Lab: Xv6 and Unix utilities Optional challenge shell exercises

2024/11/30 · 梅莉莲子糕

xv6 lab1 实验记录，用于以一个初学者的视角分享一些小想法和记录实验过程，方便日后复习，同时记录一下某些标记为 easy 但实际上一点也不 easy 的问题

1. modify the shell to not print a $ when processing shell commands from a file (moderate)

这个问题实际上是要让 shell 知道我们执行的命令是从控制台中输入的还是从文件中输入的.在 C 语言中有 isatty 函数， man isatty 可以看到其描述为：

> The isatty() function tests whether fd is an open file descriptor referring to a terminal.

于是我们的任务便是简单实现一个 isatty 函数,并在每次 sh 输出前添加一个判断语句即可.

TTY

TTY(TeleTYperwrite)原指电传打印机，现主要指操作系统内核中的 TTY driver,其中包含的 line discipline 能正确解读 terminal 中传入的信息并返回给 terminal,在这里我们可以简单的把 TTY 看成终端.

观察 xv6 提供的 stat.h 文件

```
#define T_DIR     1   // Directory
#define T_FILE    2   // File
#define T_DEVICE  3   // Device

struct stat {
  int dev;     // File system's disk device
  uint ino;    // Inode number
  short type;  // Type of file
  short nlink; // Number of links to file
  uint64 size; // Size of file in bytes
};
```

我们可以通过检测 stdin 的 stat 的 type 字段来判断其来源是目录，文件还是设备，同时，如果是设备，我们希望其设备编号(dev)为 1(stdin)于是我们有：

```
// sh.c
int isatty(int fd){
  struct stat st;
  if (fstat(fd, &st) < 0) {
    panic("Error to stat");
  }
  return st.type == T_DEVICE && st.dev == 1;
}
```

```
// sh.c
  getcmd(char *buf, int nbuf)
{
  if(isatty(0)){
    write(2, "$ ", 2);
  }
  ...
}
```

2. modify the shell to support tab completion (easy???)

难度诈骗，实际上实现 tab 补全需要非常仔细，并且需要修改内核代码

想要实现一个 tab 补全功能首先需要让用户按的 tab 能被捕获，但是一般情况下，我们在 termial 的命令行中输入字符只有在按下回车后才会被整行传递给 shell 中的处理函数，我们在按下 tab 时无法让 shell 及时做出反应，因此我们需要阅读内核中负责键盘 io 的部分，添加相应逻辑，这一部分在 xv6 中的 console.c 中.

网络上有许多关于 xv6 代码的解读，这篇博客详细解读了 console 和 uart 的交互与实现，我在这里简单介绍一下 console.c 中的部分代码

首先是 consoleintr 函数，这个函数在键盘中断发生时被调用，用于解读键盘输入的数据存入 cons.buf 中，并且在回车到达时唤醒用户程序，将 buf 中的数据传递给用户程序

```
//
// the console input interrupt handler.
// uartintr() calls this for input character.
// do erase/kill processing, append to cons.buf,
// wake up consoleread() if a whole line has arrived.
//
void
consoleintr(int c)
{...}
```

接着是 consoleread 函数，这个函数在用户程序调用 read 函数时被调用，用于将 buf 中的数据传递给用户程序

```
// user read()s from the console go here.
// copy (up to) a whole input line to dst.
// user_dist indicates whether dst is a user
// or kernel address.
//
int
consoleread(int user_dst, uint64 dst, int n)
{...}
```

consolewrite 函数在用户调用 write 时调用，用于将数据输出

```
//
// user write()s to the console go here.
//
int
consolewrite(int user_src, uint64 src, int n)
{...}
```

consputc 函数用于将字符写入到 console 中，让用户可见

```
//
// send one character to the uart.
// called by printf(), and to echo input characters,
// but not from write().
//
void
consputc(int c)
{
  if(c == BACKSPACE){
    // if the user typed backspace, overwrite with a space.
    uartputc_sync('\b'); uartputc_sync(' '); uartputc_sync('\b');
  } else {
    uartputc_sync(c);
  }
}
```

consputc 与 consolewrite 的最大区别在于 consputc 后的字符对用户而言是可修改的，而 consolewrite 后的字符是不可修改的，在接下来实现 tab 补全时非常重要.

首先我们需要实现在用户按下 tab 时，我们的 shell 能够捕获到这个事件并且做出反应，所以我们修改 consoleintr 函数，添加对 tab 的捕获.

```
// console.c  consoleintr()
  switch(c){
  case C('P'):  // Print process list.
    procdump();
    break;
  case C('U'):  // Kill line.
    while(cons.e != cons.w &&
          cons.buf[(cons.e-1) % INPUT_BUF_SIZE] != '\n'){
      cons.e--;
      consputc(BACKSPACE);
    }
    break;
  case C('H'): // Backspace
  case '\x7f': // Delete key
    if(cons.e != cons.w){
      cons.e--;
      consputc(BACKSPACE);
    }
    break;
  case '\t': // 添加对 tab 的捕获，在用户按下 tab 时将缓存区保存并返回到用户程序
    cons.buf[cons.e++ % INPUT_BUF_SIZE] = c;
    cons.w = cons.e;
    wakeup(&cons.r);
    break;
  default:
    if(c != 0 && cons.e-cons.r < INPUT_BUF_SIZE){
      c = (c == '\r') ? '\n' : c;

      // echo back to the user.
      consputc(c);

      // store for consumption by consoleread().
      cons.buf[cons.e++ % INPUT_BUF_SIZE] = c;

      if(c == '\n' || c == C('D') || cons.e-cons.r == INPUT_BUF_SIZE){
        // wake up consoleread() if a whole line (or end-of-file)
        // has arrived.
        cons.w = cons.e;
        wakeup(&cons.r);
      }
    }
    break;
  }
```

此时我们的 console 会在用户按下 tab 时返回到 sh.c 的 getcmd 函数，我们在 getcmd 函数中添加对 tab 的处理逻辑.

```
// sh.c
int
getcmd(char *buf, int nbuf)
{
  ...
  int i, cc;
  char c;

  for(i=0; i+1 < nbuf; ){
    cc = read(0, &c, 1);
    if(cc < 1)
      break;
    if(i != 0 && c == '\t' && (buf[i-1] != '\t' )){
      autocomplete(buf , &i);
      continue;
      ...
}
```

我们在已有输入时按下 tab 后调用 autocomplete 函数，将 buf 补全成当前目录下前缀相同的文件名，同时 echoback 到 console 上(非常重要),并且还要确保 echoback 到 console 上的字符是可修改的. 于是我们在 consolewrite 函数中添加一项逻辑，如果程序调用 write 时传入的第一个字符是\t，我们就开启 autocomplete 模式，将 write 的内容模拟成用户输入 echo 到 console 上.

```
// console.c
//
// user write()s to the console go here.
//
int
consolewrite(int user_src, uint64 src, int n)
{
  int i;
  int tabcomplete = 0;
  for(i = 0; i < n; i++){
    char c;
    if(either_copyin(&c, user_src, src+i, 1) == -1)
      break;
    if(i == 0 && c == '\t'){
      tabcomplete = 1;
      continue;
    }
    if(tabcomplete){
      // 模拟用户输入
      consputc(c);
      cons.buf[cons.e++ % INPUT_BUF_SIZE] = c;
    }else{
    uartputc(c);
    }

  }

  return i;
}
```

同时添加 `writetostdin` 函数用于模拟用户输入的过程

```
// sh.c
void writetostdin(char * buf){
  char bufx[128];
  strcpy(bufx , "\t");
  strcpy(bufx+1 , buf);
  write(1, bufx , strlen(bufx));
}
```

并且封装一个 `writetofront` 函数用来把光标放在行首

```
// sh.c
void writetofront(char * buf){
    printf("\r$ ");
    writetostdin(buf);
}
```

于是我们便可以在 autocomplete 函数中调用 writetofront 函数，将补全的内容写入到 console 中(匹配前缀的部分写的非常丑陋就不放出来了，代码放在了 github 里可以自己查阅)

```
// sh.c  autocomplete
  if(nr == 1){
    //只有一个匹配项，直接补全
    char newbuf[128] ;
    strcpy(newbuf , buf);
    strcpy(newbuf + strlen(buf) , names[0] + strlen(prefix));
    writetofront(newbuf);
    strcpy(buf , newbuf);
    *index = 0;
  }else if(nr > 1){
    //有多个匹配项，打印出来
    printf("\n$ ");
    for(int i = 0 ; i < nr ; i++){
      printf("%s   " , names[i]);
    }
    printf("\n$ ");
    writeToSTDIN(buf);
    *index = 0;
  } else if (nr == 0){
    writetofront(buf);
    *index = 0;
  }
```

至此我们便还原了一个看起来有模有样的自动补全功能.
无论从什么方面来看，这个问题都不是一个 easy 的问题，本人在做的时候也查阅了非常多的资料踩了很多很多坑才非常丑陋的实现了这个功能，但可以说得上非常收益匪浅，了解了许多关于 console,terminal,tty,pty 的知识,总体来看还是非常推荐大家也去尝试一下这个问题

3. modify the shell to keep a history of passed shell commands (moderate)

在有了上一个问题的经验之后再去做这个问题就非常得心应手了，问题的核心便在于读取用户的上与下方向键，再将历史命令储存到环形数组中用于回溯.
首先是实现对上下(\e[A 与 \e[B)方向键捕获的支持，我们需要在 consoleintr 函数中添加状态机来捕获由三个字符组成的方向键输入

```
// console.c
enum {
  NORMAL = 0,
  ESCAPE = 1,
  LBRACKET = 2,
} state = NORMAL;

void
consoleintr(int c)
{
  acquire(&cons.lock);

  switch (state) {
      case NORMAL:
        if(c == '\e'){
          state = ESCAPE;
        } else{
          normal_intr(c);
        }
        break;
      case ESCAPE:
        if(c == '['){
          state = LBRACKET;
        } else{
          state = NORMAL;
        }
        break;
      case LBRACKET:
        escape_intr(c);
        state = NORMAL;
        break;
  }
  release(&cons.lock);
}

void escape_intr(int c){ //处理 \e[ 类型的终端，需要引入状态机
  switch (c) {
    case 'A': // 上
    case 'B': // 下
      // 清空缓存区
      while(cons.e != cons.w){
        cons.e --;
        consputc(BACKSPACE);
      }
      // 把上下键返回给用户终端
    cons.buf[cons.e++ % INPUT_BUF_SIZE] = '\e';
    cons.buf[cons.e++ % INPUT_BUF_SIZE] = '[';
    cons.buf[cons.e++ % INPUT_BUF_SIZE] = c;
    cons.w = cons.e;
    wakeup(&cons.r);
  }

}
```

然后在 shell 中加入环形数组用于储存历史命令

```
// sh.c
#define HISTORY_MAX  21
static char history[HISTORY_MAX][100];
static int hst;
static int hend;
static int hpos;

void add_history(const char * cmd){
  int size = hend - hst;
  strcpy(history[hend] , cmd);
  history[hend][strlen(cmd) - 1] = '\0';
  hend = (hend + 1) % HISTORY_MAX;
  hpos = hend;
  if(size == -1){
    hst = (hst + 1) % HISTORY_MAX;
  }
}
```

处理上下键的逻辑

```
// sh.c getcmd
if(c == '\e'){
      char a , b;
      read(0 , &a , 1);
      read(0 , &b , 1);
      if(a != '[')
        continue;
      if(b == 'A'){
        // up
        if(hpos == hst){
          continue;
        }
        hpos = (hpos - 1 + HISTORY_MAX) % HISTORY_MAX;
        strcpy(buf , history[hpos]);
        writeToSTDIN(buf);
        i = 0;
        continue;
      }else if (b == 'B'){
        if(hpos == hend){
          continue;
        }
        hpos = (hpos + 1) % HISTORY_MAX;
        strcpy(buf , history[hpos]);
        writeToSTDIN(buf);
        i = 0;
        continue;
      }
    }
```

main 函数中添加对历史命令的处理

```
// sh.c main
  while(getcmd(buf, sizeof(buf)) >= 0){
    if(strlen(buf) > 1)  add_history(buf);
    ...
    }
```

至此便实现了一个简单的历史命令功能.