucore OS 文件系统的功能拓展，需要在文件系统抽象层与文件系统层都加入支持。而这种支持的基础是 Simple File System 的数据格式。<sup>[7]</sup>

Simple File System 采用树形结构管理数据。首先，第 0 个块是一个超级块，保存了文件系统的总体参数，包括被用掉的块的数量、未被使用的块的数量、文件系统基本信息。然后，第 1 个数据块保存了根目录的索引节点。接下来，从第 2 个数据块开始，用一位代表一个数据块，记录了文件系统的空闲块映射 (freemap)

---

① <https://objectkuan.gitbooks.io/ucore-docs/content/>

情况。记录 freemap 的数据块之后，就是与其他目录、文件有关的数据块，可能是一个索引节点，也可能是一个纯粹的数据块。

### 2.4.2 索引节点

索引节点代表一个文件，记录了这个文件的基本信息，以及文件数据所在数据块的索引值。<sup>[7]</sup>

在磁盘上，索引节点的数据结构如代码 2.1 所示。

代码 2.1 磁盘上索引节点数据结构

```
struct sfs_disk_inode {
    uint32_t size;
    // 如果inode表示常规文件，则size是文件大小
    uint16_t type; // inode的文件类型
    uint16_t nlinks; // 此inode的硬链接数
    uint32_t blocks; // 此inode的数据块数的个数
    uint32_t direct[SFS_NDIRECT];
    // 此inode的直接数据块索引值（有SFS_NDIRECT个）
    uint32_t indirect;
    // 此inode的一级间接数据块索引值
};
```

被读取到内存中之后，索引节点的数据结构如代码 2.2 所示。

代码 2.2 内存中索引节点数据结构

```
struct sfs_inode {
    struct sfs_disk_inode *din; // on-disk inode
    uint32_t ino; // inode number
    uint32_t flags; // inode flags
    bool dirty; // true if inode modified
    int reclaim_count; // kill inode if it hits zero
    semaphore_t sem; // semaphore for din
    list_entry_t inode_link;
    // entry for linked-list in sfs_fs
    list_entry_t hash_link;
    // entry for hash linked-list in sfs_fs
};
```

### 2.4.3 文件数据

索引节点的索引值指向保存文件数据的数据块。

普通文件的数据块中以二进制的方式保存了文件数据，可以按照索引节点中的顺序、文件长度操作文件。

目录文件的数据块中保存的是目录下各个文件的入口数据，具体数据结构如代码 2.3 所示。

代码 2.3 文件入口数据结构

```
struct sfs_disk_entry {  
    uint32_t ino; //索引节点所占数据块索引值  
    char name[SFS_MAX_FNAME_LEN + 1]; //文件名  
};
```



## 第 3 章 网络功能

### 3.1 层次结构

在 THINPAD 与 ucore OS 组成的软硬件平台上设计、实现网络功能时，我们参考了 OSI 模型的层次与实现的复杂程度，将整个功能划分为四层，依次为硬件层，驱动层，基础协议以及 TCP。各功能层与 OSI 模型的层次以及软硬件平台的关系如图 3.1 所示

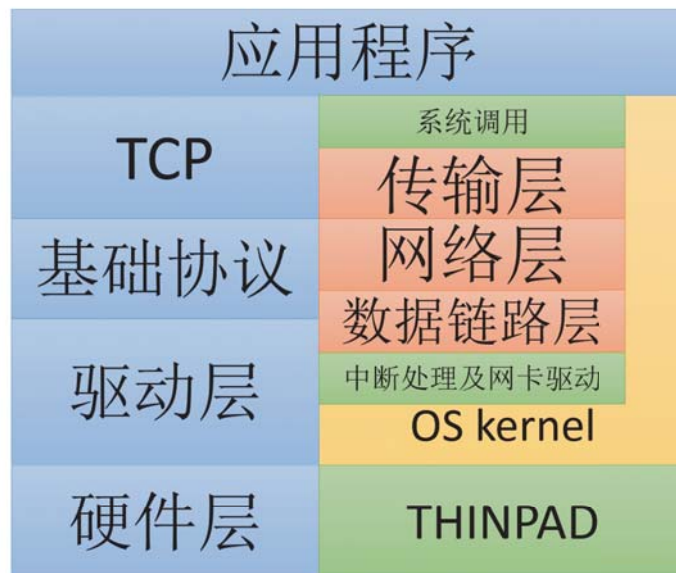


图 3.1 网络功能分层功能图

### 3.2 硬件层

硬件层实现了在 THINPAD 平台上，为了支持网络功能，为 CPU 增加的功能。在这一层，我们实现了 DM9000A 正常工作对硬件信号的要求，为更高层提供了网络中断信号，并且提供了对 DATA 和 INDEX 两个寄存器的读写操作。这一层从硬件意义上实现了 DM9000A 的启动与数据交互，对 DM9000A 的功能的进一步支持放在软件端实现。

正如 2.2.2 所述，THINPAD 上 FPGA 与 DM9000A 相连的引脚有 23 个，其



中部分引脚的输入已经说明，本节考虑的引脚有数据引脚、CMD 引脚、INT 引脚、IOR 引脚、IOW 引脚。

### 3.2.1 中断信号

INT 引脚用于 DM9000A 输出中断信号，输出高电平时表示存在中断信号。THINPAD 上已经支持了其他外设的中断信号，因此将 INT 信号直接放入中断向量即可。在我们的实现中，使用中断向量第 5 位表示网络中断。

### 3.2.2 数据读写

数据引脚、CMD 引脚、IOR 引脚、IOW 引脚用于实现对 DM9000A 读取、写入数据。首先考虑选择寄存器的问题。由于我们将对 DM9000A 内部寄存器的读写功能放在软件支持部分实现，硬件层仅仅需要实现对 DATA 和 INDEX 寄存器的读写，因此我们可以直接定义两个地址，0xBF003E0 表示 INDEX 寄存器，0xBF003E4 表示 DATA 寄存器。对寄存器的选择通过 MMU 实现，在对这两个地址进行访存时，根据地址决定 CMD 的输出。然后是读写数据的流程。我们使用的 CPU 在 MEM 阶段可以输出 busy 信号，使流水线的其他阶段暂停工作，以在不妨碍流水线工作流程的情况下实现超过一个时钟周期的访存。基于此，参考了 DM9000A 的数据手册<sup>[5]</sup>10.3.4 节与 10.3.5 节的时序要求，我们采用 busy 的机制处理对 DM9000A 处理器的数据读写。当 MMU 收到对 0xBF003E0 或者 0xBF003E4 的访存要求时，除了根据地址改变 CMD 的输出，还根据操作是读还是写来确定对 DM9000A 进行读取还是写入操作。读取的写入操作的流程类似，首先，向数据引脚输出信号，写入时输出需要写入的数据，读取时输出高阻；然后，根据操作的类型将 IOR 与 IOW 之一使能，并按照时序要求的时间保持若干个周期；接下来将 IOR 与 IOW 都改为不活跃，读取操作时读取数据引脚上的数据，再按照时序要求的时间保持若干个周期，以保证可以连续读写网卡芯片的寄存器；最后，消除 busy 信号，使 CPU 的流水线继续运行。

与之前有过的实现不同，这里我们按照时序要求让输出信号保持了足够的时间，使得软件端无需增加气泡。

### 3.3 驱动层

软件层在硬件层的基础上，实现了 DM9000A 芯片在逻辑意义上的初始化，并将数据包的发送与接收抽象成软件更容易操作的形式，使得更高层不需要关心 DM9000A 的具体细节。在这一层，我们实现了 DM9000A 芯片的初始化、发送数据、接收数据的功能，并且基于操作系统的中断处理机制对网络中断信号进行了处理。

#### 3.3.1 寄存器访问

我们参考了使用手册的第 5 章中的伪代码，对访问内部寄存器、访问 PHY 寄存器进行封装，以方便操作。访问内部寄存器的代码可以参考代码 3.1，访问 PHY 寄存器的代码较长，请参考使用手册 5.4 节。

代码 3.1 访问内部寄存器示例

```
unsigned int ethernet_read(unsigned int addr) {
    VPTR(ENET_IO_ADDR) = addr;
    return VPTR(ENET_DATA_ADDR);
}

void ethernet_write(unsigned int addr, unsigned int data) {
    VPTR(ENET_IO_ADDR) = addr;
    VPTR(ENET_DATA_ADDR) = data;
}
```

#### 3.3.2 网络芯片初始化

我们将网络芯片的初始化放在操作系统初始化过程的最后一步，以保证网络芯片需要用到的其他资源都已经就绪。正如 2.3.1.2 所介绍，使用手册的第 4 章与 5.2 节介绍了芯片初始化的流程，按照该流程实现即可，示例如代码 3.2 所示。其中，需要根据需要决定 MAC 地址，并填入中断号以使操作系统开始处理网络中断。

代码 3.2 网络芯片初始化示例

```
void ethernet_init() {
    wait_queue_init(eth_wait_queue);
    ethernet_powerup();
    ethernet_reset();
    ethernet_phy_reset();
    // auto nego
```

```

ethernet_phy_write ( 4, 0x01E1 | 0x0400 );
ethernet_phy_write ( 0, 0x1200 );
int i;
// set MAC address
for(i = 0 ; i < 6 ; i += 1)
    ethernet_write(DM9000_REG_PAR0 + i, MAC_ADDR[i]);
// initialize hash table
for(i = 0 ; i < 8 ; i += 1)
    ethernet_write(DM9000_REG_MAR0 + i, 0x00);
// accept broadcast
ethernet_write(DM9000_REG_MAR7, 0x80);
// enable pointer auto return function
ethernet_write(DM9000_REG_IMR, IMR_PAR);
// clear NSR status
ethernet_write(DM9000_REG_NSR,
    NSR_WAKEST | NSR_TX2END | NSR_TX1END);
// clear interrupt flag
ethernet_write(DM9000_REG_ISR,
    ISR_UDRUN | ISR_ROO | ISR_ROS | ISR_PT | ISR_PR);
// enable interrupt (recv only)
ethernet_write(DM9000_REG_IMR, IMR_PAR | IMR_PRI);
// enable reciever
ethernet_write(DM9000_REG_RCR,
    RCR_DIS_LONG | RCR_DIS_CRC | RCR_RXEN);
// enable checksum calc
ethernet_write(DM9000_REG_TCSCR, TCSCR_IPCSE);
pic_enable(ETH_IRQ);
}

```

### 3.3.3 接收数据

如 2.3.1.3 所介绍，使用手册 5.6 节给出了接收数据的流程，按照该流程即可实现基本的接收数据功能。但由于我们采用中断的机制来接收数据，需要对此进行改动。具体内容见 3.7。

这一层对以太帧的处理主要是检查以太帧的类型，根据类型将负载交给更高层对应的协议处理即可。目前我们支持的以太帧的负载类型有 IP 与 ARP 两种，都在更高层实现。

示例代码如代码 3.3 所示。

代码 3.3 数据接收示例

```

void ethernet_recv() {
    int start = 0;
    int ethernet_rx_len = 0;
    int i;

```

```

while (ethernet_rx_len >= 0) {
    // a dummy read
    ethernet_read(DM9000_REG_MRCMDX);
    // select reg
    VPTR(ENET_IO_ADDR) = DM9000_REG_MRCMDX1;
    int status = LSB(VPTR(ENET_DATA_ADDR));
    if(status != 0x01){
        ethernet_rx_len = -1;
        ethernet_rx_data[start] = ethernet_rx_len;
        break;
    }
    VPTR(ENET_IO_ADDR) = DM9000_REG_MRCMD;
    status = MSB(VPTR(ENET_DATA_ADDR));
    ethernet_rx_len = VPTR(ENET_DATA_ADDR);
    if(status & (RSR_LCS | RSR_RWTO | RSR_PLE |
                RSR_AE | RSR_CE | RSR_FOE)) {
        ethernet_rx_len = -1;
    }
    ethernet_rx_data[start] = ethernet_rx_len;
    start++;
    for(i = 0 ; i < ethernet_rx_len ; i += 2) {
        int data = VPTR(ENET_DATA_ADDR);
        ethernet_rx_data[start+i] = LSB(data);
        ethernet_rx_data[start+i+1] = MSB(data);
    }
    start += ethernet_rx_len;
}
// clear interrupt
ethernet_write(DM9000_REG_ISR, ISR_PR);
}

```

### 3.3.4 发送数据

这一层在数据发送的过程中，起到的作用是接收更高层的数据，然后通过硬件层发送以更高层的数据为负载的以太帧。

如2.3.1.4所介绍，使用手册 5.5 节给出了发送数据的流程，按照该流程即可实现发送数据功能。由于需要写入 DM9000A 芯片的是整个以太帧，这一层在发送数据前需要填写以太帧的报头，即目的 MAC 地址、源 MAC 地址、负载协议类型。其中目的 MAC 地址为 PC 机的 MAC 地址，源 MAC 地址为自行指定的 THINPAD 实验平台的 MAC 地址。

示例代码如代码 3.4 所示。

代码 3.4 数据发送示例

```

void ethernet_send() {
    // int is char
    // A dummy write
    ethernet_write(DM9000_REG_MWCMDX, 0);
    // select reg
    VPTR(ENET_IO_ADDR) = DM9000_REG_MWCMD;
    int i;
    for(i = 0 ; i < ethernet_tx_len ; i += 2){
        int val = ethernet_tx_data[i];
        if(i + 1 != ethernet_tx_len)
            val |= (ethernet_tx_data[i+1] << 8);
        VPTR(ENET_DATA_ADDR) = val;
    }
    // write length
    ethernet_write(DM9000_REG_TXPLH, MSB(ethernet_tx_len));
    ethernet_write(DM9000_REG_TXPLL, LSB(ethernet_tx_len));
    // clear interrupt flag
    ethernet_write(DM9000_REG_ISR, ISR_PT);
    // transfer data
    ethernet_write(DM9000_REG_TCR, TCR_TXREQ);
}

```

### 3.4 基础协议

我们支持的协议有 ARP、IP、ICMP、TCP。其中前三个协议相对比较简单，而且在 OSI 模型中位置接近，因此合在一起讨论，而将 TCP 单独讨论。需要指出的是，我们仅仅实现了协议的简化版，以满足 TCP 传输数据的基本要求，因此对于协议的功能并没有完全实现。ARP 与 ICMP 的存在是为了应对陌生的网络环境，因此我们仅仅实现了对它们的响应，并不主动发出这两个协议的数据包；IP 与 TCP 都需要实现接收与发送。对接收数据来说，虽然在驱动层需要考虑接收以太帧的策略，但对本层的三个协议与 TCP 而言，只需要对一段数据包进行处理，不需关心接收以太帧的策略；对发送数据来说，各个协议要实现的是实现本协议的数据包，然后将数据包和协议类型交给较低层，以进行进一步的处理。

#### 3.4.1 ARP

ARP, 即地址解析协议，用于根据 IP 查询 MAC 地址，从而使 IP 可以在以太网上使用。

由于 PC 端需要通过 ARP 识别 THINPAD 实验平台，我们实现了对 ARP 请求的响应。在 PC 端，我们通过配置路由表，将实验平台所在 IP 端的路由项设置为连接 PC 与实验平台的有线网络，这样当需要判断实验平台的 IP 地址对应的 MAC 地址时，PC 会发出一个 ARP 请求。在 ucore OS 中，我们针对这个 ARP 请求进行响应，返回 THINPAD 实验平台的 MAC 地址。使得 PC 端可以基于 IP 与实验平台进行通信。

### 3.4.2 IP

IP，即互联网协议，是网络层的核心协议，使得网络中的各个主机可以通过 IP 地址进行相互通信。我们实现了 IPv4 的基本功能，以支持上层协议的使用。

在处理接收的数据时，IP 要做的是检查协议类型与负载长度，调用对应协议的处理操作。在发送数据时，IP 要做的是填写 IP 报头的各项，然后将其他协议的数据包作为负载，生成 IP 数据包，交给较低层发送。

### 3.4.3 ICMP

ICMP，即互联网控制消息协议，用于 TCP/IP 网络中发送控制消息。

PC 端需要使用 ICMP 确认与 THINPAD 实验平台之间的网络畅通，因此在我们的实现中，我们实现了对 ECHO 的响应，即在收到 ECHO 请求时，生成 ECHO 响应。ICMP 的 ECHO 响应较为简单，需要填写的内容以报头为主，填充数据用于保持最终发出的以太网帧不会过于小。

## 3.5 TCP

TCP，即传输控制协议，是我们本次工作的重点。我们实现的 TCP 功能向应用程序提供的 API 参考了 POSIX socket API，根据使用环境进行了一定的简化。API 的参数及功能描述见表 3.1。

在操作系统内，我们按照 RFC793 规定的 TCP 的状态转移定义与标记定义<sup>[8]</sup>，在中断处理与 API 调用的函数中实现了 TCP 的状态变化与数据收发。在接收与发送数据时，我们使用了缓冲区，以实现发送不会阻塞与接收不会漏包，并且有利于重传的实现；采用 Stop-and-wait 策略，即停止等待协议，来管理数据流以及 ACK、SEQ。另外，我们还实现了多 TCP 机制，使得多个 TCP 可以同时工作，而不会互相影响数据流。

表 3.1 TCP API 简述

API 及参数	功能描述
<code>int socket();</code>	获取 socket 文件描述符；
<code>int bind(int sockfd, int *ip, int port);</code>	为 socket 文件描述符绑定目标 IP 地址与本地端口，用于之后对这个 socket 文件描述符进行监听；
<code>int connect(int sockfd, int* ip, int port);</code>	为 socket 文件描述符绑定目标 IP 地址与目标端口，并且主动发起 TCP 连接。这个函数会导致进程被挂起，直到受到对应的连接建立确认包之后才会唤醒进程；
<code>int listen(int sockfd);</code>	监听一个已经使用 bind 绑定过 IP 地址与端口的 socket 文件描述符，接收 TCP 连接请求。这个函数会导致进程被挂起，收到符合要求的 TCP 连接请求之后会唤醒进程；
<code>int send(int sockfd, char* data, int len);</code>	通过一个 socket 文件描述符发送数据。这个函数利用发送缓冲区进行数据的发送，因此是非阻塞的。如果缓冲区大小不足，send 函数可能发回失败；
<code>int recv(int sockfd, char* data, int len);</code>	通过一个 socket 文件描述符接收数据。这个函数会检查接收缓冲区，如果已经有足够的数据，会直接返回收到的数据，否则会挂起进程，直到收到足够的数据才会唤醒进程；
<code>int tcp_close(int sockfd);</code>	关闭一个 socket 文件描述符对应的 TCP 连接；

### 3.6 网络功能工作流程

图 3.2 以比较有代表性的 send、recv 两个系统调用的为例，介绍了接收、发送数据时网络功能的工作流程以及各层之间的调用关系。



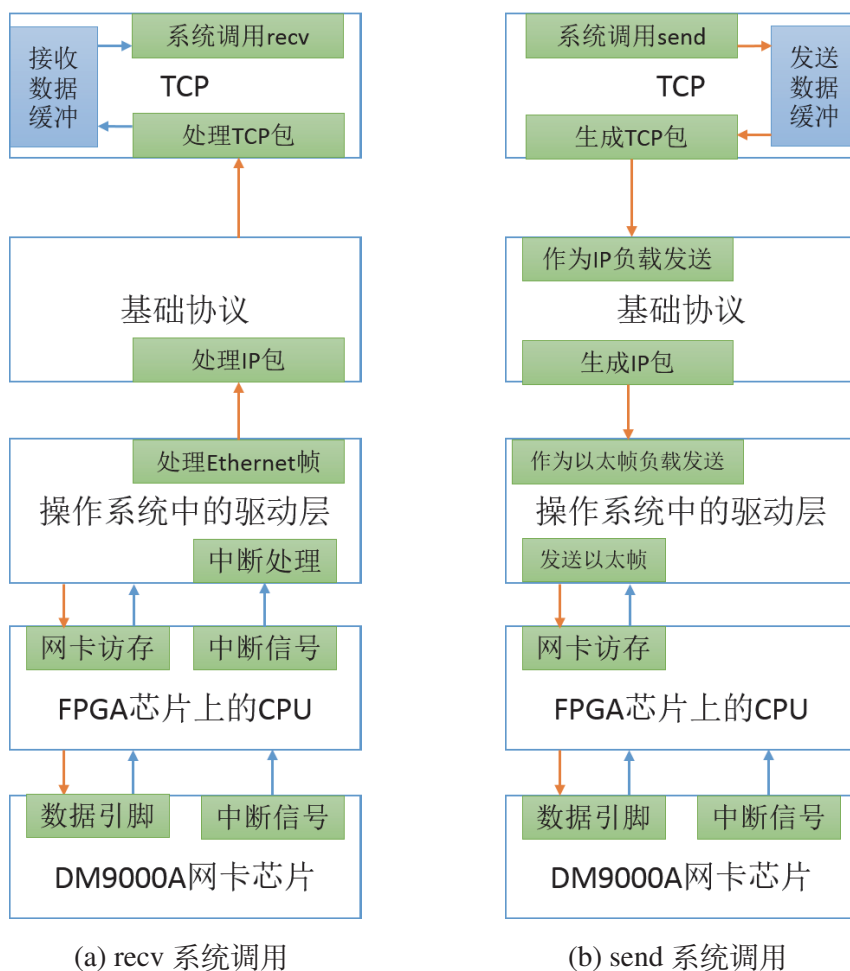


图 3.2 `recv`、`send` 系统调用的工作流程。橙色线表示指令流，即携带数据的调用；蓝线表示单纯的数据流。

### 3.7 网络功能实现细节

这一节将会对网络功能的实现中一些细节进行介绍，以给出实现中需要注意的问题。

#### 3.7.1 多数据包处理

对于轮询的数据包接收策略来说，数据包的接收按照说明手册的流程即可，每次只需要接收一个数据包。但对于基于中断的数据包接收策略而言，正如2.2.3介绍的，如果每次中断仅处理一个数据包，当多个数据包在较短时间间隔内到来时，可能处理完第一个数据包时，缓冲区还有剩余数据。这时，如果消除中断信号，缓冲区内剩余的数据不会触发新的中断信号，因此要到再收到一



个数据包时才能处理之前缓冲区中剩余的数据包，而新到达的数据包又要等待更为靠后的数据包来触发中断才能得到处理。这样，对数据包的处理就会严重延后，甚至会因为堆积过多未被处理的数据而无法接收新数据。尽管由于实际情况中远端主机会定期发送 ARP 与 ICMP 的数据包，表现出来的只是常常出现超时，但依然不利于网络功能的工作。因此，基于中断的数据接收策略，在一次中断处理中，应该将缓冲区中的所有数据包都读取出来，并进行处理。

另外，对数据的处理过程中可能还会发出新的数据，或者触发其他较为复杂的操作。为了避免相互干扰，我们先把缓冲区的数据按照以太网帧逐个读出，直到不再有数据包为止；再依次处理各个以太网帧。

### 3.7.2 TCP 状态转移

根据 RFC793 的规定，TCP 中存在 CLOSED、SYN\_RECVED、SYN\_SENT、LISTEN、ESTAB、FINWAIT、CLOSEWAIT 等状态。其中 CLOSED 为初始状态，主动连接后进入 SYN\_SENT，监听后进入 LISTEN，收到连接请求后进入 SYN\_RECVED。完成三次握手之后，状态进入 ESTAB，可以进行收发数据。在 LISTEN 状态下，主动断开连接会进入 FINWAIT，被动断开连接进入 CLOSEWAIT，最终完成四次握手，断开连接，socket 文件描述符回到 CLOSED 状态。

TCP 的状态转移主要在中断处理以及 API 的 connect、listen、close 中出现。

### 3.7.3 连接建立

TCP 建立连接通过三次握手的方式完成，分为使用 connect 的主动连接与使用 listen 的被动连接。建立连接的过程中需要注意的是状态码的计算与 IP、端口号的检查，会分别在 3.7.6 与 3.7.7 详加介绍。

### 3.7.4 接收数据

recv API 在执行时，检查缓冲区内已有数据长度，如果大于等于 recv 读取的长度，则直接读取这些数据并返回；否则，记录 recv 需要读取的数据长度，挂起进程，等待数据的到来。

中断处理时，如果连接处于 LISTEN 状态，而且收到的 TCP 包负载长度不为 0，而且标志位、状态码符合要求，可以认为这是一个合法的数据传输包。对于一个数据传输包，检查接收缓冲区是否足够保存这些数据，如果不够，则不需

处理，PC 端会尝试将数据量减少之后重传。如果足够放下，则将这个包内的负载读入缓冲区。如果此时有正在等待数据的 `recv` 操作，还会检查读入之后缓冲区的数据长度是否满足 `recv` 需要读取的长度，如果满足，则唤醒进程，让 `recv` 继续运行。

在我们的实现中，接收缓冲区长度为 1000 字节，通过循环使用 `index` 来避免频繁复制移动数据。对标志位与状态码的处理在 3.7.6 讨论。

### 3.7.5 发送数据

`send` API 在执行时，检查发送缓冲区剩余空间是否足够容纳本次 `send` 的数据长度，如果不满足则返回失败。否则，将数据复制到缓冲区中。如果此时不是在等待发送确认，可以将缓冲区内的数据都发送出去。

发送 TCP 包时，首先将需要发送的数据复制到负载的位置；然后填写 TCP 报头，其中比较重要的是标志位与状态码的填写；接下来记录这个包的长度信息，等待确认包；最后将 TCP 包交给 IP 发送。

发送一个 TCP 包之后，如果中断处理中收到含有 `ACK` 而且状态码符合要求的数据包，则可以确认发送已经被收到，可以调整状态码并取消等待确认包的状态。此时，如果缓冲区内还有数据，可以再次发送。

在等待确认包的过程中，利用时钟中断进行计时，从而可以在超时的时候进行重传。发送缓冲区可以为重传提供数据，使得 `send` 不需要阻塞。

### 3.7.6 标志位及状态码

#### 3.7.6.1 控制标志位

控制标志位，即 `control flag`，指 TCP 报头的 6 个比特，包括 `URG`、`ACK`、`PSH`、`RST`、`SYN`、`FIN`。

`URG` 表示紧急处理，`RST` 表示重置连接。我们的实现中没有支持这两个标志位。

`PSH` 表示有数据传输，在发送含有负载的数据，也就是说 `send` 处理的数据时，需要将这位置为 1。收到含有 `PSH` 位的数据包时，也表示这个数据包包含了需要传输的数据，但由于此时负载长度会大于 0，也可以不使用这个位判断。

`ACK` 位表示这个 TCP 包包含有效的 `ACK` 信息，这常用于表示确认信息，比如建立连接过程中的确认与传输数据中的确认。

SYN、FIN 分别用于建立连接和断开连接，前者仅在建立连接的过程中使用；后者会在断开连接的过程中使用，同时，如果在 ESTAB 状态下收到，也会进入断开连接的流程。

需要注意的是，这些标志位并不是互相排斥的。比如，在 ESTAB 状态下，可能收到同时包含 PSH、ACK、RST、FIN 中多个标志位的数据包，此时应该依次处理各个标志位，而不是仅处理其中一个。PC 端有的时候会将可以合并的数据包合并，比较常见的是把刚收到的传输数据的 ACK 包与需要发送的传输数据的包（包含 PSH）合并成一个包发送。

### 3.7.6.2 状态码

TCP 报文中有 SEQ 与 ACK 两种状态码。SEQ 表示序列号，用于表示发送者的状态，ACK 表示确认号，主要是对接收者的状态的回应。

**建立连接** 在建立连接时，发起连接的一方，本段称为 A，应当提供一个随机的 SEQ；收到连接请求的一方，本段称为 B，需要产生另一个随机的 SEQ，还应记录收到的 SEQ，并将收到的 SEQ 加 1 之后作为 ACK 返回；A 收到回应时，检查 ACK 是否符合最初发出的 SEQ，如果符合则可以进入 ESTAB 状态，然后 A 记录收到的 SEQ，将收到的 SEQ 加 1 之后作为 ACK 返回；B 收到回应时，检查 ACK 是否符合最初发出的 SEQ，如果符合则可以进入 ESTAB 状态。

以上说明仅仅涉及了状态码的设置，实际实现时还要考虑控制状态位的设置。

**传输数据** 本段称传输数据的发送方为 A，称接收方为 B。在整个 TCP 连接中，两方都需要记录自己与对方的 SEQ，以作为连接中的数据流验证与身份验证，保证数据流的完整性并保证收到的数据包可靠。传输数据时，A 首先将数据流的数据作为负载发送，除了设置 PSH 位，还需要设置 SEQ。B 收到数据时，将收到的 SEQ 与记录的 SEQ 进行比较，确认二者相同，然后将 SEQ 加上数据的长度，作为 ACK 返回，并修改本地的 SEQ 记录。A 收到确认信息时，检查 ACK，如果等于自己的 SEQ 加上之前发送的数据包的长度的话，则确认该数据包已经被收到，可以更新自己的 SEQ，并完成本次发送，视缓冲区决定是否需要进行新一轮的发送；如果 ACK 与 SEQ 不匹配，则不接受这个确认包，此时比较可能的结果是重传。

连接的两端都要发送数据时，两个方向的数据传输用到的标志位和状态码相互不影响，因此两个方向的数据传输实际上是可以同时进行的。对此无需做单独的处理。

**断开连接** 断开连接的流程与建立连接类似，不同之处在于，SEQ 需要符合之前的连接中规定的数值，而非随机选定；使用的标志位是 FIN 而非 SYN；对 FIN 的响应与发出的 FIN 应当分开发送，而不是像建立连接时那样在同一个包内发送。

### 3.7.7 多 TCP 连接支持

表示一个 TCP 连接，需要用到的信息包括一系列表示状态的信息，以及缓冲区。我们使用如代码 3.5 所示数据结构记录一个 TCP 连接，使用数组记录多个 TCP 连接的状态，使用 socket 文件标识符对其进行区分。

代码 3.5 TCP 连接记录示例

```
struct tcp_queue_s{
    char recv_buffer[buf_length];
    char send_buffer[buf_length];
    int send_pos, send_start, send_len, send_waiting;
    int recv_pos, recv_start, recv_len;
    int recv_len_target, recv_waiting;
    int tcp_src_port, tcp_dst_port, tcp_target_port;
    int tcp_src_addr[4], tcp_dst_addr[4];
    int tcp_ack, tcp_seq;
    int tcp_my_seq, tcp_remote_seq;
    int tcp_state;
};
```

主动调用 API 进行的操作，由于需要提供文件标识符，可以直接确定使用的是哪个 TCP 连接。

中断处理中，需要根据 IP 和端口号来确定具体是哪个 TCP 连接。对于所有连接，一定可以确定这个连接的本地 IP 与端口号，可以基于此进行检查。对于处于 LISTEN 状态的连接，由于还在等待远程主机连接，无法确定远程的 IP 与端口号；而对于其他情况而言，远程的 IP 与端口号已经确定，因此还需要检查远程 IP 与端口号。对于满足了 IP 和端口号检查的数据包，可以认为属于对应的 TCP 连接，可以进一步处理。

## 第 4 章 文件系统功能拓展

ucore OS 目前支持打开、读取、写入文件的功能。启动系统之前，要在 PC 端生成文件系统的镜像，将需要用到的文件都放入镜像，启动系统之后，不支持新建文件。本章主要介绍我们对 ucore OS 文件系统的新建文件功能的拓展，着重对与新建文件功能拓展有关的相关知识与实现方式进行介绍。

### 4.1 ucore OS 的文件系统现状

从实现的角度看，ucore OS 将文件系统分为四层，包括通用文件系统访问接口层、文件系统抽象层、Simple FS 文件系统层以及外设接口层<sup>[7]</sup>，结构图见图 4.1<sup>①</sup>。



图 4.1 ucore 文件系统架构<sup>[7]</sup>

通用文件系统访问接口层向其他内核程序、应用程序提供文件操作 API，包括 open、close、read、write 等。主要内容是在用户态程序内通过系统调用使用

① [https://objectkuan.gitbooks.io/ucore-docs/content/lab8/lab8\\_3\\_1\\_ucore\\_fs\\_introduction.html](https://objectkuan.gitbooks.io/ucore-docs/content/lab8/lab8_3_1_ucore_fs_introduction.html)



内核态的文件操作。新建文件的方式是在调用 `open` 时增加新建文件的参数，这部分在目前已有的 `ucore OS` 文件系统实现中已经实现，只是在较低层次没有进行支持。

外设接口层包括对内存、磁盘、串口等外设进行访问，用于文件系统与其他外设进行交互。目前已经实现。

## 4.2 文件系统抽象层

VFS，即文件系统抽象层，对 SFS 的操作进行了抽象，向上提供抽象的文件系统操作接口，这一层对上提供的接口与通用文件系统访问接口层非常相似。这一层主要是进行一些拷贝数据、屏蔽中断等准备工作以及根据控制参数选择对应的文件系统操作操作，使得 Simple FS 文件系统层可以专注于更加具体的操作。与新建文件有关的是 `open` 函数，如果需要新建文件，会调用 Simple FS 文件系统层的接口新建文件。此处在目前已有的 `ucore OS` 文件系统实现中已经实现，但调用的 Simple FS 文件系统层接口实际没有实现。

在我们的实现中，对这里进行了修改，以使低层操作更为简单。在 `vfs_open` 设置了新建文件参数之后，会检查要打开的文件是否存在。如果不存在，则会调用 SFS 层的接口新建文件。这样，SFS 层的新建文件接口仅用于创建新文件，在创建文件之后，还需要在 VFS 层再次打开文件。创建新文件的具体操作交给 SFS 层实现。

## 4.3 Simple FS 文件系统层

Simple FS 文件系统层，实现了一个具体的文件系统，Simple File System，向上按照抽象层的接口要求实现接口，向下调用对内存、磁盘的访存。这一层需要基于 Simple File System 的组织方式、数据结构实现，因此实现中需要按照 2.4 介绍的内容组织代码。

这里与新建文件的有关的是目录文件的 `vop_create` 函数，在目前已有的 `ucore OS` 文件系统实现中没有实现，也是我们拓展新建文件功能时实现的内容。SFS 层对上提供目录文件的 `vop_create` 函数。这个函数属于 `sfs_node_dirops` 的函数表，在表内输入实际调用的函数。具体调用的函数是 `sfs_create`。

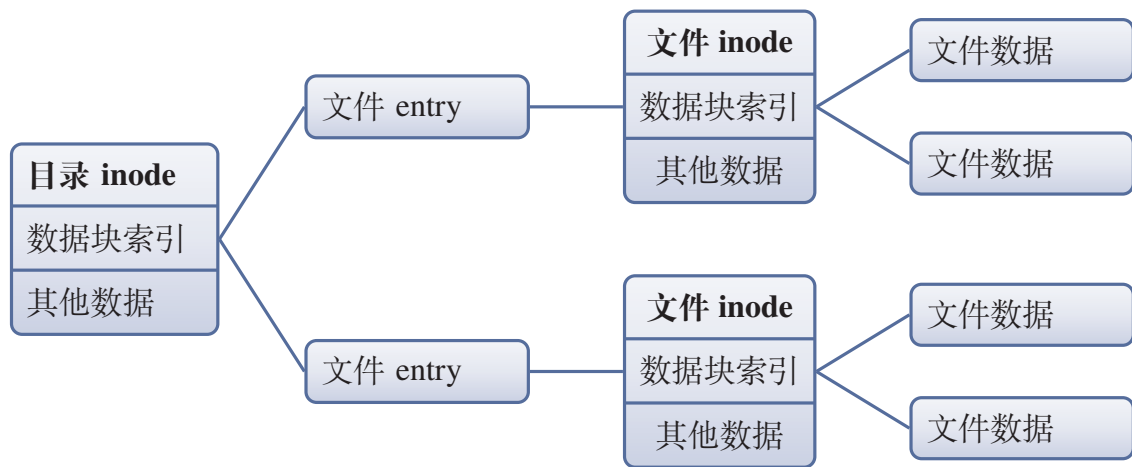


图 4.2 SFS 组织方式

Simple File System 的组织方式如图 4.2 所示。一个新的文件需要在目录中增加新文件的 entry，即文件的入口信息，然后还需要实现新文件的索引节点。由于新文件是一个空文件，此处无需增加保存新文件数据的数据块。

新建文件的流程如图 4.3 所示，具体细节见 4.4.1。

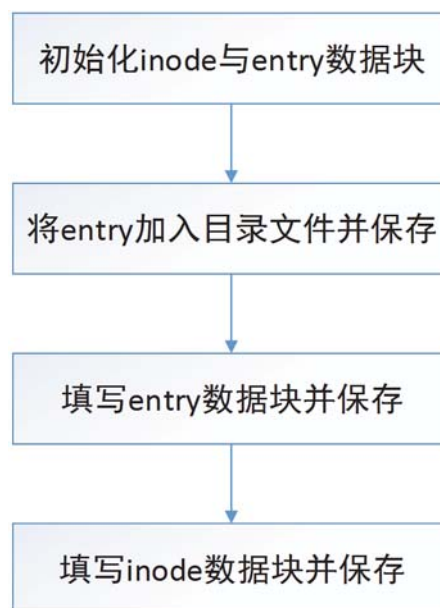


图 4.3 新建文件流程

## 4.4 具体实现

### 4.4.1 新建文件流程

首先需要获取两个 `inode` 号, 以获得两个空闲的数据块, 一个用于保存新文件的 `entry`, 另一个用于保存新文件的索引节点。

增加新文件的 `entry` 时, 首先要在目录的索引节点中将数据块数加 1, 并将新文件的 `entry` 加入目录索引节点的数据块索引列表; 然后要生成新文件的 `entry`, 使其索引节点指向新创建的另一个数据块, 并填写文件名。`entry` 的数据结构如代码 2.3 所示。

创建好新文件的 `entry` 后, 还需要生成新文件的索引节点。这个新的索引节点没有指向其他新的数据块。索引节点的数据结构如代码 2.1 所示。

将两个数据块都写好后, 还需要将修改后的信息写回磁盘, 以保存修改。

经过这些操作后, 我们就创建了一个新的文件。接下来在 VFS 层再次打开这个文件, 以确认生成文件成功。

### 4.4.2 功能函数

实现新建文件功能的过程中, 需要对 SFS 进行大量修改。在这方面, 我们用到了 SFS 的本身的一些功能函数。

`fsop_info` 与 `vop_info` 用于获取目录与文件系统的序号, 以进行后续操作。

`sfs_block_alloc` 用于从文件系统中获取一个空闲的数据块, 用于保存新文件的索引节点。

`sfs_bmap_load_nolock` 本身用于从索引节点中读取该索引节点的数据块中指定序号对应的数据块的 `inode`, 与此同时, 如果给的序号等于索引节点中保存的数据块数量, 会创建一个新的数据块。这个新创建的数据块用于保存新文件的 `entry`。

`sfs_wbuf` 可以用来向指定序号的数据块写入数据, 用来在生成新文件的 `entry` 和索引节点之后, 将两个数据块保存。目前的 THINPAD 与 ucore OS 组成的软硬件平台采用缓冲区与磁盘进行交互, 因此不会立刻将数据写回磁盘, 但读取文件也会优先从缓冲区读取, 从使用的角度看没有区别。另外, 写磁盘在后台执行, 需要一定时间, 此段时间内读取数据可能出错。

新文件的索引节点内容填写没有对应的功能函数可用, 需要我们按照 Simple



File System 的数据格式填写。

## 第 5 章 实验结果

### 5.1 功能测试

我们在 THINPAD 与 ucore OS 组成的软硬件平台上实现了基于 TCP 的网络功能，并为文件系统拓展了新建文件功能。为了对这些功能进行验证，我们进行了三个测试。

第一、二个测试用于测试网络功能的基本功能与文件系统的拓展。首先，实验平台端向 PC 端发送一个文件名。然后 PC 端在本地查找这个文件名，如果不存在则返回 0，如果存在则返回文件长度。接下来，PC 端按照每 500 字节为一组，向实验平台端发送文件内容。实验平台一端首先新建一个文件，然后接收数据，并将其写入本地文件。由此实现文件传输的功能。

第一个测试中传输的是一个纯文本文件，文件大小为 1.6Kb；第二个测试中传输的是一个使用 32 位 mips-gcc 编译的二进制文件，文件大小为 74Kb，运行结果为输出 1000 以内的所有质数。

第三个测试主要用于全面测试网络功能，主要加入了对多 TCP 连接的测试。我们让 PC 与实验平台建立 A、B 两个 TCP 连接，通过网络从 PC 向实验平台传输一段文本，其中 PC 端先通过 A 发送文本的前半段，再通过 B 发送文本的后半段；而 ucore OS 一端先通过 B 接收一段数据，再通过 A 接收一段数据，并将第二段数据放在第一段的之前。接下来，在实验平台上将这段文本进行一定修改，然后先通过 B 发送后半段，再通过 A 发送前半段。此时 PC 端按照先 A 后 B 的顺序接收数据，并将收到的数据显示出来。最后，实验平台上的程序将处理后的文本数据保存在本地，以验证结果。

本次测试中发送的是 a 到 z 的 26 种循环移位表示，每种表示后加一个换行符，因此共发送 702 个字符。实验平台收到数据后，将每 27 个字符的前 26 个，也就是换行符之外的内容，颠倒顺序，产生新的数据。

在第一、二个测试中，由 PC 端主动建立连接，实验平台被动监听；在第三个测试中，由实验平台主动建立连接，PC 端被动监听。

5.2 测试结果

5.2.1 文件传输测试

在传输文件的测试中，经过约 1.5 秒，文件被传输到实验平台上。查看文件内容，可以看到实验平台上保存的文件与测试程序发出的文件内容相同。结果如图 5.1 所示。

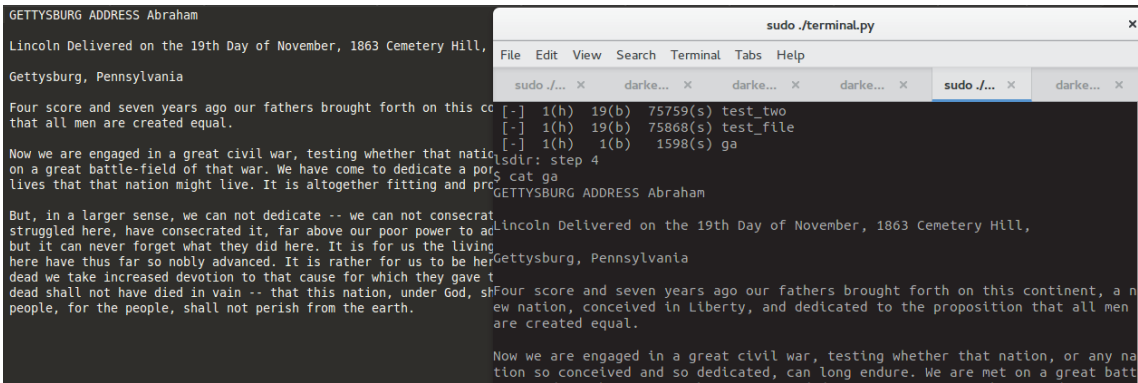
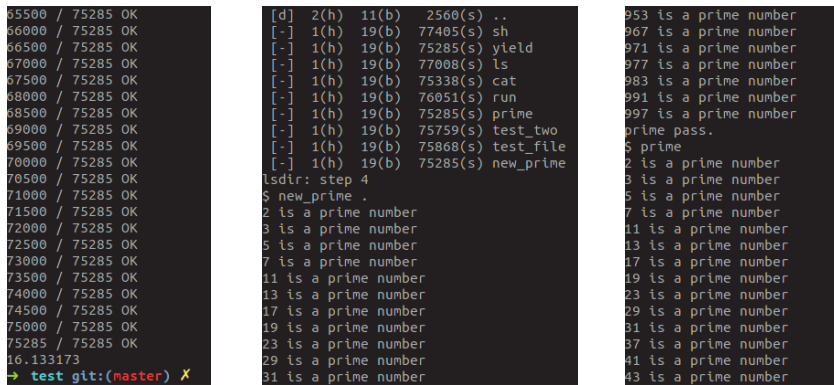


图 5.1 文本文件传输结果与原文件

在传输二进制文件的测试中，经过约 16 秒，文件被传输到实验平台上，平均数据传输速度为约 4.6Kb/s。在制作文件系统的镜像时，已经将这个二进制文件加入镜像中。如图 5.2 所示，传入的新文件 new\_prime 与原来的 prime 大小相同，执行结果也相同。



(a) 电脑端显示结果

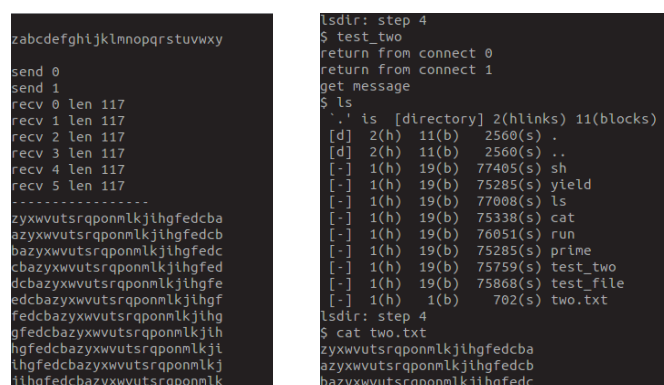
(b) 新传入文件运行结果

(c) 初始化镜像时加入的文件运行结果

图 5.2 二进制文件传输结果与运行结果

## 5.2.2 多 TCP 连接测试结果

多 TCP 连接同时运行的测试结果如图 5.3 所示。可以看到，测试的返回结果与预期相同，各行是从 z 到 a 这个字符串的循环移位。而且，在实验平台上也成功创建了文件，该文件内容与 PC 端收到的内容相同，说明多 TCP 连接正常工作，各个连接之间数据流独立。



```
zabcdefghijklmnopqrstuvwxyz
send 0
send 1
recv 0 len 117
recv 1 len 117
recv 2 len 117
recv 3 len 117
recv 4 len 117
recv 5 len 117
-----
zyxwvutsrqponmlkjihgfedcba
azyxwvutsrqponmlkjihgfedcb
bazyxwvutsrqponmlkjihgfedc
cbazyxwvutsrqponmlkjihgfed
dcbazyxwvutsrqponmlkjihgfe
edcbazyxwvutsrqponmlkjihgf
fedcbazyxwvutsrqponmlkjihg
gfedcbazyxwvutsrqponmlkjih
hgfedcbazyxwvutsrqponmlkji
ihgfedcbazyxwvutsrqponmlkj
jihgfedcbazyxwvutsrqponmlk
```

```
lsdir: step 4
$ test_two
return from connect 0
return from connect 1
get message
$ ls
'.' is [directory] 2(hlinks) 11(blocks) 2
[d] 2(h) 11(b) 2560(s) .
[d] 2(h) 11(b) 2560(s) ..
[-] 1(h) 19(b) 77405(s) sh
[-] 1(h) 19(b) 75285(s) yield
[-] 1(h) 19(b) 77008(s) ls
[-] 1(h) 19(b) 75338(s) cat
[-] 1(h) 19(b) 76051(s) run
[-] 1(h) 19(b) 75285(s) prime
[-] 1(h) 19(b) 75759(s) test_two
[-] 1(h) 19(b) 75868(s) test_file
[-] 1(h) 1(b) 702(s) two.txt
lsdir: step 4
$ cat two.txt
zyxwvutsrqponmlkjihgfedcba
azyxwvutsrqponmlkjihgfedcb
bazyxwvutsrqponmlkjihgfedc
```

(a) 电脑端显示结果      (b) 实验平台端显示结果

图 5.3 多 TCP 连接测试结果

## 第 6 章 结论

### 6.1 内容总结

在 THINPAD 与 ucore OS 组成的实验平台上，我们实现了以 TCP 为核心的网络功能，并且拓展了文件系统的新建文件的功能。

在网络功能方面，我们在软硬件两方面结合，实现了对 THINPAD 上的 DM9000A 的驱动，在操作系统内核中实现了 TCP/IP 协议栈，并通过 POSIX socket API 向用户态程序提供网络连接的相关功能。我们实现的 TCP/IP 协议栈，对数据的接收和发送使用缓冲区缓冲数据，提供了无阻塞的数据发送，并使基于中断的数据接收不会导致数据丢失；另外，还增加了基于 socket 文件描述符的多 TCP 连接支持，大大方便了 socket API 的使用。

在文件系统方面，我们根据 Simple File System 的组织结构与数据结构，完成了新建文件功能的拓展，使 ucore OS 的功能更加完善。并且，通过将本次的工作结合起来，实现了基于网络的文件传输。

### 6.2 不足与拓展

我们的工作，尤其是网络功能方面的工作，对软硬件平台都有较高的耦合性，还有很多值得改善的地方。比如四层的网络功能实现中，驱动层既与操作系统有密切的关系，又包含大量对硬件进行支持的代码，如果未来希望将 ucore OS 对网络功能的支持拓展到其他平台，这一层可以进行拆分，将对硬件的软件驱动与操作系统中处理以太帧的部分分为两层，后者及后者之上的层次可以跨平台使用，而驱动层及以下的部分依赖于具体硬件。另外，目前的网络功能对网络环境有过多假设，局限于实验平台与 PC 通过网线直接连接的情况，TCP/IP 协议栈中许多协议的实现也有所简化，可以对这部分内容进行充实，以支持更为复杂的网络环境。

另外，在我们的工作的基础上，还可以对 THINPAD 与 ucore OS 组成的实验平台进行其他方面的拓展。比如网络功能目前可以提供约 4.6Kb/s 的数据传输，这个速度要是目前将文件系统写入 THINPAD 的方案，串口传输，的三倍，而

且如果网络功能以一个独立的程序而非操作系统的一部分存在，速度还会更快，这种改动可以大大改善 THINPAD 与 ucore OS 联合实验平台的开发效率。而文件系统的功能拓展可以为大量依赖文件系统的功能拓展，比如编译，提供基础。

### 6.3 结语

我们的工作仅仅是 THINPAD 与 ucore OS 组成的实验平台上对网络功能与文件系统功能的一次功能实现，对这些功能的完善与使用还有待未来的进一步开发。希望我们的工作可以为 THINPAD 与 ucore OS 未来的相关的实验提供启发与基础。