Obligatorisk opgave 1

Jacob B. Cholewa, Rasmus L. Wismann & Nikolai Storr

24. september 2014

0.1 Introduktion

Denne rapport omhandler implementationen af vores shell BOSC. Specifikationerne er som følger.

1. bosh skal kunne virke uafhængigt. Du må ikke bruge andre eksisterende shells, f.eks. er det ikke tilladt at anvende et systemkald `system()` til at starte `bash`.
2. En bruger skal kunne indtaste almindelige enkeltstående kommandoer, så som `ls`, `cat` og `wc`. Hvis kommandoen ikke findes i operativ systemet skal der udskrives en "Command not found" meddelelse.
3. Kommandoer skal kunne eksekvere som baggrundsprocesser (ved brug af &) sådan at mange programmer kan køres på samme tid.
4. Der skal være indbygget funktionalitet som gør de muligt at lave redirection af stdin og stdout til filer. F.eks skal kommandoen `wc -l < /etc/passwd > antalkontoer` lave en fil "antalkontoer", der indeholder antallet af brugerkontoer.
5. Det skal være muligt at anvende pipes. F.eks. skal `ls | wc -w` udskrive antallet af filer.
6. Funktionen `exit` skal være indbygget til at afslutte shell'en.
7. Tryk på Ctrl-C skal afslutte det program, der kører i `bosh` shell'en, men ikke shell'en selv.

Følgende vil havde om hvordan vi har implementeret følgende features.

0.2 Feature 1

0.3 Feature 2

For at en bruger kan bruge kommandoerne `ls`, `cat` og `wc` skal vi kigge i de forskelle `bin` arkiver på vores ubuntu installation. Vi leder i folderne `./`, `/bin/` og `/usr/bin/`. Vi leder i førstnævnte for også at gøre det muligt at køre filer som ligger i den sti man står i. Hvis filen er blevet fundet returneres stien, hvis ikke returneres `NULL`. Hvis kommandoen `exit` blev fundet returneres char arrayet `exit`. Kodet stykket ses her under.

```
int isValidCmd(char **cmd){
   if(strncmp(*cmd,"exit",4) == 0) return -1;
   char str1[100];
   char str2[100];
   char str3[100];
```

```
    char *path1 = "/bin/";
    char *path2 = "/usr/bin/";
    char *path3 = "./";

    strcpy(str1,path1);
    strcat(str1, *cmd);
    strcpy(str2,path2);
    strcat(str2, *cmd);
    strcpy(str3,path3);
    strcat(str3, *cmd);
    return
       access ( str1, F_OK ) != -1 ||
       access ( str2, F_OK ) != -1 ||
       access ( str3, F_OK ) != -1;
}
```

0.4 Feature 3

Hvis en kommando køres med symbolet `&` skal processen startes som en baggrundsprocess. I shellcmd structen vil feltet `background` være sat til `1` (`true`) hvis processen skal køres som en baggrundsproces. Som det kan ses i kode stykket neden for venter vi derfor kun på child processen hvis `background` er `0` (`false`)

```
int executeshellcmd(Shellcmd *shellcmd){


    ... code to check shellcmd

    pid_t pid = fork();
    if(pid == 0){
         ... Code executing shellcmd
    }else{
       if(shellcmd -> background == 0){
          int wstatus = 0;
          waitpid(pid,&wstatus,0);
       }
    }
    return 0;
}
```

0.5 Feature 4

Hvis symbolet > optræder i kommandoen skal vi redirecte output til filnavnet på veste side. eg. `cmd > file`. På samme måde skal vi hvis symbolet < optræder

i kommandoen redirecte input til at være filen fra veste side. eg `cmd < file`. Dette har vi løst med følgende logik.

```
if(in == NULL && out == NULL){
            status = executecmd(cmdlist);
         }else{
            if(in != NULL && out != NULL) status =
                redirInOut(in, out, cmdlist);
            if(in != NULL) status = redirIn(in, cmdlist);
            if(out != NULL) status = redirOut(out, cmdlist);
         }
```

som bruger følgende hjælpe metoder.

```
// redirect in and out
int redirInOut(char *inFile, char *outFile, Cmd *cmdlist){

   int fidIn = open(inFile, O_RDONLY);
   int fidOut = open(outFile, O_WRONLY | O_CREAT | O_APPEND);
   close(0); close(1);
   dup(fidIn); dup(fidOut);
   int status = executecmd(cmdlist);
   close(fidIn); close(fidOut);
   if(status != 0)
      printf("execvp returned: %i, errno returned: %i 'no such
          file or directory' \n", status, errno);

}

// redirect in
int redirIn(char *inFile, Cmd *cmdlist){
   int fid = open(inFile, O_RDONLY);
   close(0); // close standard input
   dup(fid); // 'duplicate fileid', opens another input (file)
   int status = executecmd(cmdlist);
   close(fid);
   if(status != 0)
      printf("execvp returned: %i, errno returned: %i 'no such
          file or directory' \n", status, errno);


}

// redirect out
int redirOut(char *outFile, Cmd *cmdlist){
   int fid = open(outFile, O_WRONLY | O_CREAT | O_APPEND);
   close(1); // close standard output
   dup(fid); // duplicate file-descriptor

   int status = executecmd(cmdlist);
```

```
    close(fid);
    if(status != 0)
        printf("execvp returned: %i, errno returned: %i 'no such
            file or directory' \n", status, errno);

}
```

0.6 Feature 5

En pipe (skrives med symbol `|`) tager outputtet fra kommandoen på højre side og bruger det som input til kommandoen på venstre side. eg `c1 | c2 | c3` tager output fra c1 og giver til c2 som input osv. Vi har i vores shell implementeret dette på følgende måde

```
int executecmd(Cmd *cmdlist){
    int status;
    char **cmd = cmdlist -> cmd;
    Cmd *next = cmdlist -> next;
    if(next != NULL){

        int fd[2];
        pipe(fd);

        pid_t pid = fork();
        if(pid == 0){
            close(fd[0]);
            close(1);
            dup(fd[1]);
            close(fd[1]);
            status = executecmd(next);
        }else{
            close(fd[1]);
            close(0);
            dup(fd[0]);
            close(fd[0]);
            status = execvp(*cmd,cmd);

            int wstatus;
            waitpid(pid,&wstatus,0);
        }
    }else{
        status = execvp(*cmd,cmd);
    }
    return status;
}
```

0.7 Feature 6

Denne funktion er allerede forklaret som en del af feature 6.

0.8 Feature 7
