

Obligatorisk opgave 1

Operativsystemer og C

*Bachelor in Software Development,
IT-University of Copenhagen*

Jakob Melnyk, jmel@itu.dk
Frederik Lysgaard, frly@itu.dk
Jacob Claudius Grooss, jcgr@itu.dk

October 3rd, 2012

Contents

1	Opgavebesvarelse	2
1.1	Forord	2
1.2	Funktionalitet	2
1.3	Beskrivelse af implementation	2
	Appendices	5
A	Test pictures	6
B	Makefile	10
C	bosh.c	11
D	parser.h	17
E	parser.c	18

1 Opgavebesvarelse

1.1 Forord

I denne rapport dokumenterer vi vores valg i forhold til implementation af BOSC shell (bosh). Vores implementation kan findes i "Source Code/bosh.c". Vi har bygget videre på den implementation, som blev lagt op på bloggen¹. Vi har dog erstattet "parser.c" med v2 af "parser.c"².

Kildekode og testdokumentation kan findes i appendix på side 5. Vores git repository kan findes på <https://github.com/esfdk/BOSC>.

Hvad ville vi have gjort anderledes / yderligere funktionalitet?

Del om Ulrik?

Parser håndtering af mellemrum i argumenter

1.2 Funktionalitet

Krav til funktionalitet:

1. bosh skal køre uafhængigt.
2. Display af hostname.
3. Bruger skal kunne kalde simple kommandoer. Udskriv "Command not found" meddelelse, hvis kommando ikke findes.
4. Kommandoer skal kunne køres som baggrundsprocesser.
5. Det skal være mulighed for at lave redirect af stdin og stdout.
6. Mulighed for at anvende pipes.
7. Funktionen exit skal være indbygget.
8. Cntrl+C skal afslutte det program, som kører, men ikke bosh shell'en.

Ekstra funktionalitet, vi har valgt at lave:

1. Display af "current working directory".
2. Mulighed for at kalde "cd" for at skifte directory.

1.3 Beskrivelse af implementation

1.3.1 Delopgave 1: Køre uafhængigt

Vi kalder ikke `system()` i vores kode. De mest relevante systemkald, vi laver, er `pipe()`, `dup()`, `fork()`, `waitpid()` og `execvp()`. Desuden har vi `fileno()`, `close()` og `fopen()` kald.

¹<https://blog.itu.dk/BOSC-E2012/files/2012/09/oo1.zip>

²<https://blog.itu.dk/BOSC-E2012/files/2012/09/parserc.zip>

Tests

Vi har ikke lavet nogen decideret test af dette.

1.3.2 Delopgave 2: Display hostname

Vi har valgt at vise både user og hostname i shell'en, da vi havde ønske om, at vores shell skulle ligne Bash shell så meget som muligt.

Vi ændrede 'gethostname' i bosh.c til 'get_user_and_hostname'. Metoden tager en char pointer og sætter den til at pege på en string med formatet "user@hostname". Vi tager 'user' via getenv og vi finder hostname i "/proc/sys/kernel/hostname".

Se linje 27 - 43 samt 222-227 i bosh.c.

Tests

Vi tjekker om vores shell viser det samme navn, som terminalen gør.

Forventet resultat: Se billede ?? på side 6.

Faktisk resultat: Se billede ?? på side 6.

1.3.3 Delopgave 3: Kør programmer

Vi bruger execvp() til at køre programmer i vores shell. execvp() leder efter et program i \$PATH. Hvis den finder det, bliver programmet kørt og resten af det program, som execvp blev kørt fra, bliver termineret.

Se linje 141 i bosh.c.

Tests

Test3.1 Vi kører kommandoen ls i den normale shell, og derefter i vores egen shell til sammenligning.

Forventet resultat: Se billede A.1 på side 6.

Faktisk resultat: Se billede A.1 på side 6.

Test3.2 Vi kører kommandoen cat i den normale shell, og derefter i vores egen shell til sammenligning.

Forventet resultat: Se billede A.2 på side 6.

Faktisk resultat: Se billede A.2 på side 6.

Test3.3 Vi kører kommandoen wc i den normale shell, og derefter i vores egen shell til sammenligning.

Forventet resultat: Se billede A.3 på side 7.

Faktisk resultat: Se billede A.3 på side 7.

Test3.4 Vi kører kommandoen "42" i den normale shell, og derefter i vores egen shell til sammenligning.

Forventet resultat: Se billede A.4 på side 7.

Faktisk resultat: Se billede A.4 på side 7.

1.3.4 Delopgave 4: Baggrundsprocess

Hvis et program bliver kørt som en baggrundprocess, bliver processen tilføjet til vores array af baggrundsprocessid'er. Desuden kalder vi ikke 'waitpid()' på det processid.

Se linje 172-175 samt 193-204 i bosh.c.

Vi undgår zombieprocesser ved at kalde 'signal(SIGCHLD, SIG_IGN)' (linje 213).

Tests

Vi kører 'sleep 100 &' og derefter kommandoen `date` for at vise at sleep kører i baggrunden.

Forventet resultat: Forskellen mellem de to tidspunkter fra 'date' kommandoer er mindre end 100 sekunder.

Faktisk resultat: Se billede ?? på side 8.

1.3.5 Delopgave 5: Redirect af stdin/out

Hvis der i `shellcmd` bliver redirected `stdin`, `stdout` og/eller `stderr`, så kalder vi `close()` på dem, som bliver redirected. Derefter åbner vi filen, som er blevet redirected til. Vi kalder så `dup()` på den fil.

Se fx linje 118 i `bosh.c`.

Tests

Vi kører kommandoen `'wc -l </etc/passwd >antalkontoer'` i den normale shell, og derefter i vores egen shell til sammenligning.

Forventet resultat: Se billede ?? på side 8.

Faktisk resultat: Se billede ?? på side 8.

1.3.6 Delopgave 6: Pipes

Hvis der er mere end en enkelt command i 'shellcmd', når vi modtager den som argumentet til funktionen 'shellcmd_with_pipes', piper vi vores `fd`. (Linje 97 i `bosh.c`).

Vi kalder så `close()` på write delen af vores pipe (linje 105). Hvis der er flere commands i `shellcmd`, lukker vi `stdin` og `dup()` read delen af pipe. Hvis 'write_pipe' er mere end 0, lukker vi `stdout` og kalder `dup` på vores 'write_pipe'.

Hvis vores 'write_pipe' er 0 udenfor vores fork, så kalder vi `close()` på 'write_pipe'. Hvis der er flere kommandoer tilbage i vores 'shellcmd', kalder vi 'shellcmd_with_pipes' med `fd[1]` (write delen af vores pipe) som parameter 'write_pipe'.

Tests

Test: Vi starter med at køre kommandoen `ls` i shellen, og derefter prøver vi at køre `ls — wc -w`

Forventet resultat: Se billede ?? på side 8.

Faktisk resultat: Se billede ?? på side 8.

1.3.7 Delopgave 7: Exit

Første iteration af denne delopgave terminerede bare bosh (med `exit(0)`), men dette ændrede vi, da følte det gav bedre mening, hvis vi lod `main`-funktionen afslutte.

'executeshellcmd' checker om den sidste kommando er "exit". Hvis den er, returner vi 1 til `main`-funktionen, som så afslutter.

Se linje 58 i `bosh.c`.

Tests

Vi starter shellen og kører `exit` kommandoen.

Forventet resultat: Shellen afsluttes.

Faktisk resultat: Se billede A.6 på side 9.

1.3.8 Delopgave 8: Ctrl+C

Vores første implementation af Ctrl+C funktionaliteten stoppede alle processer (både forgrunds- og baggrundsprocesser). Dette lavede vi dog om, så det matchede måden, som Bash håndterer Ctrl+C signals.

Vi har ikke implementeret en mulighed for at hente baggrundsprocesser frem i forgrunden, hvilket gør, at vi ikke kan afslutte baggrundsprocesser, medmindre de selv terminerer. Sådan funktionalitet bør være højt på dagsordenen, hvis man udvider funktionaliteten af BoSh.

I begyndelsen af vores main-metode, kalder vi `signal(SIGINT, int_handler)`. Dette betyder, at `int_handler` bliver kørt, hver gang bosh programmet modtager en interrupt. `int_handler` er vores interrupt handler, som lukker alle forgrundsprocesser.

Se linje 210 samt 252-263 i `bosh.c`.

Tests

Vi kører kommandoen `wc` uden parameter og trykker `ctrl-c` for at se om processen afsluttes.

Forventet resultat: `wc` stoppes efter `ctrl-c` er blevet tastet.

Faktisk resultat: Se billede A.7 på side 9.

1.3.9 Ekstra funktionalitet 1: Display current working directory

Som nævnt i **1.3.2** har vi haft et ønske om at få vores shell til at ligne Bash, så deres ville vi gøre det muligt at både se og skifte directory.

Vi bruger en funktion `'getcwd'` til at finde current working directory. Dette har vi implementeret i `'getcurrentdir'`.

Se linje (47-53) i `bosh.c`.

Tests

Vi starter shellen for at se om der vises det directory der i øjeblikket arbejdes i.

Forventet resultat: Se billede A.8 på side 9.

Faktisk resultat: Se billede A.8 på side 9.

1.3.10 Ekstra funktionalitet 2: Change directory

Som nævnt i **1.3.9**, så ville vi gøre det muligt at både se og skifte directory, hvilket var grundlaget for at implementere dette.

I vores `'executeshellcmd'` tjekker vi om kommandoen er `"cd"` om den har mere end et argument. I så fald så kalder vi `chdir` på det første argument til `"cd"` kommandoen. Hvis det giver en error, printer vi det og returner 0.

Se linje 62-79 i `bosh.c`.

Tests

Vi går ind i vores shell og prøver derefter at skifte til andet directory end det der arbejdes i.

Forventet resultat: Se billede A.9 på side 9.

Faktisk resultat: Se billede A.9 på side 9.

A Test pictures

A.1 Delopgave 2

```
root@bosc:~/bosc/oo1# ./bosh
root@bosc:/root/bosc/oo1# _
```

A.2 Delopgave 3

Figure A.1: Test 1

```
root@bosc:~/bosc/oo1# ls
bosh      bosh.o    parser.c  parser.o  print.h
bosh.c    Makefile  parser.h  print.c   print.o
root@bosc:~/bosc/oo1# ./bosh
root@bosc:/root/bosc/oo1# ls
bosh      bosh.o    parser.c  parser.o  print.h
bosh.c    Makefile  parser.h  print.c   print.o
root@bosc:/root/bosc/oo1# _
```

Figure A.2: Test 2

```
root@bosc:~/bosc/oo1# cat Makefile
all: bosh
OBSJ = parser.o
LIBS= -lreadline -ltermcap
CC = gcc
bosh: bosh.o ${OBSJ}
        ${CC} -o $@ ${LIBS} bosh.o ${OBSJ}
clean:
        rm -rf *o bosh
root@bosc:~/bosc/oo1# ./bosh
root@bosc:/root/bosc/oo1# cat Makefile
all: bosh
OBSJ = parser.o
LIBS= -lreadline -ltermcap
CC = gcc
bosh: bosh.o ${OBSJ}
        ${CC} -o $@ ${LIBS} bosh.o ${OBSJ}
clean:
        rm -rf *o bosh
```

Figure A.3: Test 3

```
root@bosc:~/bosc/oo1# wc Makefile
11 25 146 Makefile
root@bosc:~/bosc/oo1# ./bosh
root@bosc:/root/bosc/oo1# wc Makefile
11 25 146 Makefile
```

Figure A.4: Test 4

```
root@bosc:~/bosc/oo1# 42
-bash: 42: command not found
root@bosc:~/bosc/oo1# ./bosh
root@bosc:/root/bosc/oo1# 42
Could not find command: 42
```


A.3 Delopgave 4

```
root@bosc:/root/bosc/oo1# date
Tue Oct  2 13:46:00 CEST 2012
root@bosc:/root/bosc/oo1# sleep 100 &
root@bosc:/root/bosc/oo1# date
Tue Oct  2 13:46:04 CEST 2012
root@bosc:/root/bosc/oo1# _
```

A.4 Delopgave 5

```
root@bosc:~/bosc/oo1# ls
bosh      bosh.o    parser.c  parser.o  print.h
bosh.c    Makefile  parser.h  print.c   print.o
root@bosc:~/bosc/oo1# wc -l < /etc/passwd > antalkontoer
root@bosc:~/bosc/oo1# ls
antalkontoer bosh.c    Makefile  parser.h  print.c   print.o
bosh          bosh.o    parser.c  parser.o  print.h
root@bosc:~/bosc/oo1# cat antalkontoer
25
root@bosc:~/bosc/oo1# rm antalkontoer
root@bosc:~/bosc/oo1# ./bosh
root@bosc:/root/bosc/oo1# ls
bosh      bosh.o    parser.c  parser.o  print.h
bosh.c    Makefile  parser.h  print.c   print.o
root@bosc:/root/bosc/oo1# wc -l < /etc/passwd > antalkontoer
root@bosc:/root/bosc/oo1# ls
antalkontoer bosh.c    Makefile  parser.h  print.c   print.o
bosh          bosh.o    parser.c  parser.o  print.h
root@bosc:/root/bosc/oo1# cat antalkontoer
25
root@bosc:/root/bosc/oo1# rm antalkontoer
```

A.5 Delopgave 6

```
root@bosc:~/bosc/oo1# ./bosh
root@bosc:/root/bosc/oo1# ls
bosh bosh.c bosh.o Makefile parser.c parser.h parser.o
root@bosc:/root/bosc/oo1# ls | wc -w
7
root@bosc:/root/bosc/oo1# _
```

A.6 Delopgave 7

```
root@bosc:~/bosc/001# ./bosh
root@bosc:/root/bosc/001# exit
Exiting bosh.
root@bosc:~/bosc/001#
```

A.7 Delopgave 8

```
root@bosc:~/bosc/001# ./bosh
root@bosc:/root/bosc/001# wc -l
root@bosc:/root/bosc/001# ^C_
```

A.8 Delopgave 9

```
root@bosc:~/bosc/001# ./bosh
root@bosc:/root/bosc/001# _
```

A.9 Delopgave 10

```
root@bosc:~/bosc/001# ./bosh
root@bosc:/root/bosc/001# cd ..
root@bosc:/root/bosc# cd 001/
root@bosc:/root/bosc/001# cd ..
root@bosc:/root/bosc# cd /root/bosc/001/
root@bosc:/root/bosc/001# ls
bosh bosh.c bosh.o Makefile parser.c parser.h parser.o
root@bosc:/root/bosc/001# _
```

B Makefile

```
all: bosh

OBJS = parser.o
LIBS= -lreadline -ltermcap
CC = gcc

bosh: bosh.o ${OBJS}
    ${CC} -o $@ ${LIBS} bosh.o ${OBJS}

clean:
    rm -rf *.o bosh
```

C bosh.c

```
/* bosh.c : BOSC shell */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <unistd.h>
#include <readline/readline.h>
#include <readline/history.h>
#include <errno.h>
#include <signal.h>
#include "parser.h"

/* ——— symbolic constants ——— */
#define HOSTNAMEMAX 100
#define SHELL_MAX_PROCESSES 50

/* Interrupt handler */
static void int_handler(int sig);

/* pid's for the running processes */
pid_t running_foreground_processes[SHELL_MAX_PROCESSES];
pid_t running_background_processes[SHELL_MAX_PROCESSES];

/* Gets user and hostnames and inserts them into a string with the format "user@hostname".
char *get_user_and_hostname(char *hostname, size_t size)
{
    FILE *hostnamefile;
    char hname[HOSTNAMEMAX];
    char line[HOSTNAMEMAX];

    hostnamefile = fopen ("/proc/sys/kernel/hostname", "r");

    while (fgets(line, HOSTNAMEMAX, hostnamefile))
    {
        if (sscanf(line, "%s", hname))
        {
            snprintf(hostname, size, "%s@%s", getenv("USER"), hname);
        }
    }

    return hostname;
}

/* Sets dir to be the current working directory. */
```

```

char *getcurrentdir(char *dir, size_t size)
{
    char cur_path_buffer[1024];
    char *cur_path = getcwd(cur_path_buffer, sizeof(cur_path_buffer));
    snprintf(dir, size, "%s", cur_path);
    return dir;
}

/* Executes a shell command. */
int executeshellcmd (Shellcmd *shellcmd)
{
    if (!strcmp(*shellcmd->the_cmds->cmd, "exit")){
        return 1;
    }

    char **cd_command = shellcmd->the_cmds->cmd;

    if (strcmp(cd_command[0], "cd") == 0 && cd_command[1] != NULL)
    {
        const char *newdir = cd_command[1];
        int error = chdir(newdir);

        switch (error)
        {
            case 0: break;
            case EACCES:
                printf("%s: access denied.\n", newdir);
                break;
            case ENOENT:
                printf("%s: no such file or directory.\n", newdir);
                break;
            case ENOTDIR:
                printf("%s: not a directory.\n", newdir);
                break;
            default:
                printf("%s: error %i\n", newdir, error);
                break;
        }

        return 0;
    }

    shell_cmd_with_pipes(shellcmd, 0);

    return 0;
}

/* Main function for executing shell commands. */
int shell_cmd_with_pipes(Shellcmd *shellcmd, int write_pipe)
{
    int fd[2];

```

```
char **cmd = shellcmd->the_cmds->cmd;
shellcmd->the_cmds = shellcmd->the_cmds->next;
int proc_pid;

/* Pipes if the shellcommand has more than one command. */
if(shellcmd->the_cmds != NULL)
{
    pipe(fd);
}

/* Fork process - if parent, just continue. If child, run if.*/
if(!(proc_pid = fork()))
{
    if(shellcmd->the_cmds != NULL)
    {
        close(fd[1]);
    }

    /* If any commands are left, close stdin and dup pipe.
       Else if stdin is redirected, close stdin and dup the
       stdin file. */
    if(shellcmd->the_cmds)
    {
        close(fileno(stdin));
        dup(fd[0]);
    } else if(shellcmd->rd_stdin)
    {
        close(fileno(stdin));
        dup(fileno(fopen(shellcmd->rd_stdin, "r")));
    }

    /* If write_pipe is more than 0, close stdin and dup write_pipe.
       Else if stdout is redirected, close stdout and dup the stdout
       file. */
    if(write_pipe > 0)
    {
        close(fileno(stdout));
        dup(write_pipe);
    } else if(shellcmd->rd_stdout)
    {
        close(fileno(stdout));
        dup(fileno(fopen(shellcmd->rd_stdout, "w+")));
    }

    /* If stderr is redirected, close stderr and dup redirected stderr.*/
    if(shellcmd->rd_stderr)
    {
        close(fileno(stderr));
        dup(fileno(fopen(shellcmd->rd_stderr, "w+")));
    }
}
```

```

        execvp(cmd[0], cmd);
        printf("Could not find command: %s \n", cmd[0]);
    }

    /* If background process, add proc_pid to list of background
       processes, else add to foreground processes. */
    if(shellcmd->background)
    {
        add_background_process(proc_pid);
    }
    else
    {
        add_foreground_process(proc_pid);
    }

    /* Close write_pipe if it is more than zero. */
    if (write_pipe > 0)
    {
        close(write_pipe);
    }

    /* If more commands are left, close read and run recursively. */
    if(shellcmd->the_cmds != NULL)
    {
        close(fd[0]);
        shell_cmd_with_pipes(shellcmd, fd[1]);
    }

    int exit_code;
    /* If shell command is not marked as background, wait for pid. Else
       do not wait.
    if(!shellcmd->background)
    {
        waitpid(proc_pid, &exit_code, 0);
    }
}

/* Adds a process to list of foreground processes. */
int add_foreground_process(pid_t process)
{
    int i = 0;
    pid_t *proc;
    while(*(proc = &(running_foreground_processes[i])) > 0
        && i < SHELL_MAX_PROCESSES) i++;

    /* TODO: Should introduce handling too many processes. */

    *proc = process;

    return 0;
}

```

```
/* Adds a process to list of background processes. */
int add_background_process(pid_t process)
{
    int i = 0;
    pid_t *proc;
    while(*(proc = &(running_background_processes[i])) > 0
          && i < SHELL_MAX_PROCESSES) i++;

    /* TODO: Should introduce handling too many processes. */

    *proc = process;

    return 0;
}

/* — main loop of the simple shell — */
int main(int argc, char* argv[]) {

    /* Handles Cntrl+c input. */
    signal(SIGINT, int_handler);

    /* Handles zombie processes. */
    signal(SIGCHLD, SIG_IGN);

    /* initialize the shell */
    char *cmdline;
    char hostname[HOSTNAMEMAX];
    char currentdir[1024];
    int terminate = 0;
    Shellcmd shellcmd;

    if (get_user_and_hostname(hostname, sizeof(hostname)))
    {
        /* parse commands until exit or ctrl-c */
        while (!terminate)
        {
            printf("%s:", hostname);
            printf("%s", getcurrentdir(currentdir, sizeof(currentdir)));
            if (cmdline = readline("# "))
            {
                if(*cmdline)
                {
                    add_history(cmdline);
                    if (parsecommand(cmdline, &shellcmd))
                    {
                        terminate = executeshellcmd(&shellcmd);
                    }
                }

                free(cmdline);
            }
        }
    }
}
```



```
        }
        else terminate = 1;
    }

    printf(" Exiting bosh.\n");
}

return EXIT_SUCCESS;
}

/* Interrupt handler. Closes all running foreground processes. */
void int_handler(int sig)
{
    sig = 0;
    int i;
    pid_t *process;

    for (i = 0; *(process = &running_foreground_processes[i]) > 0
        && i < SHELL_MAX_PROCESSES; i++)
    {
        kill(*process, SIGINT);
        *process = 0;
    }
}
```

D parser.h

```
typedef struct _cmd {
    char **cmd;
    struct _cmd *next;
} Cmd;

typedef struct _shellcmd {
    Cmd *the_cmds;
    char *rd_stdin;
    char *rd_stdout;
    char *rd_stderr;
    int background;
} Shellcmd;

extern void init( void );
extern int parse ( char *, Shellcmd *);
extern int nexttoken( char *, char **);
extern int acmd( char *, Cmd **);
extern int isidentifier( char * );
```

E parser.c

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "parser.h"

/* — symbolic constants — */
#define COMMANDMAX 20
#define BUFFERMAX 256
#define PBUFFERMAX 50
#define PIPE ('|')
#define BG ('&')
#define RIN ('<')
#define RUT ('>')
#define IDCHARS "-./~+"

/* — symbolic macros — */
#define ispipe(c) ((c) == PIPE)
#define isbg(c) ((c) == BG)
#define isrin(c) ((c) == RIN)
#define isrut(c) ((c) == RUT)
#define isspec(c) (ispipe(c) || isbg(c) || isrin(c) || isrut(c))

/* — static memory allocation — */
static Cmd cmdbuf[COMMANDMAX], *cmds;
static char cbuf[BUFFERMAX], *cp;
static char *pbuf[PBUFFERMAX], **pp;

/*
 * parse : A simple commandline parser.
 */

/* — parse the commandline and build shell commmand structure — */
int parsecommand(char *cmdline, Shellcmd *shellcmd)
{
    int i, n;
    Cmd *cmd0;

    char *t = cmdline;
    char *tok;

    // Initialize list
    for (i = 0; i < COMMANDMAX-1; i++) cmdbuf[i].next = &cmdbuf[i+1];

    cmdbuf[COMMANDMAX-1].next = NULL;
    cmds = cmdbuf;
    cp = cbuf;
```

```
pp = pbuf;

shellcmd->rd_stdin    = NULL;
shellcmd->rd_stdout    = NULL;
shellcmd->rd_stderr    = NULL;
shellcmd->background = 0; // false
shellcmd->the_cmds     = NULL;

do {
    if ((n = acmd(t, &cmd0)) <= 0)
        return -1;
    t += n;

    cmd0->next = shellcmd->the_cmds;
    shellcmd->the_cmds = cmd0;

    int newtoken = 1;
    while (newtoken) {
        n = nexttoken(t, &tok);
        if (n == 0)
        {
            return 1;
        }
        t += n;

        switch(*tok) {
        case PIPE:
            newtoken = 0;
            break;
        case BG:
            n = nexttoken(t, &tok);
            if (n == 0)
            {
                shellcmd->background = 1;
                return 1;
            }
        else
        {
            fprintf(stderr, "illegal bakgrounding\n");
            return -1;
        }
        newtoken = 0;
        break;
        case RIN:
            if (shellcmd->rd_stdin != NULL)
            {
                fprintf(stderr, "duplicate redirection of stdin\n");
                return -1;
            }
            if ((n = nexttoken(t, &(shellcmd->rd_stdin))) < 0)
                return -1;
```

```

        if (!isidentifier(shellcmd->rd_stdin))
        {
            fprintf(stderr, "Illegal filename: \"%s\"\n", shellcmd->rd_stdin);
            return -1;
        }
        t += n;
        break;
    case RUT:
        if (shellcmd->rd_stdout != NULL)
        {
            fprintf(stderr, "duplicate redirection of stdout\n");
            return -1;
        }
        if ((n = nexttoken(t, &(shellcmd->rd_stdout))) < 0)
            return -1;
        if (!isidentifier(shellcmd->rd_stdout))
        {
            fprintf(stderr, "Illegal filename: \"%s\"\n", shellcmd->rd_stdout);
            return -1;
        }
        t += n;
        break;
    default:
        return -1;
    }
}
} while (1);
return 0;
}

int nexttoken( char *s, char **tok)
{
    char *s0 = s;
    char c;

    *tok = cp;
    while (isspace(c = *s++) && c);
    if (c == '\\0')
    {
        *cp++ = '\\0';
        return 0;
    }
    if (isspec(c))
    {
        *cp++ = c;
        *cp++ = '\\0';
    }
    else
    {
        *cp++ = c;
        do

```

```
    {
        c = *cp++ = *s++;
    } while (!isspace(c) && !isspec(c) && (c != '\0'));
    --s;
    --cp;
    *cp++ = '\0';
}
return s - s0;
}

int acmd (char *s, Cmd **cmd)
{
    char *tok;
    int n, cnt = 0;
    Cmd *cmd0 = cmds;
    cmds = cmds->next;
    cmd0->next = NULL;
    cmd0->cmd = pp;

    while (1) {
        n = nexttoken(s, &tok);
        if (n == 0 || isspec(*tok))
        {
            *cmd = cmd0;
            *pp++ = NULL;
            return cnt;
        }
        else
        {
            *pp++ = tok;
            cnt += n;
            s += n;
        }
    }
}

int isidentifier (char *s)
{
    while (*s)
    {
        char *p = strchr (IDCHARS, *s);
        if (! isalnum(*s++) && (p == NULL))
            return 0;
    }
    return 1;
}
```