操作系统大作业报告

文件系统相关部分

目标

- 为MIPS-ucore中的simple file system添加文件创建功能
- 利用THINPAD上的Flash提供永久储存

实际工作

qemu相关

- 为qemu中mipssim架构添加了pflash支持, 用于模拟THINPAD上的flash.
- 分析了chenyh学长对qemu的修改, 对学长的这部分工作进行了文档上的完善, 并连同移植pflash的方法写成了一份文档, 供之后的同学作参考.

文件系统相关

- 为sfs添加了创建文件的功能

Flash相关

- 设计了Flash的使用策略, 在操作系统中具体实现, 并在修改过的qemu中模拟通过.

其他

- 将上学期《编译原理》中修改的decaf编译器, "移植"到了mips-ucore当中, 能够将ucore中的decaf文本文件转化为可执行的文件并在THINPAD上运行.

工作介绍

qemu相关

这部分的工作已经有了单独的文档(请见《(挑战性课程用)简明QEMU手册》), 故不在此赘述. 之所以单独写成一份文档, 是因为希望这份文档能够在之后的32位MIPS组成原理实验起到作用, 也算是我做了一些微小的贡献.

文件系统相关

已有工作

ucore-plus当中其实已经实现了文件的创建功能, 但在实现的时候并没有发现.

思路与设计

文件系统部分的目标是要为sfs添加创建文件的功能.

添加文件创建功能本身工作量并不大, 但要知道如何添加, 就需要对当前的sfs文件系统有比较透彻的了解. 现在的sfs中是能够实现从文件夹中读出文件的, 因此跟踪`ls`命令的运行过程, 将其中

的"读出"换成"写入"即可, 具体还要做的事情包括:

- 为新创建的文件创建inode并初始化
- 为刚创建的inode分配实际的储存块din, 并将其指向该inode, 将sfs标记为`super_dirty`
- 在parent_inode(即代表创建文件所在的文件夹的inode)中找到空的slot, 并将inode同这个slot联系起来
- 将内容同步到ramdisk上

测试场景

参考Flash相关工作的测试场景

Flash相关

已有工作

在ucore-plus当中, 已经实现了yaffs(yet another file flash system)文件系统的移植. 整个文件系统相关部分的前期工作也都是围绕将ucore-plus中的yaffs移植到mips-ucore中展开的, 但经过后来的调研, 发现yaffs文件系统并不适用于THINPAD上的nor flash. 因此放弃了yaffs的移植, 转而在sfs文件系统中添加使用flash的策略, 而这样的工作, 在之前应该没有.

预期描述

在文件系统上进行的操作, 在reset之后重新启动, 仍然能够保证之前的数据仍然存在.

思路与设计

总体思路

现有的mips-ucore并不利用Flash, 而是将文件系统建立在ramdisk之上. 因此在这次实验当中, 永久化储存采用了最简单的思路, 即将ramdisk的数据搬移到Flash上做储存, 在启动的时候, 又将Flash的数据整个搬移到ramdisk中, 然后再进行sfs的挂载.

NOR Flash的特点

常见的Flash分为两种, 一种是NAND Flash, 另一种是NOR Flash, 而THINPAD上的储存介质是NOR Flash. 总的来说, Flash的特点是读出比较快, 但写入的时候只能将1写为0, 如果想要把0写为1, 就需要对Flash进行成块的擦除, 即将一个较大的储存块(一般从几K到一百多K不等)全部写为1, 但需要注意的是, Flash是存在着擦除寿命的, 擦除次数超过一定数量, 相应块的数据就会发生损坏. 由于Flash的特殊性, 因此针对Flash会采用特殊的文件系统, 例如yaffs文件系统, 会考虑Flash各块之间的负载平衡, 并且引入纠错码对错误的数据进行矫正, 变相地延长Flash的寿命.

然而, yaffs文件系统只适用于NAND Flash, 而不适用于THINPAD上的NOR Flash, 相对于NAND Flash, NOR Flash的读速度较快, 写入速度略慢, 但在擦除上, NAND Flash一次擦除一般在毫秒级, 而NOR Flash一次擦除需要的时间是秒级, 且NAND Flash的擦除块大小是8K到32K, 而NOR Flash的擦除块大小为128K, 这就意味着对于在擦除上要付出的代价, NOR Flash会远大于NAND Flash, 因此, 如果不考虑Flash擦除的差异而选用yaffs, 那么性能上应该不可接受.

异步擦除及其实现

鉴于NOR Flash的特点, 因此在使用NOR Flash的时候应该考虑如何减少Flash擦除对系统运行速度的影响, 因此采取了异步擦除的处理方法. 具体实现如下: 在ucore启动时, 系统会初始化一个内

核线程swapper_thread, swapper_thread的工作模式是, 在全局维持一个用于记录块状态的和块同步情况的数组, 以及一个busy状态, 然后不断循环检查数组状态, 对于应该被擦除的块, 对Flash发出擦除命令, 但不做确认, 即进行 **异步擦除** 并将busy置为true, 在下一次检查到需要写入或擦除的块时, 若busy为true, 则做确认检查, 检查通过则继续执行操作, 否则跳过操作等待下一次检查; 而写入采用的是 **同步方法** .

块状态转换及时机

在之前已经说到, swapper_thread的工作模式是, 在全局维持一个用于记录块状态的和块同步情况的数组, flashswap.h为外部的代码提供了接口, 用来修改这些数组的状态. 而块状态包括以下的几种情况:

- THINFLASH_STATUS_CONSISTENT 表示块中的内容与ramdisk中一致
- THINFLASH_STATUS_DIRTY 表示块中内容与ramdisk中的不一致
- THINFLASH_STATUS_CLEANING 表示该块正在被清空
- THINFLASH_STATUS_CLEAN 表示该块已经被清空

而四者的状态转移恰好构成一个圈, 即一开始块是"一致的", 在对应地址有内容被写入后, 变成"脏的", swapper_thread发现"脏块"之后, 开始异步擦除, 状态变为"正在被清空", 当该块再次被检查时, 检查当前Flash是否忙, 若已经不忙, 则将状态置为"空块". "空块"被写入之后, 则变为"一致的".

上面的四个状态是用于控制擦除的, 而块同步情况则是用来控制写入的, 这些状态包括:

- THINFLASH_SYNC_NO_NEED 该状态表示不需同步
- THINFLASH_SYNC_SHOULD 该状态表示需要同步且需要立刻同步
- THINFLASH_SYNC_LATE 该状态表示需要同步但暂时不用同步

状态转换如下: 一个块一开始"不需要同步", 在对应的地址有内容被写入之后, 变成"需要但暂时不同步", 这是为了防止文件在短时间内反复写入而造成Flash的反复擦除而设置的状态, 当被写入的文件被关闭之后, 所有的"需要但暂时不同步"的状态, 会变为"需要立刻同步", 而当swapper thread检查到一个块的状态为"已被清空", 且"需要立刻同步", 那么就会将ramdisk中的内容, 同步写入到Flash当中.

测试场景

在用户态实现了两个小程序, 一个是touch, 用于简单创建一个空文件, 另一个是fwrite, 用于向一个制定的文件当中写入一个字符串. 测试的过程类似于:

```
touch a.txt
fwrite a.txt 12345678974454234234234234234234
```

然后进行reset重启ucore, 通过ls命令和cat命令检查a.txt是否存在, 以及a.txt中的文件内容是否是12345678974454234234234234234234, 若是则代表测试通过.

另外, 后文提到的"编译"fibonacci.decaf的测试场景也可以用于此.

反思与改进

这部分是我工作中比较满意的一部分, 但仍然觉得状态的修改过程与文件系统的耦合比较大, 如果后人想要复现我的工作, 文件系统部分可能需要用可视化的`diff`对比一下.

decaf编译器在ucore上的"移植"

已有工作

在上学期的《编译原理》课的挑战性项目中, 我们(董豪宇, 李宇轩, 王少雄)针对Olddriver cpu所支持的指令集, 对课程中的decaf编译器进行了裁剪, 使其能够将一个合法的decaf源代码程序编译为符合Olddriver指令集的mips32汇编程序.

预期描述

在ucore内部实现一个"编译器", 能够将ucore中的decaf源程序"编译"为可执行程序, 并在THINPAD上运行, 整个过程应该是在线的.

思路与设计

在ucore中完整地搭建一个编译器显然是不可行的, 因此在这里, 采取的策略是, 在ucore内部搭建程序, 将decaf源代码从ucore内部通过串口传到PC端, PC端收到decaf源代码之后, 再调用decaf编译器和mips汇编器及连接器生成可在ucore下执行的二进制文件, 再将该文件用串口传回. 下面分别介绍PC端和ucore端的设计思路, 以及两端的协议.

PC端

PC端的实现, 通过更改串口通讯工具`terminal`来实现.

具体的改动包括两处:

一是添加了`send_file`线程, 用于将本地的decaf源文件编译为ucore下可以运行的二进制文件, 再将其传输到THINPAD中实际运行. 具体的传输部分会在 **通信协议** 部分具体说明, 再次只说明编译的过程和思路.

编译包括两个步骤, 包括将decaf源程序编译为符合olddriver CPU指令集的汇编文件, 以及将汇编文件编译为二进制文件, 这一步的编译需要用到JAVA虚拟机, **需要注意的是**, 命令行中的`java`命令和c代码中使用`system("java")`所使用的java可能版本号会有不同, 这是可能会引发`java.lang.UnsupportedClassVersionError`错误, 解决方法是`which java`获知命令行`java`的绝对路径, 并用于程序当中, 即`system("/.../java")`, 对于汇编代码到二进制的编译, 直接调用目标平台为mips的`g++`即可, 但要注意的是, 需要将包含`decafS.S`和`decaf.c`以及其他`user/lib`下的文件编译得到的`libuser.a`链接到目标当中.

二是在`recv_data`线程当中, 加入两个状态机, 分别负责接收decaf源文件, 以及收到特定字符串之后就会启动`send_file`线程, 而具体的过程, 请见 **通信协议** 部分.

ucore端

ucore端的实现, 是通过odc(olddriver decaf compiler)这一应用程序实现的, 这一程序的实现的功能主要包括:

- 打开decaf源代码文件
- 创建ucore本地二进制文件(空)
- 在以上两步都成功的情况下, 将源文件从串口传出, 并将相应的二进制文件从串口读回文件

系统, 这一部分的具体细节请见 **通信协议** .

通信协议

串口在绝大部分情况下都不会有错传的情况, 因此在这部分的通信协议当中, 没有考虑验错.

`odc`运行之后, 由于decaf源代码只保含可见字符, 因此向主机发送`ENQ`和`STX`作为发送文件的开始信号, 在PC端terminal中`recv_data`按顺序收到了`ENQ`和`STX`之后, 即进入接收decaf源程序的状态, 在这个状态下, 收到的字符不会写入到`stdout`, 而是写入到一个文件当中. 发送结束之后, ucore端会发送`ETX`和`EOT`两个字符, PC端受到这两个字符之后, 就会启动`send_file`对收到的源文件进行编译, 编译完成之后, PC端会向`ENQ`和`STX`, ucore端受到之后, 就准备接收二进制文件长度, 这会由PC端分四次传过来. 在传递二进制文件的时候, 每传4096个字符, 就会有一次停等确认, 在这次停等当中, ucore也会将文件写入.

测试场景

将一个名为fibonacci.decaf的decaf源代码通过编译方式存在ucore当中, 执行odc, 应当生成可执行文件a.out, 执行a.out, 应当输出fibonacci数列的前十项, 并等待一个输入数字, 获取该数字后给出该数字对应的fibonacci项.

反思与改进

主要从设计角度来说, 一开始认为这部分是很简单的工作, 但之后就逐渐地认识到之前没有考虑到中断引发的ucore端收包丢失, 因此丑陋地将开关中断移到了用户态.

实际上, 远程传输文件应该形成一套系统的框架, 对通信协议解耦, 并对上层提供系统调用, 这才是安全且优雅的做法, 这是大概的思路:

```
struct transformer {
    // member status
    // ....
    // functions
    unsigned int (*connect)(const char *, unsigned int);
    unsigned int (*close)(const char *, unsigned int);
    unsigned int (*send)(const char *, unsigned int);
    unsigned int (*recv)(char *, unsigned int);
}
```

这样, 对于不同的协议, 采取不同的实现即可.

另外, 在terminal中还存在着很多硬编码, 应该将其参数化.