Förberedelsefrågor

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

Om man kör ett kommando i “huvudprocessen” kommer denna att avslutas när kommandot avslutas. Ofta vill man att det inte ska vara på detta vis, då lönar det sig att köra exekveringen i en separat process. Detta gäller speciellt för en kommandotolk som ju ska kunna exekvera flera kommandon efter varandra utan att själv avslutas.

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

Man riskerar att få s.k. zombie-processer. En zombie-process är en process som slutfört sin exekvering, men fortfarande har ett fält i processtabellen. Den väntar på att förälderprocessen (som “övergivit” den genom att inte köra wait()) ska läsa av dess exit-status.

3. **Hur skall man utläsa SIGSEGV?**

Segmentation violation. Det signaleras då ett program försöker referera till minne utanför det tillåtna området, vilket resulterar i det välkända meddelandet “Segmentation fault”.

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

För att man ska kunna avsluta processer ovillkorligen.

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

En funktionspekare *disp som tar en int som argument, och returnerar ingenting (void).

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

Den använder med största sannolikhet sigaction(2), vilken man som userspace-programmerare också rekommenderas att använda numera.

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

Man blockerar SIGINT i huvudprocessen, och tillåter sedan SIGINT i förgrundsprocessen (som är ett barn till huvudprocessen). På så sätt ignorerar huvudprogrammet SIGINT medan barnet avslutas direkt.

Problembeskrivning

Uppgiften gick ut på att skriva ett skal med två enkla inbyggda kommandon, *cd* och *exit*. Om något annat kommando angavs skulle programmet leta efter dessa i filsystemet, och köra det om det hittades. Skalet skulle ha stöd för både förgrunds- och bakgrundsprocesser. Statistik skulle visa hur lång tid en viss process tog på sig för att exekvera, och programmet skulle dessutom meddela när barnprocesser hade exekverats klart. Detta skulle kunna ske på två olika sätt: antingen med hjälp *wait*-pollning, eller med hjälp av signaler.

Programbeskrivning

Huvudprogrammet och nästan hela programmet körs i en loop i main. I början av loopen lyssnar vi efter kommandon från fgets, och i slutet av loopen hanterar vi dessa på lämpligt sätt, med två huvudsakliga olika villkorsförgreningar beroende på om '&' påträffades (bakgrundsprocess) efter kommandot eller ej (förgrundsprocess).

De enda funktioner som vi har definierat (utöver main) är sigchld_handler, print_exit_msg och register_sighandler, vilka beskrivs utförligare i respektive procedurkommentar.

Själva fork:nings-koden hade kunnat brytas ut till en funktion för att till det yttersta undvika kodduplicering, men påverkar så klart inte programmets beteende varför vi ansåg att det inte spelade särskilt stor roll för uppgiften. I övrigt är programmet relativt rättframt, möjligen bortsett från parsningen av användarens inmatning som är lite pillig på sina ställen.

Filkatalog

Den senaste versionen av koden finns tillgänglig via GitHub: https://github.com/joelpet/SmallShell. Observera att vi arbetade tillsammans, vilket är anledningen till den ojämna fördelningen av commits.

En klon (möjligen föråldrad) av källkodsförrådet finns att tillgå via ~joelpet/kurser/opsys/lab2/SmallShell. Användaren robertr och gruppen os bör ha både list- och läsrättigheter i den katalogen.

Utskrift med kompileringskommandon och körningar

```
$ make
gcc  smallshell.o -o smallshell
$ ./smallshell
sleep 2
==> 24612 - spawned foreground process
==> 24612 - process terminated
```

```
==> execution time: 2.001453 seconds
sleep 5 &
==> 24626 - spawned background process
sleep 3
==> 24627 - spawned foreground process
==> 24627 - process terminated
==> execution time: 3.001479 seconds
ls
==> 24629 - spawned foreground process
Makefile  smallshell  smallshell.c  smallshell.o
==> 24629 - process terminated
==> execution time: 0.001975 seconds
==> 24626 - process terminated
cd asdfg
==> ERROR: Invalid directory, sending you home...
ls
==> 24634 - spawned foreground process
… files in homedir ...
==> 24634 - process terminated
==> execution time: 0.002320 seconds
exit
```

Välkommenterad och strukturerad källkod

```
/*
 *
 * NAME:
 *   smallshell - a simple shell
 *
 * SYNTAX:
 *   smallshell
 *
 * DESCRIPTION:
 *   Smallshell reads commands from standard input to be executed, either as a
 *   foreground or background process, until user quits by typing the command
 *   `exit`.
 *
 * EXAMPLES:
 *   A shell interaction example, including external command invokations (echo,
 *   pwd, ls) and built-in commands (cd, exit).
 *
 *   $ smallshell
 *   echo Hello World!
 *   ==> 24197 - spawned foreground process
 *   Hello World!
 *   ==> execution time: 0.001101 seconds
 *   pwd
 *   ==> 24198 - spawned foreground process
 *   /home/joelpet/kth/kurser/opsys/lab2/SmallShell
 *   ==> execution time: 0.001130 seconds
 *   cd ..
```

```
 *   ls
 *   ==> 24201 - spawned foreground process
 *   SmallShell
 *   ==> execution time: 0.001894 seconds
 *   exit
 *
 * NOTES:
 *   This shell does not offer any fancy features, such as pipes. Also, the
 *   maximum length of a command string is 70 characters, and the maximum
 *   number of arguments is 5.
 *
 */

/* smallshell
 *
 * This module contains the whole smallshell program, including main() and
 * small functions for signal handling.
 */

#define _POSIX_C_SOURCE 200112

#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

/*
 * The SIGNALDETECTION macro defines whether or not signals should be used for
 * detecting terminated background processes; a non-zero value indicates signal
 * detection should be used, and 0 makes the program poll changed status
 * information instead (default).
 */
#ifndef SIGNALDETECTION
#define SIGNALDETECTION 0
#endif

/*
 * The maximum allowed command string length (70), plus two to make room for
 * new line and terminating NULL character.
 */
#define MAX_COMMAND_SIZE 72

/*
 * The maximum number of command arguments (5), plus one for the command itself.
 */
#define MAX_NO_ARGS 6

/* register_sighandler
 *
 * register_sighandler registers a signal handler, pointed out by handler, for
 * the given signal_code.
 */
void register_sighandler(
        int signal_code,       /* The signal code to register a handler for. */
        void (*handler)(int, siginfo_t*, void*)) /* Function pointer to the handler. */
{
        int return_value;      /* Stores return value from system calls. */
```

```
        struct sigaction act;   /* sigaction struct to be passed to sigaction(2). */

        /*
        * Set up the sigaction struct to use the given signal handler, no blocked
        * signals and with the SA_SIGACTION flag to let signal handler take a
        * siginfo_t pointer (among other arguments), instead of simply the signal
        * code.
        */
        act.sa_sigaction = handler;
        sigemptyset(&act.sa_mask);
        act.sa_flags = SA_SIGINFO;
        return_value = sigaction(signal_code, &act, (void *) NULL);

        if ( -1 == return_value ) {
        fprintf(stderr,"sigaction() failed\n");
        exit(1);
        }
}

/* print_exit_msg
 *
 * print_exit_msg prints an exit message, which includes the terminated
 * process' ID.
 */
void print_exit_msg(
        pid_t child_pid)       /* ID of the terminated process. */
{
        printf("==> %d - process terminated\n", (int)child_pid);
}

/* sigchld_handler
 *
 * sigchld_handler is installed as a signal handler for SIGCHLD. It tries to
 * wait for the terminated process without blocking, to make sure we do not
 * leave any zombie processes behind.
 */
void sigchld_handler(
        int signal_code,       /* The signal code. */
        siginfo_t* siginfo, /* A singinfo_t struct pointer with valuable info. */
        void* ucontext)        /* See sigaction(2). */
{
        int status;            /* Status return value placeholder. */
        pid_t child_pid = waitpid(siginfo->si_pid, &status, WNOHANG);

        /*
        * We can end up here in case of SIGCHLD from a foreground OR background
        * process. When a fg process signals SIGCHLD, we have already waited for
        * it, which will cause waitpid(2) to return -1, so we simply ignore that
        * case. When a bg process signals SIGCHLD we have to make sure that it has
        * really changed state, that is, waitpid(2) returns the process ID of the
        * child.
        *
        * Summary: The following conditional code block will only be executed when
        * a background child process has terminated.
        */
        if (child_pid == siginfo->si_pid) {
        print_exit_msg(siginfo->si_pid);
        }
}

/* main
 *
```

```
 * main handles the shell program flow and returns 0 on exit.
 */
int main () {
	pid_t child_pid;	/* Keep track of child process while forking. */
	long long timediff; /* Helps us time processes. */
	int FOREGROUND;		/* Foreground or background process branch? */
	char command[MAX_COMMAND_SIZE];		/* User-specified command. */
	char * command_args[MAX_COMMAND_SIZE];  /* Command arguments. */
	char * pchr;		/* char pointer used for string parsing. */
	int param_index = 0;	/* Holds which parameter we're at while parsing,
				   and after parsing is done equals the number of
				   parameters. */
	int child_status;   /* Keep track of child process status. */
	struct timeval time1,time2;		/* Structs used to track time. */
	int return_value;   /* Temp variable to save return values of system calls. */
	char * return_value_ptr; /* Temp variable to save return value from fgets. */

	/*
	* Ignore sig-interrupt (Ctrl-C) in the parent (this) process.
	*/
	signal(SIGINT,SIG_IGN);

	/*
	* Install handler for SIGCHLD if needed.
	*/
#if SIGNALDETECTION
	register_sighandler(SIGCHLD, sigchld_handler);
#endif

	/* main program loop */
	for (;;) {

	/* get command string */
	return_value_ptr = fgets(command,MAX_COMMAND_SIZE,stdin);

	/* if EINTR, no nothing */
	if (return_value_ptr == NULL && errno == EINTR)
		continue;

	/* empty line (only newline), do nothing */
	if (strlen(command) == 1)
		continue;

	/* if last char is '&', set FOREGROUND to 0, otherwise set it to 1 */
	if (strlen(command) >= 2 && command[strlen(command)-2] == '&') {
		/* replace '&' with '\0' */
		command[strlen(command)-2] = '\0';
		FOREGROUND = 0;
	} else {
		FOREGROUND = 1;
	}

	/* replace newline character with null terminator */
	command[strlen(command)-1] = '\0';

	/* check for exit command */
	if (!strcmp(command,"exit"))
		break;

	/* split command string at spaces */
	pchr = strtok(command," ");
```

```
/* first parameter is always the program itself */
param_index = 0;
command_args[param_index] = pchr;
++param_index;

/* go through and save parameters into string array */
while (pchr != NULL) {
        pchr = strtok(NULL," ");
        command_args[param_index] = pchr;
        ++param_index;
}

/* check for cd command */
if (!strncmp(command,"cd",2)) {
        /* cd always takes 3 parameters ("cd cd example/directory/") */
        if (param_index != 3) {
        printf("==> ERROR: Invalid argument count to cd!\n");
        continue;
        }

        /* check the return value from chdir */
        return_value = chdir(command_args[1]);
        if (return_value == -1) {

        /* the directory is not valid */
        if (ENOTDIR) {
                printf("==> ERROR: Invalid directory, sending you home...\n");
                chdir(getenv("HOME"));

        /* another error, unknown... */
        } else {
                printf("==> ERROR: Couldn't change directory to %s\n",command_args[1]);
        }
        }

        /* don't run rest of loop */
        continue;
}

/* check no. of arguments */
if (param_index > MAX_NO_ARGS)
        fprintf(stderr,"Error: Too many arguments!");

/* executed for foreground process execution */
if (FOREGROUND) {
        child_pid = fork();
        if (child_pid == 0) {
        /* unignore sig-interrupt in the child process (Ctrl-C) */
        signal(SIGINT,SIG_DFL);

        /* execute command with specified arguments */
        (void)execvp(command,command_args);

        printf("==> ERROR: Could not execute command: %s\n",command);
        exit(1);
        } else if (child_pid == -1) {
        /* couldn't fork, something is wrong */
        fprintf(stderr,"==> ERROR: Couldn't fork!");
        exit(1);
        }

        /* this is executed by the parent process, we want to wait for
```

```
                * the child process to finish execution, and wait for its finished
                * execution */

                /* get first time measurement */
                gettimeofday(&time1,NULL);

                printf("==> %d - spawned foreground process\n",(int)child_pid);

                /* wait for child to finish */
                child_pid = waitpid(child_pid,&child_status,0);

                /* get second time measurement */
                gettimeofday(&time2,NULL);

                /* calculate time difference in microseconds */
                timediff = ((time2.tv_sec*1e6+time2.tv_usec)-(time1.tv_sec*1e6+time1.tv_usec));

                print_exit_msg(child_pid);

                printf("==> execution time: %lf seconds\n",timediff/1e6);

        } else { /* background process execution */
                child_pid = fork();
                if (child_pid == 0) {
                /* unignore sig-interrupt in child (Ctrl-C) */
                signal(SIGINT,SIG_DFL);

                /* execute command with specified arguments */
                (void)execvp(command,command_args);

                printf("==> ERROR: Could not execute command: %s\n",command);
                exit(1);
                } else if (child_pid == -1) {
                /* unknown forking error, shouldn't happen */
                fprintf(stderr,"==> ERROR: Couldn't fork!"); exit(1);
                }

                /* this is executed by the parent process */
                printf("==> %d - spawned background process\n",(int)child_pid);

        }

        /*
        * Perform polling of child process state changes to detect if any such
        * has terminated. Only performed if signal detection is not activated.
        */
#if !SIGNALDETECTION
        for (;;) {
                /* Asynchronously check if any process has terminated. */
                child_pid = waitpid(-1,&child_status,WNOHANG);

                /* if nothing has changed, or if error, break the loop */
                if (child_pid == 0 || child_pid == -1) {
                break;
                }

                /* here, a process has terminated */
                print_exit_msg(child_pid);
        }
#endif

        }
```

```
        return 0;
}
```

Verksamhetsberättelse och synpunkter på laborationen

Arbetet genomfördes tillsammans, vid samma dator. Vi började med att implementera en enkel fork med en exekvering. Sedan implementerades parsning av kommandorad. Därefter implementerades att huvudprocessen väntade på barnet, och därmed var implementationen av förgrundsprocesser i stort sett klar.

Sedan lades stöd för bakgrundsprocesser till, och samtidigt implementerade vi flaggan SIGNALDETECTION. En märklig bugg uppträdde, som visade sig bero på att *fgets* avbröts av signalen SIGCHLD, vilket ledde till oförutsägbart beteende. Detta kunde dock enkelt lösas med en koll om läsningen gick rätt till.

Ett annat problem uppstod då raden `struct sigaction signal_parameters` lades till i koden. Då fick vi ett konstigt felmeddelande angående en storlek på en struktur. Det löstes genom att vi placerade `#define _POSIX_C_SOURCE 200112` i början av koden. I övrigt har labben gått bra.

Varför ska användaren skickas till hemkatalogen när en felaktig katalog anges? Vi tycker att detta är lite konstigt, även om det var en bra övning.

Arbetet tog totalt cirka 6 timmar, och var fördelat över två dagar. Laborationen gav en bra inblick i signaler och processer, som vi hoppas ha god användning för i framtiden.