

ID2200 – Laboration 2: Enkel kommandotolk

Sam Henriksson
hensa@kth.se
910530-2773

Elvis Stansvik
stansvik@kth.se
831205-3971

4 maj 2014

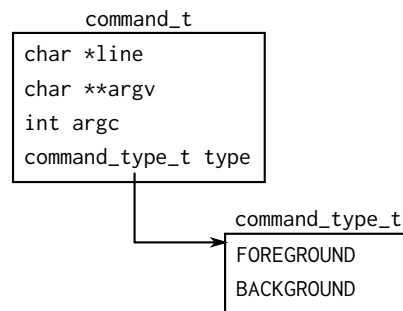
Innehåll

1	Lösning	2
2	Testkörningar	3
3	Förberedelsefrågor	4
4	Övrigt / Utvärdering	5
4.1	Tidsuppskattning	5
4.2	Betygssättning av lab-PM	5
4.3	Förslag på förbättringar	5
Appendix A: Källkod		6

1 Lösning

Uppgiften bestod av att göra en förenklad kommandotolk i vilken användaren kan starta kommandon som förgrunds- eller bakgrundsprocesser. Kommandotolken skulle även ha stöd för de inbyggda kommandona `cd` och `exit` för att byta arbetskatalog respektive avsluta tolken. Efter varje inmatat kommando skulle information om vilka bakgrundsprocesser som avslutats skrivas ut. För förgrundsprocesser skulle information om kommandots tidsåtgång skrivas ut direkt när kommandot avslutats.

Vi började med att skapa en datastruktur `command_t` för att hålla information om ett inmatat kommando. Datastrukturen håller information om kommandot som ska köras, dess parametrar samt huruvida det ska köras i en bakgrunds- eller förgrundsprocess. En illustration av datastrukturen visas i figur 1 nedan.



Figur 1: Datastrukturen `command_t`.

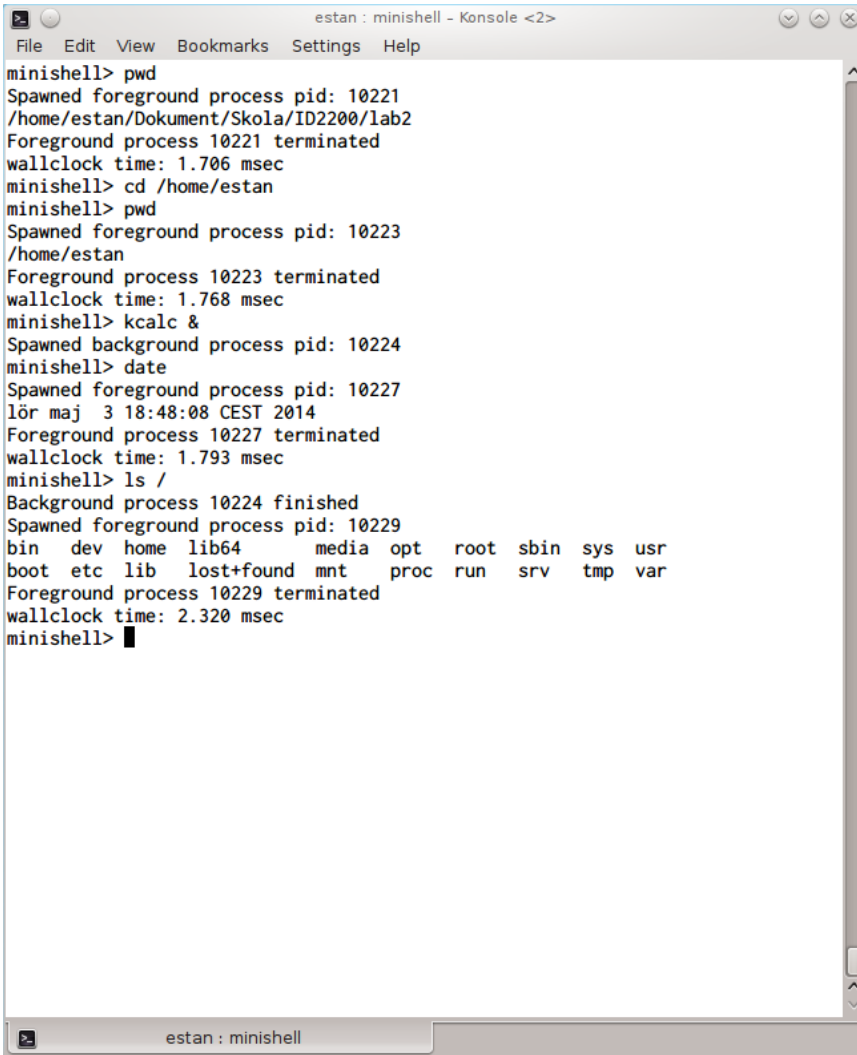
Vi skrev sedan en funktion `read_command()` som läser in en rad från användaren och utifrån denna konstruerar ett kommando som sedan returneras. För inläsningen av rader använde vi biblioteket GNU `getline()`, vilket gav oss kommandohistorik och tabbkomplettering av filnamn. Vi skrev även en funktion `free_command(..)` för att frigöra minnet som allokerats för ett kommando. Den inlästa raden allokeras på heapen av GNU `getline()`, och vi sparar därför en pekare till denna i fältet `line` i `command_t` för att kunna frigöra den senare.

I programmets huvudloop anropas först `read_command()` för att läsa in nästa kommando. Sedan kontrolleras om det finns avslutade bakgrundsprocesser genom upprepade anrop till `waitpid(2)` med flaggan `WNOHANG`. Om det inlästa kommandot är `cd` eller `exit` hanteras dessa separat genom anrop till `chdir(2)` eller genom att bryta huvudloopen. Annars skapas en barnprocess upp med `fork(2)` i vilken kommandot exekveras med `execvp(3)`. Om kommandot ska köras i förgrunden påbörjar föräldraprocessen en tidsmätning med `gettimeofday(2)` och ställer sig sedan och väntar på att barnprocessen ska avslutas med `waitpid(2)`. Om kommandot ska köras i bakgrunden skrivs bara ett meddelande om den uppstartade bakgrundsprocessen ut.

Källkoden för vår lösning finns listad i Appendix A. Den finns även att ladda hem på adressen <https://github.com/estan/ID2200/> och kan kompileras genom att skriva `make`.

2 Testkörningar

Programmet testades genom att köra `cd`, `exit`, `pwd`, `ls`, `date` och `kcalc` (grafiskt) i olika kombinationer i förgrunden respektive bakgrunden (åtföljda av "&"). Vi testade även att skriva in kommandon som inte finns.



```
estan : minishell - Konsol <2>
File Edit View Bookmarks Settings Help

minishell> pwd
Spawned foreground process pid: 10221
/home/estan/Dokument/Skola/ID2200/lab2
Foreground process 10221 terminated
wallclock time: 1.706 msec
minishell> cd /home/estan
minishell> pwd
Spawned foreground process pid: 10223
/home/estan
Foreground process 10223 terminated
wallclock time: 1.768 msec
minishell> kcalc &
Spawned background process pid: 10224
minishell> date
Spawned foreground process pid: 10227
lör maj 3 18:48:08 CEST 2014
Foreground process 10227 terminated
wallclock time: 1.793 msec
minishell> ls /
Background process 10224 finished
Spawned foreground process pid: 10229
bin dev home lib64 media opt root sbin sys usr
boot etc lib lost+found mnt proc run srv tmp var
Foreground process 10229 terminated
wallclock time: 2.320 msec
minishell> █
```

Figur 2: Utmatning från

Figur 2 ovan visar till exempel utmatningen från sekvensen `pwd`, `cd /home/estan`, `pwd`, `kcalc &`, `date`, stängning av `kcalc`-fönstret, `ls /`. Vi testade även att döda bakgrunds- och förgrundsprocesser som var igång med `kill` och `kill -9`, samt att avsluta förgrundsprocesser med `<CTRL-C>`.

3 Förberedelsefrågor

1. *Motivera varför det ofta är bra att exekvera kommandon i en separat process.*

Svar: Exekvering av en process med en funktion ur `execve`-familjen ersätter den anropande processens kod fullständigt, endast PID behålles. Dvs exekveringen fortsätter inte i den anropande processen. Om man vill göra annat vid sidan av den exekverade processen måste man därför skapa upp en separat process i vilken man kan starta exekveringen.

2. *Vad händer om man inte i kommandotolken exekverar `wait()` för en barn-process som avslutas?*

Svar: Barnprocessen kommer ligga kvar i processtabellen som en s.k. zombie-process. Processer som avslutats försvinner inte ur tabellen förrän en annan process avläser dess status.

3. *Hur skall man utläsa `SIGSEGV`?*

Svar: `SIGSEGV` levereras av operativsystemet till en process när denna försöker utföra en otillåten minnesaccess. Signalen kan fångas upp genom att installera en signalhanterare med t.ex. `sigaction(2)` på följande vis:

Källkod 1: `sigsegv.c`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <signal.h>
4
5 static void handle_sigsegv(int signum) {
6     if (signum == SIGSEGV) {
7         fprintf(stderr, "Received SIGSEGV!\n");
8         exit(1);
9     }
10 }
11
12 int main() {
13     struct sigaction sa;
14     sa.sa_handler = handle_sigsegv;
15     if (sigaction(SIGSEGV, &sa, NULL) == -1) {
16         perror("sigaction");
17     }
18     return *((int*)0); // Invalid memory reference.
19 }
```

```
$ ./sigsegv
Received SIGSEGV!
```

4. *Varför kan man inte blockera `SIGKILL`?*

Svar: Syftet med `SIGKILL`-signalen är att fungera som en sista utväg för att kunna tvinga processer som betar sig illa eller låst sig till att avslutas. Om

processer tilläts blockera/ignorera denna signal skulle detta vara omöjligt. Att **SIGKILL** inte kan blockeras kan alltså ses som en säkerhetsåtgärd för att operativsystemet, och i förlängningen användaren, ska ha full kontroll över processerna som körs på systemet.

5. *Hur skall man utläsa deklarationen `void (*disp)(int)`?*

Svar: Det är en deklaration av en pekare `disp` till en funktion som tar en `int` som enda parameter och saknar returvärde.

6. *Vilket systemanrop använder `sigset(3C)` troligtvis för att installera en signalhanterare?*

Svar: Troligtvis `sigaction(2)` och `sigprocmask(2)` som är ett kraftfullare API. `sigset(3P)` markerades som förlegad i POSIX.1-2008 och rekommenderas inte att användas i ny kod.

7. *Hur gör man för att din kommandotolk inte skall termineras då en förgrundsprocess i den termineras med `<CTRL-C>`?*

Svar: Man kan låta `minishell`-processen ignorera signalen `SIGINT` som skickas till alla förgrundsprocesser när användaren trycker `<CTRL-C>`. Vi gör detta genom att installera signalhanteraren `SIG_IGN` på rad 28 i `minishell.c` (källkod 2 i Appendix A).

8. *Förklara varför man inte har bytt "working directory" till `/home/ds/robertr` när man avslutat `miniShell:et`?*

Svar: Varje process har sin egen arbetskatalog. När `minishell` byter arbetskatalog med `chdir(3P)` är det alltså bara arbetskatalogen för `minishell`-processen och dess framtida barnprocesser som förändras. När `minishell` avslutas återgår kontrollen till processen som startade `minishell`, som har ju har kvar sin egen arbetskatalog.

4 Övrigt / Utvärdering

4.1 Tidsuppskattning

Vi uppskattar att vi lagt cirka 5 timmar på programmering och testning, och 3 timmar på rapportskrivning. Så totalt cirka 8 timmar nedlagd tid.

4.2 Betygssättning av lab-PM

Vi har inga direkta anmärkningar på lab-PM utan ger det betyget 5/5.

4.3 Förslag på förbättringar

Kanske skulle laboration 1 och 2 kunna länkas samman, så att man i laboration 2 även lägger till stöd för pipes i kommandotolken? Instruktionerna för det inbyggda kommandot `cd` bör ändras så att det fungerar som "vanligt": Använd `HOME` endast om ingen katalog angavs, skriv annars bara ut felmeddelandet från `chdir(2)`.

Appendix A: Källkod

Källkod 2: minishell.c

```
1  /**
2   * A minimalistic shell.
3   *
4   * @author Elvis Stansvik <stansvik@kth.se>
5   * @author Sam Henriksson <hensa@kth.se>
6   */
7  #include <stdlib.h>
8  #include <stdio.h>
9  #include <string.h>
10 #include <signal.h>
11 #include <sys/types.h>
12 #include <sys/wait.h>
13 #include <sys/time.h>
14 #include <unistd.h>
15
16 #include <readline/history.h>
17
18 #include "command.h"
19 #include "util.h"
20
21 int main(int argc, char *argv[]) {
22
23     /* The shell process itself ignores SIGINT. */
24     struct sigaction action;
25     action.sa_handler = SIG_IGN;
26     action.sa_flags = 0;
27     sigemptyset(&action.sa_mask);
28     if (sigaction(SIGINT, &action, NULL) == -1) {
29         perror("sigaction");
30         return EXIT_FAILURE;
31     }
32
33     while (1) {
34         // Read the next command.
35         command_t *command = read_command();
36
37         // Check for finished background processes.
38         int status;
39         pid_t pid;
40         while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
41             printf("Background process %d finished\n", pid);
42         }
43
44         if (!command)
45             continue; // Ignore empty commands.
46
47         if (match(command->argv[0], "exit")) {
48             // Handle built-in exit command.
49             if (handle_exit(command)) {
50                 free_command(command);
51                 break;
52             }
53         } else if (match(command->argv[0], "cd")) {
```

```

54         // Handle built-in cd command.
55         handle_cd(command);
56     } else {
57         // Fork a child process.
58         pid = fork();
59         if (pid == -1) {
60             perror("fork");
61         } else if (pid == 0) {
62             // Execute command.
63             execvp(command->argv[0], command->argv);
64             perror(command->argv[0]);
65             exit(EXIT_FAILURE);
66         } else {
67             if (command->type == FOREGROUND) {
68                 printf("Spawned foreground process pid: %d\n", pid);
69
70                 struct timeval t0, t1;
71
72                 // Wait for foreground process.
73                 gettimeofday(&t0, NULL);
74                 waitpid(pid, &status, 0);
75                 gettimeofday(&t1, NULL);
76
77                 printf("Foreground process %d terminated\n", pid);
78                 printf("wallclock time: %.3f msec\n", elapsed_ms(&t0, &t1));
79             } else {
80                 printf("Spawned background process pid: %d\n", pid);
81             }
82         }
83     }
84
85     free_command(command);
86 }
87
88 clear_history(); // Clear GNU readline history.
89
90 return EXIT_SUCCESS;
91 }

```

Källkod 3: command.h

```

1  /**
2   * Command data structures and functions declarations.
3   *
4   * @author Elvis Stansvik <stansvik@kth.se>
5   * @author Sam Henriksson <hensa@kth.se>
6   */
7  #ifndef COMMAND_H
8  #define COMMAND_H
9
10 /**
11  * Command types.
12  */
13  typedef enum command_type_e {
14      FOREGROUND, /**< This is a foreground command. */
15      BACKGROUND, /**< This is a background command. */
16  } command_type_t;

```



```

17
18 /**
19  * Command structure.
20  */
21 typedef struct command_s {
22     char *line;          /**< Original command line. */
23     char **argv;         /**< Command + arguments. */
24     int argc;            /**< Number elements in argv. */
25     command_type_t type; /**< Type of command. */
26 } command_t;
27
28 /**
29  * Reads a command from the user and returns it.
30  *
31  * The caller must free the returned command with free_command(). If
32  * the entered command was empty, NULL is returned.
33  */
34 command_t *read_command();
35
36 /**
37  * Frees all memory associated with @a command.
38  */
39 void free_command(command_t *command);
40
41 /**
42  * Handle the built-in command "cd".
43  *
44  * Returns 1 if successful, otherwise 0.
45  */
46 int handle_cd(command_t *command);
47
48 /**
49  * Handle the built-in command "exit".
50  *
51  * Returns 1 if successful, otherwise 0.
52  */
53 int handle_exit(command_t *command);
54
55 #endif // COMMAND_H

```

Källkod 4: command.c

```

1 /**
2  * Command functions definitions.
3  *
4  * See command.h for documentation.
5  *
6  * @author Elvis Stansvik <stansvik@kth.se>
7  * @author Sam Henriksson <hensa@kth.se>
8  */
9 #include <stdlib.h>
10 #include <stdio.h>
11 #include <unistd.h>
12
13 #include <readline/readline.h>
14 #include <readline/history.h>
15

```

```

16 #include "util.h"
17 #include "command.h"
18
19 static const char *WHITESPACE = " \t\r\n\v\f";
20
21 command_t *read_command() {
22     command_t *command = malloc(sizeof(command_t));
23
24     /* Read a line of input with GNU readline. */
25     command->line = readline("minishell> ");
26     if (is_empty(command->line)) {
27         free(command);
28         return NULL;
29     }
30
31     /* Add to GNU readline history. */
32     add_history(command->line);
33
34     /* Parse command and arguments. */
35     int argv_len = 4;
36     command->argv = malloc(sizeof(char *) * argv_len);
37     command->argv[0] = strtok(command->line, WHITESPACE);
38     command->argc = 1;
39     while ((command->argv[command->argc] = strtok(NULL, WHITESPACE))) {
40         if (argv_len < command->argc + 2) {
41             // Increase size of argv.
42             argv_len *= 2;
43             command->argv = realloc(command->argv, sizeof(char *) * argv_len);
44         }
45         ++command->argc;
46     }
47     command->argv[command->argc] = NULL;
48
49     /* Determine command type. */
50     if (command->argc > 1 && match(command->argv[command->argc - 1], "&")) {
51         // Trailing '&', so it's a background command.
52         command->argv[command->argc - 1] = NULL;
53         command->type = BACKGROUND;
54         --command->argc;
55     } else {
56         // It's a foreground command.
57         command->type = FOREGROUND;
58     }
59
60     return command;
61 }
62
63 void free_command(command_t *command) {
64     if (!command)
65         return;
66     free(command->line);
67     free(command->argv);
68     free(command);
69 }
70
71 int handle_cd(command_t *command) {

```

```

72     // Check syntax.
73     if (command->argc > 2) {
74         fprintf(stderr, "Usage: cd [<dir>]\n");
75         return 0;
76     }
77
78     // Change directory.
79     if (chdir(command->argv[1]) == -1) {
80         if (command->argc > 1)
81             perror(command->argv[1]);
82
83         // Try HOME instead.
84         fprintf(stderr, "cd: Trying HOME...\n");
85         const char *home = getenv("HOME");
86         if (!home) {
87             fprintf(stderr, "cd: HOME is not set!\n");
88             return 0;
89         } else if (chdir(home) == -1) {
90             perror(home);
91             return 0;
92         } else {
93             return 1;
94         }
95     }
96
97     return 1;
98 }
99
100 int handle_exit(command_t *command) {
101     // We just check that the syntax is OK.
102     if (command->argc != 1) {
103         fprintf(stderr, "Usage: exit\n");
104         return 0;
105     } else {
106         return 1;
107     }
108 }

```

Källkod 5: util.h

```

1  /**
2   * Utility functions declarations.
3   *
4   * @author Elvis Stansvik <stansvik@kth.se>
5   * @author Sam Henriksson <hensa@kth.se>
6   */
7  #ifndef UTIL_H
8  #define UTIL_H
9
10 #include <sys/time.h>
11
12 /**
13  * Returns 1 if @a str is empty or just whitespace, otherwise 0.
14  */
15 int is_empty(const char *str);
16
17 /**

```

```

18  * Returns 1 if @a s1 and @a s2 match each other, otherwise 0.
19  */
20  int match(const char *s1, const char *s2);
21
22  /**
23   * Returns the number of milliseconds between @a t0 and @a t1.
24   */
25  double elapsed_ms(const struct timeval *t0, const struct timeval *t1);
26
27  #endif // UTIL_H

```

Källkod 6: util.c

```

1  /**
2   * Utility functions definitions.
3   *
4   * See util.h for documentation.
5   *
6   * @author Elvis Stansvik <stansvik@kth.se>
7   * @author Sam Henriksson <hensa@kth.se>
8   */
9  #include <string.h>
10 #include <ctype.h>
11 #include <sys/time.h>
12
13 int is_empty(const char *str) {
14     if (!str) {
15         return 1;
16     }
17     while (*str) {
18         if (!isspace(*str)) {
19             return 0;
20         }
21         ++str;
22     }
23     return 1;
24 }
25
26 int match(const char *s1, const char *s2) {
27     int s1_len = strlen(s1);
28     return s1_len == strlen(s2) && strncmp(s1, s2, s1_len) == 0;
29 }
30
31 double elapsed_ms(const struct timeval *t0, const struct timeval *t1) {
32     return (t1->tv_sec * 1000 + (double)t1->tv_usec / 1000) -
33            (t0->tv_sec * 1000 + (double)t0->tv_usec / 1000);
34 }

```