2015 年《操作系统》课程设计总结报告

计 24 班 田博 2012011346

2015 年 6 月 18 日

Chapter 1

序言

在本次《操作系统》课程设计，我主要完成了两项工作，Intel Edison 上的 ucore+ 移植和 Linux E1000 网卡驱动的 ucore+ 移植，以下将分两章来介绍。

这两个项目并不是完全独立的，它们共享 LAPIC 和 IOAPIC 的代码，这部分代码主要参考之前课程 chanllege 的 SMP 实现[1]，报告[2]，这里不再赘述 APIC 相关的实现，总体比较简单。

[1] https://github.com/dxmtb/ucore_lab/tree/master/labcodes/smp

[2] https://github.com/dxmtb/ucore_lab/blob/master/labcodes/smp/report.md

Chapter 2

Intel Edison 上的 ucore+ 移植

2.1 完成功能概要

- ucore+ 在 uboot 上的启动
- Edison 上的串口读写（ioremap）
- 时钟中断支持
- RAMDISK 支持
- PCI 支持

2.2 uboot 的定制

uboot 中需要加入载入 E820 表和载入 ELF 到内存并执行的功能。
需要进行一下修改：

- 添加 loade820 指令（uboot 中有读取 E820 的函数）
- 添加 bootu 指令，直接从 MMC 中的 ELF 镜像载入到正确的地址，然后启动

2.3 uboot 的烧入和启动

在出现 GADGET DRIVER: usb_dnl_dfu 时，通过

sudo dfu-util -v -d 8087:0a99 –alt u-boot0 -D u-boot-edison-2014.04-1-r0.bin

可以刷入新的 uboot，加载运行后可以看到 U-Boot 2014.04 (Apr 15 2015 - 01:19:51) 的字样，表明是新的 uboot。

刷入新的 uboot 以后，先引导到 Linux，拷贝要加载的 kernel 到分区，假设 kernel 名为 kernel-i386-new.elf，则在重启后在 uboot 下运行：

loade820

bootu mmc 0:9 kernel-i386-new.elf

即可加载 uCore。

2.4 uCore 的修改

2.4.1 ELF Program Headers

首先编译以后的 ELF 需要进行修改。具体而言，ELF 各个 Program Header 的 p_pa（physical addr）实际上是错误的，是虚拟地址。而在 ucore 自己的 bootloader 中屏蔽了物理地址的高位得到了正确的值，但 uboot 中并没有也不应该做这样的事情。

所以需要写一个程序修改 kernel 的 ELF 的 header 使其值正确，具体可以参考 uCore 的 bootloader 和 uboot 的 bootelf 指令的实现。

2.4.2 串口输出方式

Edison 的串口输出是 UART 的 MMIO，通过对地址 0xff010180 进行读写。页机制启用前、后地址访问不到，通过临时加入新的段、ioremap 来解决。

另外需要注释掉原先串口初始化和输出的部分，否则似乎会影响 UART。

2.4.3 时钟中断支持和关闭 Watchdog

类似 SMP 中的时钟中断，需要 LAPIC（初始化中断）和 IOAPIC（启用中断）。中断处理完后需要调用 lapiceoi（不过 SMP 已经碰到过了）。另外 Edison 中无法动态获取 APIC 物理地址，硬编码一个值（Linux 中也是这样做的）。

Edison 的 watchdog 需要关闭，否则会定时重置 Edison。

movl 0x10f8 to 0xff009000

2.4.4 RAMDISK 支持和 PCI 支持

RAMDISK 是在内存中开辟出一块区域作为虚拟的硬盘，好处是不用涉及硬件 MMC 的读写驱动，主要是要实现 ramdisk_[read, write] 这两个函数。

具体做法是添加 initrd.S.in，内容为加载文件系统的 img，并在 ucore.ld.in 中加入一段，载入 initrd 符号。

Chapter 3

Linux E1000 网卡驱动的 ucore+ 移植

3.1 项目介绍

本项目的目标是将 Linux 内核（3.10 版本）中的 E1000 网卡驱动程序[1]移植到 ucore+[2]中正常运行。

截至项目结束，可以通过调用相应函数封装 skbuff 结构体交给驱动程序中的函数发包，当接受到数据包时，会引发系统中断并处理数据包。

配合 LWIP 协议栈，可以实现一个 HTTP 服务器，访问根目录下的文件。如果放上 HTML 页面，就可以访问静态网页。由于 LWIP 相关代码的潜在 bug，一次 TCP 连接结束后 LWIP 就会使得 kernel 崩溃，但可以验证 E1000 的驱动运行正常。由于 LWIP 并不是本次课程设计的目的，而只是一个展示手段，所以并没有针对这个 bug 进行修复。

3.2 自制 E1000 驱动

由于直接移植 Linux 的 E1000 驱动难度较大，可能造成无从下手和最后没有展示成果，所以打算先实现一个简单的 E1000 驱动，确保自己对

[1] drivers/net/ethernet/intel/e1000

[2] `https://github.com/chyyuu/ucore_plus`，6f3d82b

E1000 有一个基本的了解[3]。

3.2.1 PCI 支持和 E1000 驱动

PCI 和 E1000 驱动主要参考`https://github.com/bcalmeida/JOS` 的实现。由于 JOS 和 ucore 基本相似，所以 E1000 驱动部分代码基本可以直接重用。

3.3 LWIP 移植

在 E1000 驱动基础上，移植 LWIP 协议栈（版本 1.4.1）到 ucore+，确保一个基本的展示环境。这一部分相对复杂很多，并不能直接重用 JOS 的代码，原因是 JOS 是一个微内核的 OS，内核的很多请求分发给了内核线程，ucore 不能这么做。

LWIP 的移植主要要实现两部分代码，一部分是 semaphore 和 message box，另一部分是 ethernet interface，负责将接受到的包封装成 LWIP 的数据结构，并且根据包的类型调用 LWIP 的相应函数进行处理。这两部分的实现主要参考了`https://code.google.com/p/os-xv6-network/`

semaphore 和 message box 需要实现 timeout，然而 ucore 并没有相应的支持。semaphore 这里的实现比较暴力，通过一个循环不断尝试获取锁，直到超时。然而这并不会完全阻塞内核，原因是这一部分函数都是在另一个 kernel thread 调用的，时间片用完自然会运行其它 thread。然而为了高效比较好的方法是通过 timer 实现，不过项目过程中为了确保代码的正确性使用了简单粗暴却不容易出错的方法。message box 实际上可以通过调用 semaphore 相关的函数实现，对锁操作的细节要求较高，不过基本可以照搬 os-xv6-network。

更加麻烦的部分是 ethernet interface 的部分。在 lwip/netif/ethernetif.c 中基本实现好的一个框架，需要填充具体的收包、发包部分。然而这一部分并没有找到什么官方文档，甚至全靠自己摸索才知道要实现这样一个东西，确实是有点坑。找到这一部分代码就成功了一半，接下来仿照 os-xv6-network 调用适当的函数进行实现即可。

[3] `https://github.com/dxmtb/ucore_plus/tree/lwip`

另一个令人头疼的是 LWIP 的初始化问题，由于 LWIP 的版本问题，通过大量代码的阅读、文档查找，通过 tcpip_init 和 netif_add 可以实现 LWIP 的初始化。

发包的话在合适的时机 LWIP 会调用我们编写的相应函数，但是收包由于没有中断，实际上是通过启动一个内核线程不断查看 E1000 的状态寄存器，如果有包就收包实现的。

项目中还发现了 ucore+ 内核线程调度有问题。由于 trap 中如果判断是在 kernel 中就不会调用 schedule，本意是为了避免中断中调度程序，但是却造成了没有一个其它 thread 能抢占不主动放弃执行权的 kernel thread。而 LWIP 使用了 kernel thread，所以这个问题必须要解决。这里通过判断 thread 的名字，如果是可以调度的 thread 就调度，但这样无法避免嵌套中断时调度的风险。比较好的做法是记录中断的层数。然而由于之前的做法并没有在实际中出现大问题，所以没有修改。

3.4 Linux E1000 驱动移植

3.4.1 简介

Linux 的 E1000 驱动本质和之前的简化版本是一样的，通过 PCI 获取的 MMIO 地址，驱动程序对 E1000 进行监控、控制，只是 Linux 的驱动版本支持更多的功能和版本。通过 header-gen 对 E1000 驱动进行分析，主要需要实现以下种类的接口函数：

- PCI 的支持
- net_device 的创建、配置、操作
- DMA 地址的分配、映射
- ioremap 为物理地址分配虚拟地址
- NAPI 相关函数处理收包
- skbuff 的创建、操作
- 中断注册

3.4.2 PCI 支持

由于 Linux 的 PCI 支持比较完善、复杂，之前实现的 PCI 代码全部废弃。PCI 重要的有两部分：pci_dev 结构体的填充，pci bar 的读取，由于比较复杂，考虑直接从 drivers/pci 中扣取相关代码。

比较基本的是 pci bar 的读取，这些代码集中在 access.c 中，找到需要的代码将不需要的部分注释调。

pci_dev 结构体填充的关键函数是 pci_setup_device 函数，幸运的是这个函数基本只依赖一些基本的 bar 访问函数。手动把这个函数依赖的函数都保留，注释掉其它不需要的部分。

这一部分的困难之处在于 PCI 在 Linux 中十分复杂，甚至有 fixup 这种东西（我们并不需要）。要抽丝剥茧，需要对 Linux 中的 PCI 这一套系统有足够的了解，建议通读《Linux 内核源代码情景分析》中 PCI 的部分。最后定位到 pci_setup_device 这个函数，一切就变得简单很多。由于对于驱动而言，并不涉及 pci_dev 的填充，所以这一部分 header-gen 无法分析出来，需要我们自己分析。

3.4.3 其它

net_device 的创建主要是移植 alloc_netdev_mqs 函数，基本参考 Linux 的实现就可以了，去掉一些不需要的部分。

DMA 的实现实际可以很简单。首先硬件只需要知道 DMA 区域的物理地址就可以正常的 DMA，并不需要什么干预。由于 ucore 的内存机制比较简单，分配一块 DMA 区域（dma_alloc_coherent）只需要在 kmalloc 后获取相应的物理、虚拟地址，而虚拟内存的 DMA 映射（dma_map_single_attrs）只需要返回虚拟地址的物理地址即可（减去偏移）。

ioremap 的实现是通过分配 KERN_TOP 以上没有使用的虚拟地址空间，映射到需要的物理地址。

收包会调用 NAPI 相关的函数，这一部分整体可以参考 Linux 的实现，在合适的部分我们只要掐断数据包的传输，传输到自己需要的部分即可。这一部分由于依赖时间，而 Linux 的 HZ 和 ucore 并不统一，所以会有潜在的 bug，然而测试中并没有碰到。

skbuff 的操作主要是发包时要考虑。实际上 skbuff 相当于管理一块内存区域和多块其它片段，在报文的前后都保留有空间，所以比较复杂。在 ucore 中我们只需要用它表示一整个包就可以了，所以理解结构以后实现比较简单。

中断处理比较粗糙，直接 hard code 了中断号，实际上可以更加细致的控制。并不是做不到，只是稍微麻烦些。

3.4.4 有关 Header-gen 的思考

header-gen 是十分强大的工具，可以分析出一些 Linux 源代码所需要的最少的头文件，以及生成一份没有定义的函数原型（dummy）。比较理想的情况是填充完 dummy 函数就可以正常运行，但目前来看这只是一个难题。

运行移植的函数，如何提供正确的输入给这些函数也是很大的困难。很多 Linux 中的结构体结构复杂，填充每一个项并不现实，可靠的方法是提取必须的部分，然而这需要通读移植的函数及其依赖才能确定，实际上是工作量很大的部分。

实际上 header-gen 并没有像想象的那样减少工作量，原因是一方面即使使用所有的头文件，也是可以编译的，没有必要生成一份最少的头文件，生成函数原型确实减少了工作量，但这也仅仅是体力活，通过手动编译也可以发现没有实现的函数。

3.4.5 潜在工作

目前很多函数实现很粗糙，可以继续完善。资源的释放都没有实现。LWIP 的相关接口可能还有 bug。