JOS的ARM移植

曹胜操 - 1500012838

前言

我们将把JOS的部分实现移植到ARM平台上。目前已完成的进度：Lab2的代码和文档。所有内容均上传至GitHub：https://github.com/Friedrich1006/JOS/tree/arm。

首先，我们需要准备好所需要的工具。

- 由于我们需要在（比如说）运行在x86-64平台上的Linux编写出运行在ARM平台上的JOS，交叉编译是必须的。这里我通过`apt`完整地安装了`arm-none-eabi`的一套工具，包括`binutils-arm-none-eabi` `gcc-arm-none-eabi` `gdb-arm-none-eabi`。有了这些工具之后，我们就可以编译出合适的ARM平台上运行的代码。使用时需要注意前缀，如用`arm-none-eabi-gcc`代替`gcc`来进行编译。
- 我们可能已经习惯于x86所提供的一套很标准的计算机系统，但是到了ARM这里情况却不太一样。ARM硬件之间的差异可能非常大，我们需要特别地选择并采用其中的某一种。QEMU提供了五十种左右的ARM硬件的模拟，可以参见https://wiki.qemu.org/Documentation/Platforms/ARM。鉴于我并不识得其中的大多数，并且参考了前人所写的报告，并且相关的社区讨论和文档也比较充足，所以最终选择了Raspberry Pi 2这个平台（虽然我也并没有一台实际的机器）。

  `apt`以及6.828课程网站上面所提供的QEMU版本似乎都有些老，并没有对Raspberry Pi 2的支持，所以我选择了使用QEMU官方网站提供的2.11.0版本的源代码（https://www.qemu.org/download/）进行编译安装。其过程和课程网站的提示类似，注意在`configure`时，`--target-list`参数中需要包括`arm-softmmu`。
- 最后参考http://wiki.osdev.org/Raspberry_Pi_Bare_Bones，就可以写出一个简单的内核并在模拟器中打出`Hello World!`了。

接下来学习一个ARM的基本知识（http://wiki.osdev.org/ARM_Overview）。

- ARM中有7种操作模式，总的来说就是一种特权级较低的用户模式和特权级较高的其它几种模式。每种模式有着自己独立的栈和一些寄存器。
- ARM定义了7种异常：Reset、Undefined Instruction、Software Interrupt、Prefetch Abort、Data Abort、Interrupt Request、Fast Interrupt Request，这比x86要少得多。通过异常向量表中具体的指令，发生异常后处理器可以跳转到对应的处理程序，同时也会自动进入对应的模式。
- ARM中的寄存器非常的多。通用寄存器包括`R0`到`R15`，其中`R13`代表stack pointer，`R14`代表link register，`R15`代表program counter。
- ARM中的指令长度是固定的，A32模式长度为32位，Thumb32模式长度为16位。ARM属于RISC体系结构，只有较为简单的一些指令。
- 大多数ARM处理器也是有MMU和TLB的。页表结构与x86有所不同，支持各种不同大小的页，这部分内容将在下面叙述。另外，由于ARM硬件通常都是特别定制的，没有一种通用的方法可以探测到有多少物理内存，我们可能需要把这个信息直接编码到系统中去。

Lab 1: Booting a PC

Raspberry的启动

根据网上查到的一些资料（以https://stackoverflow.com/questions/16317623/how-does-raspberry-pis-boot-loader-work为代表），Raspberry与我们之前熟悉的x86机器的启动是不太一样的。首先是GPU先启动，然后把（比如说）SD卡中储存的内核装载进内存，再是ARM处理器启动，似乎无需考虑x86中那些较为复杂的启动套路。因此在JOS的ARM移植中，我们移除了boot loader的部分。同时在QEMU模拟时，参数也有所变化。

在x86的JOS内核启动的过程中，主要做了以下事情：

- `boot/boot.S`：关中断，初始化寄存器，打开A20，载入初始GDT，打开保护模式，设置初始栈，进入`bootmain()`。
- `boot/main.c`：从磁盘读入整个内核，跳转到入口。
- `kern/entry.S`：载入初始页表，打开分页，设置初始栈，进入`i386_init()`。
- `kern/init.c`：一系列的系统初始化。

移植到ARM平台上之前，我们需要先了解系统启动时初始的状态。为此，修改上面已经写出来的`Hello world!`简单内核，在最前面加上读取一些重要的协处理器寄存器的指令：

```
    mrc p15, 0, r0, c0, c0, 0
    mrc p15, 0, r0, c1, c0, 0
    mrc p15, 0, r0, c2, c0, 0
    mrc p15, 0, r0, c3, c0, 0
```

然后用`arm-none-eabi-gdb`进行调试，查看它们的值。

- 首先可以发现的是，整个的入口位于`0x8000`。也就是说，内核默认被装载到这个位置并从此开始执行代码。当然，我们之后也可以通过`kernel.ld`中的设置选择其它的位置。
- `mrc p15, 0, r0, c0, c0, 0`代表读取MIDR (Main ID Register)，其值为`412fc0f1`。经过一番资料的搜索，发现这代表模拟的处理器为Cortex-A15 r2p1（文档：http://infocenter.arm.com/help/topic/com.arm.doc.ddi0438d/DDI0438D_cortex_a15_r2p1_trm.pdf），和Raspberry Pi 2所使用的Cortex-A7不同，不过似乎问题不大。
- `mrc p15, 0, r0, c1, c0, 0`代表读取SCTLR (System Control Register)，其值为`00c50078`。查看文档，可知此时的控制状态包括：禁止MMU，禁止地址对齐检查，禁止数据和指令缓存，禁止跳转预测。我们稍后应该把这些选项打开。
- `mrc p15, 0, r0, c2, c0, 0` `mrc p15, 0, r0, c3, c0, 0`分别代表读取TTBR0 (Translation Table Base Register 0)和DACR (Domain Access Control Register)，其初始值均为`0`。我们稍后应该进行设置以使用分页机制。
- 还可以查看CPSR (Current Program Status Register)，值为`400001d3`，代表Supervisor模式，运行ARM指令状态，使用小端法，关闭IRQ和FIQ。

在实际实验时还发现一件比较奇怪的事情，QEMU模拟Raspberry Pi 2至少是四核，并且是四个CPU同时执行内核的代码。在网上搜索了很久，并没有找到相关解决办法。目前为止我们还不需要多核环境，更何况操作系统的初始化也应该是一个CPU去完成的。所以在内核代码的入口，先通过`mrc p15, 0, r0, c0, c0, 5`读取MPIDR (Multiprocessor Affinity Register)，检查其中的CPU ID，只允许一个CPU运行后面的代码，其它的进入死循环。希望有更优雅的办法。

总结一下，在ARM的`kern/entry.S`中，我们必须做的事情有：

- 检查CPU ID
- 载入TTBR0
- 载入DACR
- 设置SCTLR，允许MMU（不妨把上面提到的选项也都打开）

- 跳转到高地址
- 设置初始栈
- 进入`kern/init.S`中的`arm_init()`

虚拟内存地址翻译

这部分内容参考了ARM Architecture Reference Manuel (ARMv7-A)。ARM的MMU支持两种地址翻译表格式，Short-descriptor（32位）和Long-descriptor（64位）。我们采用Short-descriptor。

地址翻译的过程大致如下：

- MMU收到一个虚拟地址`va`
- 通过TTBR0或者TTBR1找到First-level table
- 以`va[31:20]`为索引在First-level table中找到First-level descriptor`d1`，First-level descriptor共有4K个，First-level table大小为16KB
- 根据`d1`的最低两位来确定接下来的翻译
  - `00`：无效
  - `01`：需要Second-level的翻译
  - `10`：直接指向1MB的Section（`d1[18] == 0`）或者16MB的Supersection（`d1[18] == 1`），对于前者用`d1[31:20]`拼上`va[19:0]`就是物理地址`pa`
  - `11`：保留
- 对于Second-level
  - 此时`d1[31:10]`表示Second-level table地址的高22位。找到Second-level table后，再以`va[19:12]`为索引找到Second-level descriptor`d2`，Second-level descriptor共有256个，Second-level table大小为1KB
  - 再根据`d2`的最低两位来确定页大小
    - `00`：无效
    - `01`：64KB大小的页
    - `10`或`11`：4K大小的页，用`d2[31:12]`拼上`va[11:0]`就是物理地址`pa`

我们主要只关注1MB Section和4KB Page的情况。系统初始页表我们采用前者，之后采用后者。

ARM中还有域(Domain)的概念，表示一些内存的集合。前面提到过DACR，其每两位可以表示一个域的权限，共可以有16个域。地址翻译过程中的descriptor中也有代表权限的`AP`位和所属域的编号。MMU对于内存访问权限的处理是这样的：

- 先看DACR中对应域的位
  - `00`：不可访问
  - `01`：客户级，需要进一步检查AP位
  - `11`：管理级，无需检查
- 对于客户级，`AP`是3位，根据设置`AP[0]`也可以用作Access flag
  - `000`：特权级不可读写，无特权级不可读写
  - `001`：特权级可读写，无特权级不可读写
  - `010`：特权级可读写，无特权级只读
  - `011`：特权级可读写，无特权级可读写
  - `101`：特权级只读，无特权级不可读写

    - `111`：特权级只读，无特权级只读

参考Linux的设置（https://github.com/torvalds/linux/blob/master/arch/arm/include/asm/domain.h），我们做一个简化处理：只用第一个域，并设置为客户级，由`AP`位来决定权限。于是将DACR设置为`0x1`即可。

输入输出

ARM中是没有`in` `out`这种对I/O端口读写的指令的，都是通过内存映射来完成。在Raspberry的环境下，我们并没有CGA，也并没有键盘。我们通过Raspberry的GPIO (General Purpose Input/Output)来实现输入输出。同样参考http://wiki.osdev.org/ARM_RaspberryPi_Tutorial_C，我们用GPIO来取代原来`kern/console.c`中的内容。此外，我们也需要把这部分内存映射做好。

函数调用栈

ARM中的函数调用栈布局与x86也有所不同。如前所述，`R13`代表stack pointer，`R14`代表link register，`R15`代表program counter，还有一个`R11`可以作为frame pointer，需要在编译时加上`-fno-omit-frame-pointer`的选项。

以`test_backtrace()`的递归为例，函数调用过程中对于栈的操作大致如下：

- 调用者准备好参数，前四个参数放在`R0-R3`中，再多可以放在栈里，这里就是把`x`放在`R0`里
- 用`bl test_backtrace`实现调用中的跳转，此时`bl`表示带链接的跳转，还会把下一条指令的地址放在`lr`中
- 进入被调用者，先把旧寄存器压栈`push {r3, r4, fp, lr}`，注意从右向左压
- 用`add fp, sp, #12`修改`fp`，此时`fp`指向当前栈帧底
- 正式开始函数的执行
- 返回时，旧寄存器弹出`pop {r3, r4, fp, lr}`，从左向右弹
- 用`bx lr`实现通过`lr`的链接返回

可以看到，正式执行函数之前栈的结构从高地址到低地址依次为：

- 旧的`lr`，此时的`fp`指向这里
- 旧的`fp`
- 旧的`r4`
- 旧的`r3`，此时的`sp`指向这里

根据这个就可以修改`mon_backtrace()`了。当然，我们没有办法在ARM中通过栈中的内容确定函数调用的参数，所以在测试中删除了参数相关的内容。

保留并修改的文件

下面列出在ARM移植中Lab1部分保留下来的文件及其主要修改，可能并非全部：

- `.gdbinit.tmpl`：删除了x86相关
- `GNUmakefile`：将选项`-m32` `elf_i386`分别修改为`-marm` `armelf`，`IMAGES`修改为`$(OBJDIR)/kern/kernel`，`QEMUOPTS`前面部分修改为`-kernel $(OBJDIR)/kern/kernel -m 256 -M raspi2`，调试器修改为`arm-none-eabi-gdb`
- `conf/env.mk`：定义了`GCCPREFIX='arm-none-eabi-'` `QEMU=qemu-system-arm`
- `inc/arm.h`：取代了`x86.h`
- `inc/mmu.h`：修改了MMU地址翻译相关内容，删除了x86特殊寄存器和描述符相关内容
- `kern/console.c`：重写了底层I/O

- `kern/console.h`：修改了声明
- `kern/entry.S`：重写了入口代码
- `kern/entrypgdir.c`：重写了初始页表
- `kern/init.c`：重写了`arm_init()`取代`i386_init()`
- `kern/kernel.ld`：删除了x86相关
- `kern/Makefrag`：删除了`boot`的内容，不再生成`obj/kern/kernel.img`
- `kern/monitor.c`：修改了`mon_backtrace()`适应ARM中的栈结构
- `lib/string.c`：删除了对x86优化的汇编代码

Lab 2: Memory Management

物理内存

正如标题所示，Lab2的主要内容就是对内存进行管理，其中最重要的是建立虚拟地址的映射。在这个部分的ARM移植中，我们主要就是修改`kern/pmap.c`中的代码。

前面已经提到过，由于ARM硬件通常都是特别定制的，没有一种通用的方法可以探测到有多少物理内存，我们可能需要把这个信息直接编码到系统中去。在JOS中，由于`KERNBASE = 0xF0000000`，最大只能支持`0x10000000`即256MB的物理内存，我们在使用QEMU模拟时也设置参数`-m 256`，这样我们就固定内存大小为256MB。如此也就不需要`kern/kclock.c`中的内容了。

在ARM中没有单独的I/O端口地址空间，所以外围设备的寄存器都被编码到地址空间中去了。我们只使用256MB的内存，所以不会影响到这部分地址空间。参考Raspberry Pi的官方文档 https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2835/BCM2835-ARM-Peripherals.pdf和 https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2836/QA7_rev3.4.pdf，可以获知这些外围设备的具体情况。

对于虚拟内存地址空间，我们完整保存原来JOS的内存布局，即`inc/memlayout.h`，特别的只是`[0xef800000, 0xefc00000)`这部分MMIO虚拟地址要映射到`[0x3f200000, 0x3f600000)`，作为GPIO的映射。

对于物理内存地址空间，我们在`page_init()`中进行设置，和原来类似，只是不需要再考虑IO hole的问题：

- `[0x0, 0x1000)`：主要为Generic interrupt controller保留
- `[0x1000, 0x0x100000)`：空闲
- `[0x0x100000, PADDR(boot_alloc(0, PGSIZE))`：内核及其它
- `[PADDR(boot_alloc(0, PGSIZE)), npages * PGSIZE)`：空闲

地址翻译

在经过`mem_init()`对内存管理进行初始化之后，我们就只使用12-8-12这样的两级翻译了。这个和x86的10-10-12有点不太一样，其中descriptor每个位的意义也有所不同，这些都写在`inc/mmu.h`中了。

我们仍然使用`KERNBASE`，可以通过虚拟地址`[0xf0000000, 0x100000000)`来直接访问全部的物理地址`[0x0, 0x10000000)`。

在x86的JOS中有一个十分精巧的`UVPT`，但是由于ARM中12-8-12这种奇妙的划分，我们没有办法直接把它迁移过来，只好舍去，通过其它的办法访问两级翻译表中的descriptor。还需要注意的是，First-level table大小为16KB，Second-level table大小为1KB，在`boot_alloc()`中，我们最好还是加上一个额外的对齐量的参数，比如`kern_pgdir`就应该是16KB对齐的。但是在有了物理页管理之后，如果需要通过`page_alloc()`申请一个

Second-level table，只能占用掉一整个4KB页，会浪费掉3/4的空间。因此，我们需要做一个优化，写一个函数`page_alloc_quar()`来申请四分之一个页作为Second-level table。

保留并修改的文件

Lab2中主要就是修改`kern/pmap.c`，此外也删除了一些内容，如`kern/kclock.c`等。

`kern/pmap.c`中的函数修改主要如下：

- `arm_detect_memory()`：取代了`i386_detect_memory()`，直接得到内存大小
- `boot_alloc()`：加上了对齐量参数
- `mem_init()`：修改了调用各子函数时的参数，特别是`boot_map_region()`的参数，以及用ARM的`TTBR0`取代x86的`cr3`
- `pgdir_walk()` `boot_map_region()` `page_insert` `page_lookup`：修改了各个宏
- `check_***()`：根据我们的修改后的内容检查。这部分修改非常的繁琐，而且其中对于页面的强行操作也可能导致一些bug，在`make grade`通过之后最好将这些检查都注释掉