Lab report, ID2200, Operating Systems,
Spring Semester, 2015

Josselin Vallee, 931103-T034
josselin@kth.se
Martin Andersson, 660918-0150
fager9hu@kth.se

I.Problem description

The assignment has been to design and code an application which replicates a shell/CLI-environment. Many features have been added as specified, execution of foreground processes, background processes, return of information on these on inception and completion, PID, elapsed time in the case of foreground processes, several built-in commands (more on these below) and protection against interruption through <Ctrl-C>.

Details will be illuminated further under headings “Program description” and “Printout of compile commands and instances of execution”.

II.Program description

The outermost control structure of the application is the loop realized by the function *shell_loop()*. It executes on the value of the variable *run ,* a value of 1 keeps the loop going with execution of the function *reapBgProcesses()* that will reap terminated background zombie processes according to one of two methods, chosen at compilation time. According to specifications, we disable a relevant set of disrupting signals (e.g CTRL-C, CTRL-\ etc...).

Function *reapBgprocesses()* is recursive in nature when using the polling method, and keeps recurring on itself as long as new terminated background processes are found by the wait system call. Note also that depending on how the user chooses to compile the application, *reapBgprocesses()* either is a signal handler based function (with SIGDET=1 on the command line) or a polling based function. The second choice (with signals) only introduces a *linkTo Handler()* function that minds the setting up of the connection between signal (SIGCHLD in the case of waiting on background processes with a signal handler) and the desired signal handler. In our case the signal handler is defined in *handlerSIGCHLD()* , where we wait in a loop for any terminated child processes as we are ensured that there is at least one by the signal SIGCHLD.

After *reapBgProcesses()* , the shell prompts its standard line. Following this, user input is captured with *fgets()*.

Then, user input is processed and parsed in the *parseInput().* The main operation achieved in this function is to tokenize the command and its parameters.

Function *executeCmd()* follows and separates out regular process commands from built-in commands.

Built-in commands, *exit*, *cd*, *checkEnv*, then result in the execution of their respective

functions which should be self-explanatory, with the exception of *checkEnv* , which gets a special discourse here: *checkEnv* (see below in point III) in is a command which displays environment variables by piping the output of system call *printenv* to stdout through one of three possible pagers depending on which one is available. Potentially, with exactly one option added to the command, checkEnv also sorts variables through also adding a pipe to the string filter utility *grep* to the daisy chain.

Function *regCommand()* also separates out background process commands from foreground process commands. Background process commands are dealt with as represented above. Foreground processes are managed in function *foregroundProcess()* that is similar to *backgroundProcess()* except from the fact that termination of the child process is waited, execution time is computed and relevant signals are enabled back. It is informative to note that the solution to the problem of preventing user signals (e.g. <Ctrl-C>,<Ctrl-\>) from terminating the shell itself, namely to treat them all with the format *signal(SIGXXX,SIG_IGN)* , becomes reversed in *foregroundProcess()*, which fulfills the specification that foreground child processes forked off the shell should, by contrast, be interruptible. Interruptibility is accomplished through the format *signal(SIGXXX,SIG_DFL)*.

For further and more technical enquiries, the code is heavily commented and explicit and its inspection should answer to any supplementary interrogations.

III.The checkEnv command :

The checkEnv command constructs a daisy chain of pipes between system calls/commands in order to present the environment variables to the user. There exist two alternatives, one 3-link chain with two pipes which represents all environment variables, and one four-link chain which filters them with *grep.* Furthermore, a pager is always used before the final pipe to stdout; the pager is chosen as, in order a) the environment variable list's pager, b) *less* and c) *more.* The utility *printenv* is the data source. The choice between 3-link and 4-link is based on the value of the length of its parameter list.

The function checkEnv also produces error messages on erroneous input, e.g. two strings to filter on, which is demonstrated in the printout section below (V).

IV.File catalog:

C file containing the shell code :

/afs/kth.se/home/j/o/josselin/OS_project/shell.c

Supplementary versions of the code are also available on Bilda, along with the report.

Versions of the report are available on Bilda on both Martin Andersson and Josselin Vallee's accounts. Also a supplementary version is available at :

/afs/kth.se/home/j/o/josselin/OS_project/Lab_report.pdf

V.Printout of compile commands and executions:
All the executions were executed on shell.it.kth.se with the following compilation of the shell.c file :

[josselin@colombiana OS_project]$ gcc -pedantic -Wall -ansi -O4 -o shell.o shell.c -D SIGDET=1

The shell is launched by the command :

[josselin@colombiana OS_project]$ ./shell.o

Testing the built in commands :

- Cd :

josselin@shell > mkdir test
Started foreground process : 17310
Terminated foreground process: 17310
Elapsed command execution wallclock time: 5.000000 ms
josselin@shell > cd test
josselin@shell > pwd
Started foreground process : 17312
/afs/kth.se/home/j/o/josselin/OS_project/test
Terminated foreground process: 17312
Elapsed command execution wallclock time: 1.000000 ms

- exit :
josselin@shell > exit
Killed
[josselin@colombiana OS_project]$

-checkEnv :

josselin@shell > checkEnv SSH

SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass
SSH_CLIENT=130.237.223.55 53851 22
SSH_CONNECTION=130.237.223.55 53851 130.237.212.173 22
SSH_TTY=/dev/pts/34
(END)

-ls -a -b -c -d -f

josselin@shell > ls -a -b -c -d -f
Started foreground process : 18853
.
Terminated foreground process: 18853
Elapsed command execution wallclock time: 16.000000 ms

-sleep 10 & into sleep 7 into CTRL-C.
josselin@shell > sleep 10 &
Spawned background process : 18957.
josselin@shell > sleep 7
Started foreground process : 18962 //Type in CTRL-C
Foreground process : 18962 , did not terminate normally
Elapsed command execution wallclock time: 1040.000000 ms
josselin@shell >
Terminated background process : 18957
josselin@shell >

Here we can see that the sleep 7 has been stopped with CTRL-C after 1 second.

VI.What we have learned:

A deeper understanding of system calls and intimate C interactions with the kernel. Moreover, working with signals and inter-process communications gave us a better insight into developing deep cooperative applications with the kernel and into asynchronous phenomena.

Needless to say, we also learned a great deal about shell mechanisms and functionality.

Finally, having no previous knowledge of the C-preprocessor utility and macros, we learned core and practical new abilities about this.

VII.Outline:

The project work is a result of a successful cooperation where different skills could find their application and enable access to the final goal. Moreover, the minimalistic lab-PM directed us into independent efforts, thorough readings and personal search in order to deepen our knowledge and understanding so as to provide the best solution possible.

VIII.Source code:

```
/*
*
* NAME:
*    miniShell is a program which emulates a shell. Both background and foreground processes can be
*    played out, and are given representation in the form of output on PID, initiation and termination. * *
*    Foreground processes are timed. Background processes can be nested within each other
*    without a need for extant processes to terminate first. Other features exist, e.g. two built-in commands,
*    disabling of <Ctrl-C> and other killing commands so they don't terminate the shell itself, these commands
*    and signals are enabled back for foreground processes.
*
* SYNTAX:
*    $./shell.o
*    shell prompt is produced: user@shell >
*    shell$[legit bash command with up to eighty characters command]
*
*
* ENVIRONMENT:
*    The program, for its proper function, needs implementing so that the home directory's value is available
*    in an environment variable.
*
*/



#define _XOPEN_SOURCE

#include <math.h>
#include <string.h>
```

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


#define MAX_INPUT 80
#define MAX_PARAMS 10


/*Main loop of the shell*/
void shell_loop(void);
/*Utilitary fuction to parse the command line.*/
char **parseInput(char *);
/*method used to poll background processes for termination*/
void reapBgProcesses(void);
/*Utilitary function for command execution*/
int executeCmd(char **);
int regCommand(char **);
int foregroundProcess(char **);
int backgroundProcess(char **);
int exitCommand(void);
int cdCommand(char **);
int checkEnvCommand(char **);
/*Utilitary function to print shell line*/
void printShell(void);
void linkToHandler(int signal, void(*handler)(int));
void handlerSIGCHLD(int signal);
/*Function containing the code for CheckEnv*/
void checkEnv(char **);



int main(void){

        /*main loop of the shell*/
        shell_loop();
```

```
        return EXIT_SUCCESS;
}

/*Loop of the shell.*/
void shell_loop(void){
        /*Variable for user input.*/
        char user_input[MAX_INPUT + 1];
        /*Array storing every argument of the command*/
        char **params;
        /*status of the shell*/
        int run = 1;

        /*Loop continues while run is not zero.*/
        while(run){
                /* Ignore incoming signals that would stop the shell without exit */
                signal(SIGINT, SIG_IGN);
                signal(SIGQUIT, SIG_IGN);
                signal(SIGTSTP, SIG_IGN);
                signal(SIGTTIN, SIG_IGN);
                signal(SIGTTOU, SIG_IGN);


                /*Poll background processes.*/
                reapBgProcesses();

                /*Prompt shell message.*/
                printShell();

                /*Read input from standard input.*/
                if(fgets(user_input, MAX_INPUT, stdin)== NULL){
                   continue;
                }

                /*If enter was pressed, avoid unecessary computations. */
                if(user_input[0] == '\n'){
                        continue;
                }else{
                        /*Parse the input.*/
                        params = parseInput(user_input);
                        /*execute the command*/
                        run = executeCmd(params);
```

```
            }

            /*Clear all input at end of loop*/
            strcpy(user_input, "");
            free(params);
        }
}

char **parseInput(char *user_input){
        /*pointer to current token*/
        char *token;
        /*Variables for size and position regarding the parameter array*/
        int size = MAX_PARAMS, pos = 0;
        /*parameter array*/
        char **par = malloc(size * sizeof(char *));
        /*Remove trailing \n.*/
        strtok(user_input, "\n");
        /*First token :*/
        token = strtok(user_input, " ");

        /*Loop that goes through user input and tokenizes it*/
        while(token != NULL){
            /*Stores token in params.*/
            par[pos] = token;
            /*Increases position in params.*/
            pos += 1;

            /*Check if we have more parameters than the limit : In this case realloc
more memory.*/
            if(pos >= size){
                /*Add memory block of size MAX_PARAMS*/
                size += MAX_PARAMS;
                par = realloc(par, size * sizeof(char*));
                if (!par) {
                    fprintf(stderr, "par: reallocation error\n");
                    exit(EXIT_FAILURE);
                }
            }
            /*Recursion : obtain the next token.*/
            token = strtok(NULL, " ");
        }
        /*Insert a NULL mark at the end of the parameter array.*/
```

```
        par[pos] = NULL;
        /*Return the tokenized input.*/
        return par;
}


void reapBgProcesses(){
#if defined SIGDET && SIGDET==1
   /*Reap with signals*/
   linkToHandler(SIGCHLD, handlerSIGCHLD);
#else
         /*Function polling for terminated child processes:
         It will call itself recursively until it doesn't find any zombie processes.*/

        /*Variable for background process*/
        pid_t bg_pid;
        /*Status variable*/
        int status;
        /*Try reaping zombie processes.*/
        bg_pid = waitpid(-1, &status, WNOHANG);
        /*Didn't find any terminated process*/
        if(bg_pid < 0){
                if (errno==ECHILD){
                        ;
                }else {
                        perror("Problem occured while checcking for terminated background
processes.\n");
                        exit(1);
                }
        }else if(bg_pid > 0){ /*Found zombie process.*/
                /*Check for exit status*/
                if (WIFEXITED(status)){
                        /*Process terminated successfully : print message*/
                        printf("Terminated background process : %d \n",bg_pid);
                } else {
                        /*Process didn't terminate normally*/
                        printf("Background process : %d an error occured during
termination.\n", bg_pid);
                }
                /*recursion : while we find terminating processes, we search for more*/
                reapBgProcesses();
        }
```

```
#endif
}

/*Function that test for built-in function or regular call.*/
int executeCmd(char **par){
	/*First : Check for built-in commands.
	Exit command*/
	if(strcmp(par[0], "exit") == 0){
		return exitCommand();
	}else if(strcmp(par[0], "cd") == 0){ /*cd command*/
		return cdCommand(par);
	}else if(strcmp(par[0], "checkEnv") == 0){ /* checkEnv command*/
		return checkEnvCommand(par);
	}else{
		/*Not a built in command.*/
		return regCommand(par);
	}
}

/*Regular command*/
int regCommand(char **par){
	int last = 0;
	int bg;

	/*Go to last element.*/
	while(par[last] != NULL){
		last += 1;
	}
	/*Check if background process*/
	bg = (strcmp(par[last-1], "&") == 0);
	if(bg){
		/*Remove the &*/
		par[last-1] = NULL;
		return backgroundProcess(par);
	}else{
		return foregroundProcess(par);
	}
}

/*Foreground process*/
int foregroundProcess(char **par){
	/*Pids*/
```

```
pid_t pid, wpid;
int status;
struct timeval tStart, tFinish;
/*Time of start and finish of the process*/
time_t tStart_ms, tFinish_ms, tAfter, tBefore, tTotal;
/* Compile macro link back SIGCHLD to default handler
in order to avoid trying both reaping and waiting explicitely the process*/
#if defined SIGDET && SIGDET==1
   signal(SIGCHLD, SIG_DFL);
#endif
if(gettimeofday(&tStart, NULL) == -1){
        perror("Error occured in call of gettimeofday.\n");
        exit(1);
}

pid = fork();

if(pid == 0){/*Child process*/

   /*Enable signals to be able to stop foreground process with actions like Ctrl.C*/
   signal(SIGINT, SIG_DFL);
   signal(SIGQUIT, SIG_DFL);
   signal(SIGTSTP, SIG_DFL);
   signal(SIGTTIN, SIG_DFL);
   signal(SIGTTOU, SIG_DFL);

  printf("Started foreground process : %d \n", getpid());
  if(execvp(par[0], par) == -1){ /*Execute te command*/
        perror("Can't execute foreground process \n");
        exit(1);
   }
}else if(pid == -1){/* fork systems call failed*/
        perror("fork system call failed");
        exit(1);
}else if(pid > 0){ /*PArent process*/
   /*Disable signals as soon a we return to the parent process (shell)*/
   signal(SIGINT, SIG_IGN);
   signal(SIGQUIT, SIG_IGN);
   signal(SIGTSTP, SIG_IGN);
   signal(SIGTTIN, SIG_IGN);
   signal(SIGTTOU, SIG_IGN);
   /*Wait for the child process */
```

```
            wpid = waitpid(pid, &status, 0);
            if(wpid == -1){
                perror("Wait system call failed \n");
                exit(1);
            }
            if(WIFEXITED(status)){/*Check for termination of the child process*/
                printf("Terminated foreground process: %d \n", pid);
            }else{
                printf("Foreground process : %d , did not terminate normally \n", pid);
            }
        }
        if(gettimeofday(&tFinish, NULL) == -1){
                perror("Error occured in call of gettimeofday.\n");
                exit(1);
        }
        tStart_ms = tStart.tv_usec; /* Execution time gets calculated */
        tBefore = tStart.tv_sec;
        tFinish_ms = tFinish.tv_usec;
        tAfter = tFinish.tv_sec;
        tTotal = 1000*(tAfter-tBefore)+0.001*fabs(tStart_ms-tFinish_ms);
        printf("Elapsed command execution wallclock time: %f ms\n",(float)tTotal);

        return 1;
}

/*Background process: Executes the command without waiting for it.*/
int backgroundProcess(char **par){
        pid_t pid;
        #if defined SIGDET && SIGDET==1
            linkToHandler(SIGCHLD, handlerSIGCHLD);
        #endif

        pid = fork();
        if(pid == 0){ /*Child process : fork successful*/
            if(execvp(par[0], par) == -1){ /*execute the command*/
                perror("Cannot execute background process\n");
                exit(1);
            }
        }else if(pid == -1){ /*problem occured with the fork system call.*/
                perror("fork system call failed");
                exit(1);
        }else if (pid > 0){ /*Parent process : Background proces spawned successfully*/
```

```
                printf("Spawned background process : %d.\n", pid);
        }
        return 1;
}

/*Built-in command : exit.*/
int exitCommand(void){
        if(kill(0, SIGKILL) == -1){
           perror("Failed killing active background processes upon exiting\n");
           exit(1);
        }
        return 0;
}

/*Built-in command : cd*/
int cdCommand(char **par){
        if(chdir(par[1]) == -1){
                perror("Can't change to that directory.\n");
        }
        return 1;
}

/*Built-in command : checkEnv*/
int checkEnvCommand(char **par){
        checkEnv(par);
        return 1;
}

/*Utilitary functions*/
/*Simple shell string.*/
void printShell(){
        char* user = getenv("USER");
        printf("%s@shell > ", user);
}

/* Link a signal to handler */
void linkToHandler(int signal, void(*handler)(int signal)){
   struct sigaction settings;
   settings.sa_handler = handler;
   sigemptyset(&settings.sa_mask);
   settings.sa_flags =0;
   if(sigaction(signal, &settings, NULL) == -1){
```

```
        perror("Sigaction system call failed\n");
        exit(1);
    }
}

/*handler for the SIGCHLD signal */
void handlerSIGCHLD(int signal){
        pid_t bg_pid;
        int status;
        if(signal){}
        bg_pid = waitpid(-1, &status, WNOHANG);
        while(bg_pid > 0){
        /*Check for exit status*/
                if (WIFEXITED(status)){
                        /*Process terminated successfully : print message*/
                        printf("\nTerminated background process : %d \n",bg_pid);
                } else {
                        /*Process didn't terminate normally*/
                        printf("\nBackground process : %d an error occured during
termination.\n", bg_pid);
                }
                bg_pid = waitpid(-1, &status, WNOHANG);
        }
}

/*
 NAME:
   checkEnv is a function which daisy-chains several predetermined shell commands and
pipes
   and pipes their filtered input/output from beginning to end.

 DESCRIPTION:
   The chain of commands, with pipes, that the program replicates is either $printenv|
sort|pager
   (with no arguments) or $printenv|grep [parameters]|sort|pager (with arguments).
   The separation between these two alternatives in the code is made by a condition
   on the number of arguments to the program on the command line; argc>1 produces the
latter,
   all other numbers of arguments produce the former (the program name itself is always
input,
   so the only other possible argument number is exactly one).
   The pager is chosen from the list 1) environment variable list pager, or 2) less, or 3)
```

more, in that order.

EXAMPLES:
$ checkEnv UB (will usually produce one line of output, UBUNTU_MENUPROXY=libappmenu.so)
$ checkEnv KRR (will produce empty output, no such string will filter out)
$ checkEnv (will produce an alphabetically sorted list of all environment variables
as per current installation)
$ checkEnv L U (will produce an error message; ./digenv.o: U: No such file or directory
because shell command grep takes either only one or zero arguments,
see NOTES below)

ENVIRONMENT:
The function, for its proper function, depends on shell commands being placed in some
available directory and the path to be so set that this directory is available. This is the normal
configuration for a UNIX/Linux installation which should make the program portable.

NOTES:
The output produced deviates minimally from the output of the command/pipe series
that it replicates in that the error message, in the case of a faulty argument set,
begins with ./digenv.o as the source of the fault, instead of with grep as in the
actual command series result from the actual CLI. Faults emanating from other services
than grep, if provoked (difficult), will be similarly represented. This should be
immaterial to the user. Depending on stochastical premises, output may also go either
through the pager or end up directly in the CLI. Because the output is the same output
and the user would have had to exit the pager anyway, this too should be immaterial.
*/
```
void checkEnv(char **args){

        pid_t pid0,pid1,pid2,pid3;      /* Child process ID's */
        int fd[3][2];                   /* Vector of integer vectors to be used as pipe*/
        int status;                     /* Input to wait()-function*/
        char *pager;                    /* String to store a possible environment list pager */

        if (args[1] != NULL)                                    /* Separation condition; on true, grep is used, otherwise not */
```

```
        {
        if (-1==pipe(fd[0]))                /* Vectors are given pipe format */
        {
                exit(1);
        }
        if (-1==pipe(fd[1]))
        {
                exit(1);
        }
        if (-1==pipe(fd[2]))
        {
                exit(1);
        }
        pid0 = fork();                              /* First child process, will shift to printenv
*/
        if (-1==pid0)
        {
                exit(1);
        }
        if (0==pid0)
        {
                if (-1==dup2(fd[0][1],STDOUT_FILENO)) /* Redirection */
                {
                        exit(1);
                }
/*
 *              Format below recurs once everywhere in each process, rigorously closes
idle pipe ends
 */

                if ((-1==close(fd[0][0])) || (-1==close(fd[0][1])) || (-1==close(fd[1][0])) || (-
1==close(fd[1][1]))
                  || (close(fd[2][0])) || (close(fd[2][1])))
                {
                        exit(1);
                }
                if (-1==execlp("printenv","printenv",(char *)0))
                {
                        exit(1);
                }

        }
```

```
        pid1 = fork();                  /* Second child process, will shift to grep */
        if (-1==pid1)
        {
                exit(1);
        }
        if (0==pid1)
        {
                if ((-1==dup2(fd[0][0],STDIN_FILENO)) || (-1==dup2(fd[1]
[1],STDOUT_FILENO))) /* Redirection */
                {
                        exit(1);
                }
                if ((-1==close(fd[0][0])) || (-1==close(fd[0][1])) || (-1==close(fd[1][0])) || (-
1==close(fd[1][1]))
                  || (close(fd[2][0])) || (close(fd[2][1])))
                {
                        exit(1);
                }
                if (execvp("grep",args)<0)
                {
                        exit(1);
                }

        }
        pid2 = fork();          /* Third child process, will shift to sort */
        if (-1==pid2)
        {
                exit(1);
        }
        if ((0==pid2))
        {
                if ((-1==dup2(fd[1][0],STDIN_FILENO)) || (-1==dup2(fd[2]
[1],STDOUT_FILENO))) /* Redirection */
                {
                        exit(1);
                }
                if ((-1==close(fd[0][0])) || (-1==close(fd[0][1])) || (-1==close(fd[1][0])) || (-
1==close(fd[1][1]))
                  || (close(fd[2][0])) || (close(fd[2][1])))
                {
                        exit(1);
                }
```

```
                if (execlp("sort","sort",(char *)0)<0)
                {
                        exit(1);
                }
        }
        pid3 = fork();
        if (-1==pid3)                   /* Fourth child process, will shift to pager */
        {
                exit(1);
        }
        if (0==pid3)
        {
                if (-1==dup2(fd[2][0],STDIN_FILENO))
                {
                        exit(1);
                }
                if (-1==close(fd[0][0]) || (-1==close(fd[0][1])) || (-1==close(fd[1][0])) || (-
1==close(fd[1][1]))
                || (-1==close(fd[2][0])) || (-1==close(fd[2][1])))
                {
                        exit(1);
                }
                pager = getenv("PAGER");
/*
 * Format of if-conditions below will cover for BOTH the case that the
 * pager is found in the environment list but does not work, AND for the case that
 * it is not found at all.
 */
                if (NULL != pager)
                {
                        if (execlp(pager,pager,(char *)0)<0)
                        {
                                if (execlp("less","less",(char *)0)<0)
                                {
                                        if (execlp("more","more",(char *)0)<0)
                                        {
                                                exit(1);
                                        }
                                }
                        }
                }
                else if (execlp("less","less",(char *)0)<0)
```

```
                {
                        if (execlp("more","more",(char *)0)<0)
                        {
                                exit(1);
                        }

                }
        }
        if ((-1==close(fd[0][0])) || (-1==close(fd[0][1])) || (-1==close(fd[1][0])) || (-
1==close(fd[1][1]))
          || (-1==close(fd[2][0])) || (-1==close(fd[2][1])))
        {
                exit(1);
        }

        if (waitpid(pid0,&status,0)<0)  /* All four processes are waited on */
                exit(1);
        if (waitpid(pid1,&status,0)<0)
                exit(1);
        if (waitpid(pid2,&status,0)<0)
                exit(1);
        if (waitpid(pid3,&status,0)<0)
                exit(1);

        }
        else    /* Beginning of other leg as per separation condition, no arguments, no
grep, mutatis mutandis */
        {
                if (-1==pipe(fd[0]))
                {
                        exit(1);
                }
                if (-1==pipe(fd[1]))
                {
                        exit(1);
                }
                if (-1==pipe(fd[2]))
                {
                        exit(1);
                }
                pid0 = fork();
                if (-1==pid0)
```

```
		{
			exit(1);
		}
		if (0==pid0)
		{
		if (-1==dup2(fd[0][1],STDOUT_FILENO))
		{
			exit(1);
		}
			if ((-1==close(fd[0][0])) || (-1==close(fd[0][1])) || (-1==close(fd[1]
[0])) || (-1==close(fd[1][1]))
		  || (close(fd[2][0])) || (close(fd[2][1])))
			{
				exit(1);
			}
			if (-1==execlp("printenv","printenv",(char *)0))
			{
				exit(1);
			}

		}
		pid1 = fork();
		if (-1==pid1)
		{
			exit(1);
		}
		if ((0==pid1))
		{
			if ((-1==dup2(fd[0][0],STDIN_FILENO)) || (-1==dup2(fd[1]
[1],STDOUT_FILENO)))
			{
				exit(1);
			}
			if ((-1==close(fd[0][0])) || (-1==close(fd[0][1])) || (-1==close(fd[1]
[0])) || (-1==close(fd[1][1]))
		  || (close(fd[2][0])) || (close(fd[2][1])))
			{
				exit(1);
			}
			if (execlp("sort","sort",(char *)0)<0)
			{
			exit(1);
```

```
                }
        }
    pid2 = fork();
    if (-1==pid2)
    {
        exit(1);
    }
    if (0==pid2)
    {
        if (-1==dup2(fd[1][0],STDIN_FILENO))
        {
            exit(1);
        }
        if (-1==close(fd[0][0]) || (-1==close(fd[0][1])) || (-1==close(fd[1][0])) || (-
1==close(fd[1][1]))
        || (-1==close(fd[2][0])) || (-1==close(fd[2][1])))
        {
            exit(1);
        }
        pager = getenv("PAGER");
        if (NULL != pager)
        {
            if (execlp(pager,pager,(char *)0)<0)
            {
                if (execlp("less","less",(char *)0)<0)
                {
                    if (execlp("more","more",(char *)0)<0)
                    {
                        exit(1);
                    }
                }
            }
        }
        else if (execlp("less","less",(char *)0)<0)
        {
            if (execlp("more","more",(char *)0)<0)
            {
                exit(1);
            }

        }
    }
```

```
        if ((-1==close(fd[0][0])) || (-1==close(fd[0][1])) || (-1==close(fd[1][0])) || (-
1==close(fd[1][1]))
          || (-1==close(fd[2][0])) || (-1==close(fd[2][1])))
        {
                exit(1);
        }
        if (waitpid(pid0,&status,0)<0)
                exit(1);
        if (waitpid(pid1,&status,0)<0)
                exit(1);
        if (waitpid(pid2,&status,0)<0)
                exit(1);

        }
}
```