操作系统方案一实验报告

SHELL 优化与改进

邱泓钧 岳一飞 张佳瑜 肖翱 ▸软件二字班 ▸2015-1-23

目录

项目简介

项目特色

改进 xv6 的 shell。除了基本的补全、历史、光标，还有其他新增功能，具体包括翻页、菜单、便笺、命令权限、命令修正、编写 python 解释器、brainfuck 语言解释器等。

运行环境

Ubuntu 14.04　Qemu　Xv6

其中 xv6 是通过 git clone git://pdos.csail.mit.edu/xv6/xv6.git 所更新的最新版本。

人员分工

人员	学号	模块	内容
邱泓钧	2011013259	交互优化	窗口模块划分，特殊按键响应
岳一飞	2012013280	命令增改	新增命令、优化命令、命令权限
张佳瑜	2012013304	脚本支持	Python 解释器
肖翱	2012013314	脚本支持	Brainfuck 语言解释器，正则表达式

交互优化

窗口划分

模块	高度	公式	内容
Main 主体	15	25*0.618	显示当前命令与历史命令
Menu 菜单	4	25*0.382*0.385	菜单提示栏，用于智能提醒
Memo 便笺	6	25*0.382*0.618	便笺显示栏，方便记录数据命令等

通过黄金分割比将 25 行的窗口分隔成不同区域，并区分不同颜色，使得窗口划分较为合理、美观。此外，三者的高度具有一定的扩展性，可以在文件 console.c 中修改。

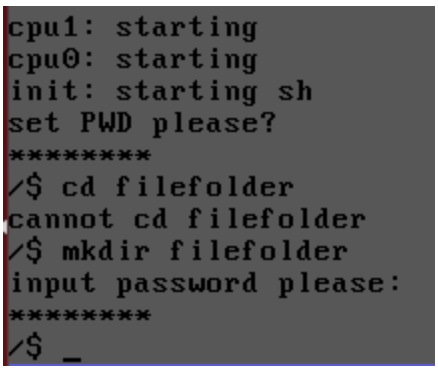
按键响应

实现功能	响应按键
密码屏蔽	NULL
下拉菜单	NULL

便笺文本	ESC
自动补全	TAB
历史记录	UP、DOWN
光标移动	LEFT、RIGHT
回退删除	BACKSPACE、DELETE、INSERT
翻页查询	PGUP、PGDN、HOME、END

在 kbd.h 中可以查询到不同按键对应的键码。交互部分主要通过修改 console.c 中的函数 consoleintr(int (*getc)(void))读取到的字符来判别。

密码屏蔽



为了提高隐私，用户在屏幕输入密码时需要用‘*’代替。

判断用户输入的代码在 sh.c 中，而修改屏幕显示的代码在 console.c 中。

实现两者之间通信特别麻烦，尝试过 extern 或 include 同一个头文件都不行，询问方案二的大神说只有 pipe 的方式。最后另辟蹊径，让命令部分的文件通过 printf 函数输入一个特别的符号如 0xCC，console.c 显示时如果读取到该字符，则切换密码模式。

便笺文本

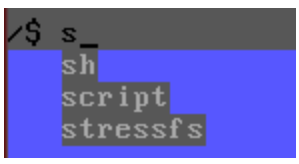
当初选题的时候，一直在考虑如何做出具有特色的 SHELL。考虑到平时大家经常使用 windows 的便笺记录一些简单的事项，因此决定在其中做一个简单的便笺，用于记录简单的命令，或者运行 python 等脚本时的结果等。

通过 ESC 可以切换主体和便笺模块，分别用 mainPos 和 memoPose 记录切换前的位置。

便笺模式由于与主体模式不同，不能套用先前的字符处理函数，自行实现了 handleMemo(int c, int pos)函数。支持 ENTER、INSERT、DELETE、翻页和方向键。

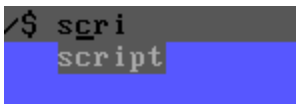
为了提升友好性，用户向上翻页时光标不能停留在主体，因此切换到便笺模式；相反，切换到便笺模式时会自动将页面拉倒最下方显示光标。

下拉菜单



下拉菜单是对 SHELL 交互体验最有创新的提升。对比 windows 下的 cmd 和 ubuntu 下的 terminal，两者的补全不太相同，但用户都不能一目了然地知道自己可以输入的命令名称，因此 windows 经常补全发现不是自己想要的要只能不断 tab 切换甚至回退，ubuntu 经常补全发现没有反应然后再试着多输入几个字符再补全。判定为密码锁定或 python 解释时不进行匹配显示。

用户每次输入一个字符，我们便会动态判断是否有匹配的命令，如果有，则将对应可补全的命令列入菜单栏中，如果不符合则清空菜单。此外，当用户移动光标修改时，菜单也能匹配到正确的位置。



另外，考虑到上下键用于历史命令提示，因此没有选择下拉菜单的功能。用户补全会自动选择默认第一个菜单选项。

自动补全

当用户输入命令时，通过 tab 可以补全第一个命令，并用特殊的颜色表明。通过自动补全，我们可以使得命令输入更加快捷，如右图 usertests 命令只需要 u+TAB 两次按键即可。

具体实现方法是当用户输入字符并匹配成功时，记录匹配信息如命令个数 matchCmdNum、第一匹配后缀长度 suffixLen、匹配命令索引 matchCmdIndex[]等。当用户下一次输入'\t'时，根据记录计算出应该正确显示的命令部分，赋予 crt 显示以及 input 执行，即可呈现自动补全效果。

历史记录

上下可以选择历史输入。将用户输过的命令通过数组记录下来。

光标移动

shell 中的光标是根据 pos 的位置显示的，具体的获取与显示方式见图。

```
int pos;
outb(CRTPORT, 14);  pos = inb(CRTPORT+1) << 8;
outb(CRTPORT, 15);  pos |= inb(CRTPORT+1);
pos++;
outb(CRTPORT, 14);  outb(CRTPORT+1, pos>>8);
outb(CRTPORT, 15);  outb(CRTPORT+1, pos);
```

当用户移动左右键时，记录新的 posoffset 记录光标距离 input.e 的偏移量。每次移动更新 pos 和 posoffset。根据 input 设定左移上限，根据 posoffset 设定右移上限。

之所以把光标移动单独写出来是因为当时不知道光标移动后回车需要修改 input 花了很多时间调试，而不是因为强迫症要给每个不同按键单独列一个标题。

回退删除

主要是除了修改 pos 还要修改 input 结构体，对于 input.r、input.w、input.d。配合光标移动可以修改输入字符。

翻页查询

考虑到整页过于庞大，将 pageUp 和 pageDown 改成逐行翻页。同时支持 home 键和 end 键。只是 kbd.h 文件中提及 home 键的键码并不正确，因此无法正确运行（但将函数中改成其他键码可以正确实现 home 的效果）。

具体实现是设置数组记录历史，每当 scroll up 时记录更新的页面（第一次记录全部），此外当 pageUp 和 home 的时候会暂时记录最新的一行。通过 showPage(int startScreenLine) 函数根据索引现实在窗口中显示对应记录。

命令增改

添加权限

当首次启动 xv6 时需要设置密码。当用户使用 mkdir 和 rm 命令时，需要输入密码后才可以继续操作。若 xv6 代码进行改动，则重新 make 后启动时需要重新设置密码。输入密码时用“*”显示。

xv6 启动时系统执行 firstInit()判断是否是首次启动。首次启动时，提示用户设置密码。用户密码经过加密后以一个根目录下不显示的文件夹名的形式保存。当用户输入 mkdir 和 rm 等命令时，系统提示用户输入密码。将输入的密码加密后的序列与系统中保存的加密后密码进行比对，确定是否输入正确。若输入不正确，则系统拒绝相应命令。

优化命令 cd

使用方式：cd path（path 可以为绝对路径或相对路径；“cd /”或“cd ~ ”返回根目录）
原有 xv6 中 cd 进任一目录（非根目录）后，就不能执行除 cd 以外的其他命令，其他命令只能在根目录下执行。改进后，用户 cd 进任一目录下时可以执行 mkdir, cat, rm 等任一操作。并且在等待用户输入命令行首端显示当前所在路径。
具体**实现思路**如下：

1. 系统命令以文件形式保存，因此原有系统只能在根目录下执行这些命令。修改思路为：在全局申请一个变量保存当前路径，在调用任一命令对应的函数时，多传递一个参数（当前路径），先通过 chdir()进入根目录，然后执行这一函数，在具体实现过程中使用参数中的“当前路径”。
2. cd 命令执行时，先判断参数 path 是相对路径还是绝对路径，然后修改变量中保存的当前路径值。
3. 在每次指令执行结束后打印行首“$”字符前，先打印变量中保存的当前路径。

优化命令 cat

使用方式：catfilename
原有 cat 命令后的文件名若不存在，则提示文件不存在。改进后，若文件不存在，则创建名为 filename 的新文件，并提示用户新文件已创建。

新增命令 cp

使用方式：cp srcfile destfile
具体**实现思路**如下：
1. 将 srcfile 对应文件复制为当前目录下名为 destfile 的文件。
srcfile 可以为当前目录下某一文件名或任意位置文件的绝对路径。系统识别后，先进入文件所在目录（当前目录下文件则不需这一步），打开文件并读取文件内容；在当前文件夹下创建名为 destfile 的新文件，写入源文件的内容并保存。
2. 将 srcfile 对应文件夹复制为当前目录下名为 destfile 的文件夹。
srcfile 可以为当前目录下某一文件夹名或任意位置某一文件夹名的绝对路径。系统识别后，先创建名为 destfile 的新文件夹，然后将源文件夹下所有文件复制到新文件夹下（若原有文件夹下无文件则不需这一步）。这时调用了复制文件的功能函数。

新增命令 rename

使用方式：rename prename name
具体**实现思路**如下：
1. 将文件名为 prename 的文件重命名为 name
实现：使用 link 将当前文件与新文件名链接，然后使用 unlink 取消与原有文件名的对应。
2. 将名为 prename 的文件夹重命名为 name
实现：先将原文件夹复制为名为 name 的新文件夹（文件夹中文件也同时复制），若原文件夹为空，则直接删除原来文件夹，否则需要删除原文件夹中所有文件，最后删除原文件夹。

脚本支持（Python）

实现功能

变量赋值，四则运算，条件判断，循环控制，错误信息，定义函数，内置函数等

具体实现

程序所需函数全部定义在 python.c 中，不需任何外部依赖，在 xv6 中引入该文件，修稿

makefile 增加文件即可。

程序模仿 python 语言的交互式运行过程，下面的功能所对应代码与 python 写法基本相同，不同之处我们会加以说明。

进入 main 函数之后，程序按行读取用户输入，并调用相应函数判断所输入命令是否为变量显示，变量赋值，条件判断等，并调用相应函数进行处理。处理结束后进入下一轮循环。For，if 与 def 的代码体不是立刻执行，因此在相应的执行函数里实现。如果一条命令没有任何语法能够匹配，则输出 syntaxerror。

变量表

程序维护一个 hashtable tab 存放已定义的变量值。Hashtable 在主函数运行之初定义，之后作为参数以指针形式被传入各个函数。哈希表相应的操作函数可以在文件中找到。目前程序只能正确处理纯字母的变量名。包含数字的变量名可能有问题。

四则运算

四则运算的实现为网络代码修改而成。具体过程为：首先将中缀表达式转化为后缀表达式，并使用特殊符号标记变量的初始与结束位置，然后用后缀表达式的求值方法进行求值。如果变量名为字母，则在变量表中寻找，如果是数字，就调用 toInteger 函数转换为数字。如果出现除零错误，则输出 ZeroDivisionError 异常。

程序目前只能支持整数运算，因为变量表中只能存储整数。

异常信息

文件中定义了全局错误码 errorcode 记录发生错误的类型。文件中通过宏定义定义了一些错误类型，有 SyntaxError，NameError，IndentationError，ZeroDivisionError,TypeError,ValueError。程序执行时会在特定时候检测错误码，发现有错误则返回，返回至最上层函数时，函数检查错误码与错误信息，输出错误信息并将错误码置为 0。程序中定义了 errorinfo 存储错误信息，但只是留作后续完善接口，程序中没有实际用到。

If，for 语句

程序判断一条语句是否为 if，for 语句的方法是将输入按照空格分解，并检查分解后得到的字符串是否满足一定的规则。程序不断读取用户输入，直到用户输入不再以四个空格开头为止。然后程序根据命令中的信息执行语句块的内容。目前只能支持 if 语句，不支持 else 和 elif；对于 for 循环，只能支持 for 变量名 in range(数字): 的格式，变量名可以任意，数字不能以变量名表示

函数定义

程序没有实现局部变量空间。函数作用域为全局作用域。可以在括号中增加变量，但没

有实际作用。函数不能有返回值。

由于函数没有返回值，在程序中利用函数的返回值作为变量进行计算暂不支持。

脚本支持（brianFuck）

语言介绍

Brainfuck 是一种极小化的编程语言，由 Urban Müller 在 1993 年创建，由于名字中包含不雅词汇，故而又名 brainf*ck、brainf***、BF。此语言看似非常简单，相比于 C、C++、Java 等高级语言而言，它只有 8 种指令，分别是“><.,+=[]”。但它却符合图灵的完全思想，可以与图灵机相媲美，能完成任何计算任务，尽管它的代码看起来时如此怪异。根据个人感受而言，这种代码就像它名字里所表达的意思那样晦涩难懂，简直可以被当做密码来使用。下面是针对 brainfuck 中 8 种指令的详细介绍：

指令	含义	等价的C代码
>	指针加一	++ptr;
<	指针减一	--ptr;
+	指针指向的字节的值加一	++*ptr;
-	指针指向的字节的值减一	--*ptr;
.	输出指针指向的单元内容（ASCII码）	putchar(*ptr);
,	输入内容到指针指向的单元（ASCII码）	*ptr = getchar();
[	如果指针指向的单元值为零，向后跳转到对应的]指令的次一指令处	while (*ptr) {
]	如果指针指向的单元值不为零，向前跳转到对应的[指令的次一指令处	}

就像惯例一样，学习任何一门编程语言，我们总是要从最简单的“Hello world!”程序入手，下面就是用 brainfuck 编写的一段“Hello world!”代码：

```
++++++++++[>+++++++>++++++++++>+++>+<<<<-]>++.>+.+++++++..+++.>++.<<+++
++++++++++++.>.+++.------.--------.>+.>.
```

运行结果是：Hello World!

在此次大作业中，我们针对 shell 进行了一些功能拓展，这里的 brainfuck 解释器就是其中的特色功能之一。突破 shell 的常规传统功能，让命令行窗口也能成为我们编写 brainfuck 代码的平台，方便有趣，很有意义。

实现思路

因为 brainfuck 语言的特点本身就是简洁且功能强大，因此写解释器本身的代码量并不是特别的复杂。概括来说就是当用户输入 brainfuck 代码之后，我们获取这些代码，并将这些代码中的每一个符合其语言规范的指令翻译成相应的电脑能够确切执行的操作，根据翻译后得到的计算机应该执行的一系列有序操作序列，我们将一一执行，直到最终执行完毕，

得到预期的结果并返回或输出。列举一个生动形象的例子，这个过程就像是 C++语言也会被电脑里的解释器解释为机器可以执行的一系列操作一样。

具体实现

Brainfuck 解释器可以用很多种语言来写，比如 C、C++、Java 等，甚至是 brainfuck 语言本身也可以被用来编写解释器。这里选择个人比较熟悉的 C 语言，在用户输入 brainFI 指令后，就进入 brainfuck 编程状态，用户会进行 brainfuck 编程，我们拿到这些代码之后，将其存储在一个结构体 vm 中。vm 中包括一些类似指针的 ip、sp、bp，也包括相应的类似内存块的东西，cs、ss、ds 分别模拟代码段、栈、数据段。然后会通过 8 个函数，将 brainfuck 的每一个字符解释成机器可以执行的有效操作，通过 setup（）函数可以根据得到的 brainfuck 代码设置相应的指针，内存段，并计算得到拥护输入的 brainfuck 代码对应的一系列有序的操作。接着调用 run（）函数，有序运行这些操作，并将结果直接输出。在写 brainfuck 解释器的过程中，如果要说一些技术难点的话，我想就是 xv6 本身的标准库太弱了，很多东西需要自己手动实现，非常耗时。

THANK YOU FOR YOUR PATIENCE !