Framsida för labrapport

Operativsystem, ID220 6

Period 2, läsår 20 13

Fyll i alla uppgifter!

Labnr.	Labnamn
2	Small-Shell för UNIX

Efternamn, förnamn	Personnummer	*Tydlig* datorpostadress
Cederlund, Mattias	920926-2410	mcede@kth.se

Inlämningsdatum: 2013-12-09

Kommentarer

För internt bruk

Godkänd	Komplettera	Meddelad	Datum	Signatur

Registrerad	Ny	Gammal	G/B/U

Problembeskrivning

Uppgiften bestod i att skriva en egen kommandotolk, `smallshell` i syfte att ge kunskap och förståelse för hur man använder processer i UNIX.

Till programmet ska användaren kunna mata in kommandon. Shellet ska hantera både de interna kommandona cd och exit samt externa kommandon med flera parametrar. Processer ska kunna exekveras i förgrunden eller i bakgrunden om & anges som sista parameter till kommandot.

Kommandotolken ska skriva ut information om när processer skapas och terminerar samt hur lång tid det tog att exekvera förgrundsprocesser. Programmet ska självklart kontrollera returvärden från systemanrop och ge felmeddelanden därefter och zombieprocesser får inte lämnas efter.

För att detektera när bakgrundsprocesser terminerat ska två olika metoder imlementeras: pollning och signalhanterare. Den metod som kommandotolken ska använda definieras vid kompilering genom makrot SIGNALDETECTION.

Programbeskrivning

Programmet består av en inledande fas där signalhanterare registreras med hjälp av `register_signalhandler` och av en loop för att kunna läsa av indata godtyckligt många gånger.

Det första som händer i loopen är att en promt skrivs ut och sedan väntar kommandotolken på input från användaren. När input kommer läses den av med `fgets` och man trimmar bort eventuella inledande mellanslag. Sedan delas inputen upp och parametrar till de program shellen ska exekvera sammanställs. Man undersöker här också om kommandot ska köras i en bakgrundsprocess om sista parametern är tecknet &. Är så fallet sätts en flagga som man i senare skede kontrollerar för att bestämma hur man ska avsluta child-processen på ett korrekt sätt.

Därefter kontrolleras om kommandot som matats in är ett internt kommando. Är så fallet utförs motsvarande uppgift i parent-processen. Är kommandot ett externt kommando skapas en ny process med `fork` som sedan sätts att exekvera inmatat kommando med inmatade parametrar.

Om kommandot ska köras i förgrunden skrivs information ut om att det är en förgrundsprocess som skapats, man tar en timestamp och sedan väntar man på att child-processen ska returnera genom metoden `wait_for_child`, en blockande metod som använder sig av `waitpid` för att avsluta child-processer korrekt, samt skriva ut eventuella felmeddelanden. Därefter tas en ny timestamp och man skriver ut statistik över hur lång tid processen tog att exekvera.

Ska programmet köras i bakgrunden skrivs det ut och man nollställer flaggan inför nästa kommando som användaren kan komma att mata in.

Innan en ny promt ges görs en pollning där man kontrollerar om några bakgrundsprocesser har terminerat ifall man valt att använda den metoden. Detta görs via ett anrop till `poll_background_child` som använder en ickeblockerande `waitpid` för att kontrollera om några childprocesser ändrat status. På samma sätt som `wait_for_child` ger denna funktion även felhantering.

Använder man signalhanterare för att hantera terminerande av bakgrundsprocesser kommer man lyssna på SIGCHLD och när en sådan signal påträffats kommer man även här köra `poll_background_child`. Man blockerar signalen SIGCHLD medan vi väntar på input från

användaren eftersom en inkommande signal skulle avbryta systemanropet `fgets`, vilket skulle resultera att ingen input läses in och nästa steg i programmet refererar till ogiltigt minne.

Förberedelsefrågor

1. För att kunna utnyttja parallellitet. Om man vill kunna ha bakgrundsprocesser är det det självklara valet. Intuitivt tänker jag också att man kan få problem om man vill exekvera flera externa kommandon efter varandra om du startar dem med exec i föräldraprocessen.
2. Den blir en så kallad zombie-process.
3. SIGSEGV – segmentation fault fås i allmänhet när man försöker accessa minne som man inte har rätt att accessa.
4. I så fall kan man ha processer som aldrig kan avslutas. Det hade varit väldigt fördelaktigt för virusskapare och därför är det inte tillåtet av säkerhetsskäl.
5. Det är en pekare till en funktion disp som returnerar void och tar en int som parameter.
6. sigaction(2)
7. Jag löste det så att jag fångar SIGINT med en signalhanterare och sedan skickar vidare signalen till den förgrundsprocess som ska termineras. På så sätt överlever föräldraprocessen.
8. Working directoryt ändras bara lokalt för den process som kommandot körs i. Därför när du avslutar shellet kommer inte working directory ha bytts.

Resultat från testkörningar

```
[mcede@avril lab2]$ ./smallshell
smallshell> pwd
Spawned foreground process pid: 32252
/afs/ict.kth.se/home/m/c/mcede/os/lab2
Foreground process 32252 terminated
Wallckock time: 1061 microseconds
smallshell> cd
smallshell> pwd
Spawned foreground process pid: 32253
/afs/ict.kth.se/home/m/c/mcede
Foreground process 32253 terminated
Wallckock time: 982 microseconds
smallshell>
```

```
[mcede@avril lab2]$ ./smallshell
smallshell>
smallshell>
smallshell> ls -la &
Spawned background process pid: 447
smallshell> total 26
drwxr-xr-x 2 mcede users2  2048 Dec  7 11:30 .
drwxr-xr-x 4 mcede users2  2048 Dec  7 10:25 ..
-rw-r--r-- 1 mcede users2    48 Dec  7 10:44
C:\nppdf32Log\debuglog.txt
-rwxr-xr-x 1 mcede users2 11516 Dec  7 11:30 smallshell
-rw-r--r-- 1 mcede users2  8329 Dec  7 11:11 smallshell.c
echo tihi
Spawned foreground process pid: 448
tihi
Foreground process 448 terminated
Wallckock time: 830 microseconds
Backgroundprocess 447 terminated
smallshell>
```

Väl kommenterad kod

```
/*
 *
 * NAME:
 *   smallshell - a simple shell
 *
 * SYNTAX:
 *   smallshell
 *
 * DESCRIPTION:
 *   Running smallshell will open a simple shell that handles two
 *   internal commands, cd and exit. You can also run external
 *   commands such as ls. To run a command as a background process
 *   you specify & as the last parameter.
 *
 * NOTES:
 *     Ctrl-c will cause smallshell to exit if executed when a foreground
 *     process is not running. If a foreground process is running,
 *     a interupt signal will be sent to the foreground process causing
 *     it to terminate.
 *
 */

#include <sys/types.h> /*definierar typen pid_t*/
#include <sys/wait.h> /*definierar bland annat WIFEXITED*/
#include <errno.h> /*definierar errno*/
#include <stdio.h> /*definierar bland annat stderr*/
#include <stdlib.h> /*definierar bland annat rand() och RAND_MAX*/
#include <string.h> /*definierar stringfunktioner som bl.a. strcmp och strlen*/
#include <sys/time.h> /*definierar funktioner för att hämta tid*/
#include <signal.h> /*deinierar signalnamn med mera*/
#include <unistd.h> /*definierar bl.a. fork*/

int childpid;
```

```c
/*register_signalhandler
 *
 * register_signalhandler will register a signal handler for specified
 * signal and function to run on signal
 *
 * @ signal_code - the signalcode to run handler on
 * @ *handler - the hanlder function to run on signal
 *
 * */
void register_signalhandler(int signal_code, void (*handler)(int sig)) {
	int returnvalue;
	struct sigaction signal_parameters;
	signal_parameters.sa_handler = handler;
	sigemptyset(&signal_parameters.sa_mask);
	signal_parameters.sa_flags = 0;

	returnvalue = sigaction(signal_code, &signal_parameters, (void *) 0);
	if (returnvalue == -1) {
		perror("sigaction() failed");
	}
}

/*wait_for_child
 *
 * wait_for_child will wait for a specified child process to terminate
 * and will print error messages if it terminated with an error.
 *
 * @pid - the pid to run wait with.
 *
 * */

void wait_for_child(pid_t pid) {
	int status; /*för returvärden från child-processer*/
	childpid = waitpid(pid, &status, 0);
	if (childpid == -1) {
		perror("wait() failed unexpectedly");
	}

	/*Child-processen har kört klart*/
	if (WIFEXITED(status)) {
		int child_status = WEXITSTATUS(status);
		/*Om child-processen hade problem*/
		if (child_status != 0) {
			fprintf(stderr, "Child (pid %ld) failed with exit code %d: %s\n",
					(long int) childpid, child_status,
strerror(child_status));
		}
	}
	else {
		/*Om child-processen avbröts av signal*/
		if (WIFSIGNALED(status)) {
			int child_signal = WTERMSIG(status);
			fprintf(stderr, "Child (pid %ld) was terminated by signal no. %d\n",
					(long int) childpid, child_signal);
		}
	}
}

/*poll_background_child
 *
 * poll_background_child performs a non blocking waitpid for any process
 * and will print error messages if something terminated with an error.
```

```
 *
 * */

void poll_background_child() {
	int status; /*för returväden från child-processer*/

	/*Så länge det finns childprocesser som ändrat status*/
	while ((childpid = waitpid(-1, &status, WNOHANG)) > 0) {
		if (childpid > 0) {
			/*Child-processen har kört klart*/
			if (WIFEXITED(status)) {
				int child_status = WEXITSTATUS(status);
				/*Om child-processen hade problem*/
				if (child_status != 0) {
					fprintf(stderr, "Child (pid %ld) failed with exit code
%d: %s\n",
							(long int) childpid, child_status,
strerror(child_status));
				}
				else {
					printf("Backgroundprocess %d terminated\n", childpid);
				}
			}
			else {
				/*Om child-processen avbröts av signal*/
				if (WIFSIGNALED(status)) {
					int child_signal = WTERMSIG(status);
					fprintf(stderr, "Child (pid %ld) was terminated by
signal no. %d\n",
							(long int) childpid, child_signal);
				}
			}
		}
	}
}

/*kill_child
 *
 * kill_child sends the signal SIGTERM to the current child foreground process
 *
 * */

void kill_child() {
	kill(childpid, SIGKILL);
}

int main() {
	char buffer[80]; /*Buffer till fgets*/
	char * input; /*För att spara input från fgets*/
	int returnvalue; /*Variabel för att spara returvärden från systemanrop*/
	struct timeval tv; /*Används för att hämta ut timestamps*/
	struct timezone tz; /*Används för att hämta ut timestamps med gettimeofday*/
	long starttime; /*För att spara tidpunkten då en förgrundsprocess startat*/
	long endtime; /*För att spara tidpunkten då en förgrundsprocess avslutat*/
	long diff; /*För att spara ner tidsskillnaden mellan start och sluttid*/
	int runinbackground = 0; /*För att spara om process ska köra i bakgrunden eller
inte*/

	sigset_t set; /*Set av signaler som vi ska blockera*/
	sigemptyset(&set);
	sigaddset(&set, SIGCHLD); /*Lägg till SIGCHLD till signalerna som ska blockeras*/
```

```
	/*Om vi ska använda signaldetection, registrera signalhanterare*/
#ifdef SIGNALDETECTION
	register_signalhandler(SIGCHLD, poll_background_child);
	printf("USING SIGNAL DETECTION\n");
#endif

	register_signalhandler(SIGINT, kill_child); /*Starta signalhanterare för att ta
hand om ctrl-c*/

	while (1) {
		printf("smallshell> ");

		/*Om vi använder signaldetection, blockera signaler medan inläsning av
indata*/
#ifdef SIGNALDETECTION
		returnvalue = sigprocmask(SIG_BLOCK, &set, NULL);
		if (returnvalue == -1) {
			perror("Could not block signals");
		}
#endif
		input = fgets(&buffer[0], 70, stdin); /*Läs in en rad*/
		while(input[0] == ' ') input++; /* Ta bort inledande space */
		int i = (int) strlen(input) - 1; /*Hitta radens sista index*/
		/*Om inmatningen Ã¤r annat Ã¤n tom*/
		if (i > 0) {
			input[i] = '\0'; /*Radens sista tecken, \n ska bytas ut mot \0*/
			char * args[6];
			args[0] = strtok(input, " "); /*Starta uppdelning av parametrarna
till args*/
			int k = 1;
			/*Lägger till resterande del av parametrarna till args*/
			while ((args[k] = strtok(NULL, " ")) != NULL) {
				if(strcmp(args[k], "&") == 0) {
					runinbackground = 1;
					args[k] = (char *) 0;
				}
				k++;
			}
			args[k] = (char *) 0; /*Nollställ biten efter sista parametern*/

#ifdef SIGNALDETECTION
			returnvalue = sigprocmask(SIG_UNBLOCK, &set, NULL);
			if (returnvalue == -1) {
				perror("Could not block signals");
			}
#endif

			/*Om internt kommando exit, avsluta smallshell*/
			if (strcmp(args[0], "exit") == 0) {
				printf("smallshell exiting\n");
				exit(0);
			}
			/*Om internt kommando cd, byt directory till inläst dir*/
			else if (strcmp(args[0], "cd") == 0) {
				returnvalue = chdir(args[1]);
				if (returnvalue == -1) { /*Om det misslyckas, byt till HOME,
som läses från environmentvariablerna*/
					char* home = getenv("HOME");
					returnvalue = chdir(home);
					if (returnvalue == -1) {
						printf("Could not change working directory");
					}
```

```
                        }
                    }
                    /*Annars externt kommando, skapa childprocess*/
                    else {
                        childpid = fork(); /*Forka ny child-process*/
                        if (childpid == 0) {
                            execvp(args[0], args); /*Exekvera inmatat kommando*/
                            perror("Cannot execute the given command");
                            exit(1);
                        }
                        else if (childpid == -1) { /*Om fork misslyckades, meddela
fel*/
                            perror("Cannot fork()");
                        }
                        else {
                            /*Annars, om parentprocess och child är
förgrundsprocess, vänta på child*/
                            if (runinbackground == 0) {
                                /*Läs ut starttid från gettimeofday*/
                                returnvalue = gettimeofday(&tv, &tz);
                                if (returnvalue == -1) {
                                    perror("Could not get starttime");
                                }
                                starttime = (long) tv.tv_usec;
                                printf("Spawned foreground process pid: %d\n",
childpid);
                                wait_for_child(childpid); /*Vänta på att
childprocessen ska returnera*/
                                printf("Foreground process %d terminated\n",
childpid);
                                /*Läs ut sluttid från gettimeofday*/
                                returnvalue = gettimeofday(&tv, &tz);
                                if (returnvalue == -1) {
                                    perror("Could not get starttime");
                                }
                                endtime = (long) tv.tv_usec;
                                /*Räkna ut skillnaden menllan start och sluttid
och skriv ut resultatet*/
                                diff = endtime - starttime;
                                printf("Wallckock time: %li microseconds\n",
diff);
                            }
                            else {
                                /*Om bakgrundsprocess, skriv ut info och
nollställ bakgrundsprocess-flaggan*/
                                printf("Spawned background process pid: %d\n",
childpid);
                                runinbackground = 0;
                            }
                        }
                    }
                }
                /*Om signaldetection inte används, polla efter ändrade statusar från
bakgrundsprocesser*/
#ifndef SIGNALDETECTION
                poll_background_child();
#endif
        }
        return 0;
}
```

Var källkoden finns

Källkoden finns på /afs/ict.kth.se/home/m/c/mcede/os/lab2 där programmet heter smallshell.c.

Koden kompileras med gcc -W -o smallshell smallshell.c om man vill använda sig av pollning för att kolla exitstatus på childprocesser och gcc -W -D SIGNALDETECTION -o smallshell smallshell.c om man vill använda sig av signalhanteraren.

Verksamhetsberättelse och utvärdering av laborationsuppgiften

Jag utgick från den föreslagna arbetsgången nästintill exakt när jag byggde mitt smallshell. De delar som kunde återanvändas från lab1 användes, t.ex. rutiner för att vänta på childprocesser med korrekt felhantering och kod för forkning av childprocesser. Den sista funktionaliteten jag la till var signalhanteringen, vilken också var det enda som ställde till någon form av problem. Det visade sig dock att problemen kom utifrån att jag inte hade lagt till blockering av signaler och att det var därför jag fick segmentation fault vid inläsning av kommandon efter att en bakgrundsprocess terminerat. Det var alltså något som inte kunde implementeras i flera steg, utan allt behövde vara på plats på en gång för att det skulle fungera!

Uppskattad tid lagd på laborationen: 8 timmar
Betyg för labPM: 5
Svårighetsgrad för laborationen: 3
Vad som fåtts ut av laborationen: 4
Vad som kan förbättras/kommentarer: Att denna labb för mig tog mindre tid att genomföra än föregående, även att den på kurshemsidan uppskattats vara svårare än föregående är för att den också bygger på de kunskaper som förvärvats under lab1. Den största delen av labben var trots allt hantering av child-processer, precis som lab1.