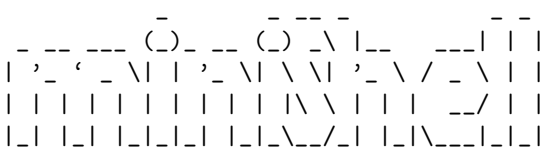
miniShell and printenv

Isac Arnekvist 860202-2017 isacar@kth.se

May 8, 2015



Summary

This describes the overall method of implementing a simple shell and a tool to list your environment variables. It took about 30-40 hours to complete and was made as two separate projects. It was a challenge to get everything to work cross platform with backwards compatibility. If done again: factor out prompt code from rest of code and try to do unit testing. Also, the use of pipes in the shell might be unnecessary.

1 Introduction

I started this task right after the course started by reading the instructions on the *ID2206* course webpage. I have not had any lab partner.

2 Task

The task is to make a simple shell and a tool to list your environment variables. The shell should be able to handle foreground and background process and time the foreground ones. The tool for listing environment variables should take a regular expression as its argument to only show those lines matching it. I initially followed the instructions from the course *ID2206*, which turned out not being the same as this course. This led to *miniShell* and *digenv* being made as two separate executables. Of course, if *digenv* is in your PATH, you can start it from anywhere using *miniShell*. The same holds for changing to the *digenv* folder and executing the local binary. For exact requirements see here.

3 Method

3.1 Input

Input was done using the library *readline*, which enables tab completion and also can enable history. The latter was never included since there was a bug when implemented and I felt that I did not want to spend time on it until everything else worked. After that, I check for `&`, remove it and then tokenize the rest of the input. The reason not tokenizing before removing it is that in bash, it works both when separated with a whitespace character and when not, and the tokenizer I use, for simplicity, splits on whitespaces. So when the input arguments are tokenized, an interpret method is invoked to check for special cases like built-in commands like *exit*. If not a special case, then forks are made and I try to execute the commands given.

3.2 Polling

This was implemented in a little peculiar fashion, the reason is that I started with timing the background processes as well. The shell always forks of a child A, which in turn forks off an other child B. Child B then executes the arguments given. The reason for this is that child A can time B and wait for it no matter if the shell returns immediately to the prompt. If it is supposed to be run as a background process, the shell doesn't wait for process A to finish and returns to the prompt while A still can wait in the background with a timer running. When B finishes, A stopped the timer and sent the data over a pipe using that as a cue of terminated processes no matter if it was a background or foreground process. I had problems when doing the same with signal detection, so to have a congruent system, timing is no longer done of background processes. When the interpret function returns, the main loop has a function to poll the pipe for finished processes. The data sent over the pipe contains the process id and the time it was executing. Since A was never waited for, an additional wait is issued to clear it off the process table. This has a (not very pretty) downside. That is that when issuing the *ps* command, both the shell and the A process shows up as ./minishell in the table. Sometimes the second one also gets the `<defunct>` tag because it had not cleared it yet before printing. This is *not* the same as leaving a zombie process though, the process always get cleaned up right away.

3.3 Signal detection

To use signal detection for background processes instead, I do not use waitpid when `&` was given. Instead, I return to the prompt and a signal handler is assigned to catch `SIGCHLD` signals. From the signal handler, I send all the processes that finished to the main loop via a pipe. The pipe is then read at an appropriate moment. There is a tricky problem here, who knows if foreground process terminated first, waitpid or the signal handler? If the signal handler acknowledges a terminated process before the waitpid, I will have an

error. This is solved by using `sighold` and `sigrelse` while a foreground process is being run. This merely delays all signal handling until waitpid got to acknowledge the foreground process. Then when the signals are turned back on again, signal handler will automatically be invoked and a loop handles any non-acknowledged processes from the process table. The problem I had here was with timing, if I time it from the signal handler, the time might be way to long of the foreground process was running for a longer time than the background processes.

3.4 cd

Most of the commands you use work out of the box, the exception is `cd`. This has to change the current working directory for the process, so if a fork is made, this change would die with that process. So how solve this? I use the system call chdir and give the second argument as argument to this call. The downside of just giving the second argument is that if I would do the following:

```
cd .. foo
```

This would lead to a change to the parent directory no matter if foo is an existing directory or not.

3.5 Ctrl-C

Since Ctrl-C should only started processes and not *miniShell* itself, I simply assigned a handler for SIGINT which only returns. Something that is different from how bash handles this is if I start a couple of sleeps in the background, and then one in the foreground, all the sleeps gets killed by the Ctrl-C.

3.6 digenv

This was solved basically by forking four times in a row and setting up three pipes to communicate between them.

```
printenv | grep | sort | pager
```

The communication is to redirect printenv's stdout to grep's stdin and so on. Some of the tricky parts was to close the pipes at the right moment. As an example, sort naturally waits for EOF before it can start sorting its input. So therefore, the pipe has to be closed before the pager pipe is closed.

4 Verification

Only manual testing was done to make sure the *miniShell* lives up to the requirements. Separate functions were tested after the introduction of each and one of them by doing manual testing, print statements and asserts. Regarding testing of the final shell, these following main themes were tested:

4.1 Simple usage

Basic existent and non-existent command was entered to make sure the behaviour is as expected. Some of the commands entered were:

```
> vim test.c
  ...
> sleep 5
  ...
> sleep 5 &
```

```
  ...
> echo hej
  ...
> ps
  ...
> foo
  ...
```

4.2 Concurrent processes

I manually did:

```
> sleep 5 &
> sleep 4 &
> sleep 6
```

with SIGSET set to both 0 and 1 to make sure that background processes that terminates while a foreground process is running behaves as expected. The expected result is that termination is not noticed until foreground process has finished. Also that all processes started give a notice about their termination. I also check with *ps* so that no *sleep* process is still there and that none remains as a *defunct* process.

4.3 digenv

This was also tested manually by first leaving the environment variable PAGER unset and testing, then setting to less and more to see that it behaved as expected. Then I tried to give arguments to see that correct lines were filtered out. If a regex was given that did not match, it should of course not show any lines at all.

4.4 Possible tests to do

A lot of more automated tests could be done if the shell was to be used over a longer time and maintained. Although, I think som major effort is going to be needed to automatically check the forks and also check process tables for expected behaviour. Especially when the "view" and "controller" is in the same file.

5 Installation and manual

Both the *miniShell* and the *digenv* comes with a compile script which compiles with flags *gcc -pedantic -Wall -ansi -O4 ...* and also a run script which first compiles and then runs the compiled binary. (Not very needed for digenv, but for symmetry...) The files are in my home directory, isacar, under `minishell/`. The digenv folder also lies within minishell folder.

6 Other

I needed maybe about 30-40 hours to complete the assignment. I think it was interesting, especially how hard it was to write code with these restrictions that does not give warnings and work on both macintosh and a linux server. If I started the project again I would have done some things different. That includes not having such a complicated fork process, since timing is not needed on background processes. I also would have split the program into a *(M)VC* pattern to allow unit testing of all "non-prompt" functions.

Now that I know that timing of background processes is not needed, I think that the use of pipes might no longer be needed. Prints could be done from the signal handler instead of sending data to the main

function. On the other hand, I am not quite sure that prints from within the signal handler is guaranteed to work all the time. I have trouble finding this in the manuals, but I feel I have seen it somewhere. It might be something with that signal handlers are asynchronous and two calls might be called at approximately the same time. So what happens if another handler is called in the middle of a print within an other handler?

7 Appendix

```
/* This is a simple shell with built-in commands cd and exit. 'digenv' is
 * assumed to be in the PATH variable since this is an external program. */
#define _XOPEN_SOURCE 500
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <signal.h>
#include <poll.h>
#include "helpers.h"
#include <readline/readline.h>
#include <readline/history.h>

#define CMD_MAX_LEN 80
#define TRUE 1
#define FALSE 0
#define READ_SIDE 0
#define WRITE_SIDE 1

#define SIGDET 1
/* If SIGDET = 1, then program will listen for signals from child processes
 * to determine that they finished running.
 * If SIGDET = 0, program will use waitpid */
#ifndef SIGDET
#define SIGDET 0
#endif

/* Signal handler for SIGCHLD */
void child_listener(int sig);
/* Catches SIGINT */
void sig_handler(int sig);

/* This struct is used to send timing stats when process finished */
typedef struct ptt {
    pid_t   pid;
    int     delta_millis;
    int     was_background;
} proc_time_t ;

/* A pipe to send proc_time_t through when processes finish */
int proc_time_pipe[2];

void interpret(char **, int);

/* Sets everything up, loop for checking input and printing stats */
int main() {
    char *input_string;
```

```
char **args;
int read_length;
/* int return_val; */
int is_background;
struct pollfd time_poll;
proc_time_t proc_time;
char prompt[] = "> ";

/* Set up timing stats pipe */
if(-1 == pipe(proc_time_pipe)) {
    perror("pipe");
    exit(-1);
}

/* Set up signal handling */
if(SIGDET) {
    if(-1 == (long)sigset(SIGCHLD, child_listener)) {
        perror("sigset");
        exit(-1);
    }
}
if(-1 == (long)sigset(SIGINT, sig_handler)) {
    perror("sigset");
}

/* Set up polling of proc_time_pipe */
proc_time.pid = 0;
proc_time.delta_millis = 0;
time_poll.fd = proc_time_pipe[READ_SIDE];
time_poll.events = POLLIN;

printf("           _       _ __ _           _ _ \n");
printf(" _ __ ___ (_)_ __ (_) _\\ |__   ___| | |\n");
printf("| '_ ` _ \\| | '_ \\| \\ \\ \\| '_ \\ / _ \\ | |\n");
printf("| | | | | | | | | | |\\ \\ | | |  __/ | |\n");
printf("|_| |_| |_|_|_| |_|_\\__/_| |_|\\___|_|_| 2.0\n");
printf("Author: Isac Arnekvist\n");

while(1) {
    /* Read input */
    input_string = NULL;
    input_string = readline(prompt);
    if(input_string == NULL) {
        read_length = -1;
    } else {
        read_length = strlen(input_string);
    }

    switch (read_length) {
        case -1:
            /* If only eof was entered or something went wrong, quietly quit
```

```
                 * (this is bash behavior) */
                exit(0);
                break;
                return 0;
            case 0:
                /* Empty line, new prompt */
                break;
            default:
                /* Handle '&' first. Might not be space separated, so easier to
                 * do here */
                if('&' == input_string[read_length-1]) {
                    /* '&' acknowledged, change to null */
                    input_string[read_length-1] = '\0';
                    is_background = TRUE;
                } else {
                    is_background = FALSE;
                }

                /* Get array of input tokens */
                args = args_tokenized(input_string);
                interpret(args, is_background);
        }

        /* 1: Check if any process sent stats over pipe after they terminated
         * 2: If it was backgrond and SIGDET = 0, clean up process table */
        while(poll(&time_poll, 1, 0)) {
            if(-1 == read(proc_time_pipe[READ_SIDE], &proc_time, sizeof(proc_time))) {
                perror("read");
                exit(-1);
            }
            /* Print stats */
            if(proc_time.was_background) {
                printf("Process %d terminated.\n", proc_time.pid);
            } else {
                printf("Process %d terminated, %d milliseconds.\n",
                        proc_time.pid,
                        proc_time.delta_millis);
            }

            if(proc_time.was_background && 0 == SIGDET) {
                /* Acknowledge the child that did timing from process table */
                wait(NULL);
            }
        }
        if(NULL != input_string) free(input_string);
    }
    return 0;
}

/* Tries to execute the given commands. If is_background is set to TRUE, the
 * prompt will immediately return. If false, prompt will return when command is
```

```
 * done executing */
void interpret(char **args, int is_background) {
    int pid;
    struct timeval start_time;
    struct timeval stop_time;
    proc_time_t proc_time;
    char *home;

    /* Check for built in commands
     * - minor bug: 'cd .. foo' is interpreted as 'cd ..' */
    if(strcmp(args[0], "cd") == 0) {

        /* cd to given directory, if fail, go to $HOME */
        if(-1 == chdir(args[1])) {
            home = getenv("HOME");
            if(-1 == chdir(home)) {
                perror("chdir");
                /* Not necessary to exit program here right? */
            }
        }

    } else if(strcmp(args[0], "exit") == 0) {

        printf("Goodbye!\n");
        /* TODO Keep track of started processes and kill any non-terminated
         * processes before quitting? */
        exit(0);

    } else {
        /* No built in command was entered, so now we try to execute the command */
        if(SIGDET) {

            /* Do not wait for background processes, let child_listener handle
             * them */

            if(!is_background) {
                /* Block any signals from background processes during
                 * foreground process */
                sighold(SIGCHLD);
            }

            pid = vfork();
            if(0 == pid) {

                /* in child, execute the command */
                execvp(args[0], args);

                /* If we get here, execlp returned -1 */
                perror("exec");
                exit(-1);
```

```
    } else {
        /* In parent */
        if(!is_background) {
            /* In parent of executing process */
            /* Start timer */
            gettimeofday(&start_time, NULL);
            if(-1 == waitpid(pid, NULL, 0)) {
                perror("waitpid");
                exit(-1);
            }

            /* Reset signal handling */
            sigrelse(SIGCHLD);

            /* Stop timer */
            gettimeofday(&stop_time, NULL);

            /* Prepare stats to send */
            proc_time.pid = pid;
            proc_time.delta_millis = (stop_time.tv_sec - start_time.tv_sec)*1000 +
                                     (stop_time.tv_usec - start_time.tv_usec)/1000;
            proc_time.was_background = is_background;

            /* Send */
            if(-1 == write(proc_time_pipe[WRITE_SIDE], &proc_time, sizeof(proc_time))) {
                perror("write"); exit(-1);
            }
        } else {
            /* Process was background and signal when terminated should be catched
             * by child_listener */
             printf("Process %d started in background\n", pid);
        }
    }

} else {
    /* Try to execute given command with polling */
    pid = fork();
    if(0 == pid) {
        /* Child of shell */
        pid = vfork();
        if(0 == pid) {

            /* in child of shells child, execute the command */
            execvp(args[0], args);

            /* If we get here, execlp returned -1 */
            perror("exec");
            exit(-1);

        } else {
            /* In parent of executing process, child of shell */
```

```
                    /* Start timer */
                    gettimeofday(&start_time, NULL);
                    waitpid(pid, NULL, 0);
                    /* Stop timer */
                    gettimeofday(&stop_time, NULL);

                    /* Prepare stats to send */
                    proc_time.pid = pid;
                    proc_time.delta_millis = (stop_time.tv_sec - start_time.tv_sec)*1000 +
                                             (stop_time.tv_usec - start_time.tv_usec)/1000;
                    proc_time.was_background = is_background;

                    /* Send */
                    if(-1 == close(proc_time_pipe[READ_SIDE])) {
                        perror("close"); exit(-1);
                    }
                    if(-1 == write(proc_time_pipe[WRITE_SIDE], &proc_time, sizeof(proc_time))) {
                        perror("write"); exit(-1);
                    }
                    if(-1 == close(proc_time_pipe[WRITE_SIDE])) {
                        perror("close"); exit(-1);
                    }

                    exit(0);
                }
            } else {
                /* If background process, process table is cleared in main()
                 * when that process finishes */
                if(!is_background) {
                    /* Wait for process if foreground */
                    waitpid(pid, NULL, 0);
                } else {
                    printf("Process %d started in background\n", pid);
                }
            }
        }
    }
    /* The array of arguments was allocated and must be freed */
    free_args(args);
}

/* Signal handler for SIGCHLD. Checks if the signal was due to a terminated
 * process. If it was, acknowledge entry in process tabel via wait() */
void child_listener(int sig) {
    pid_t pid;
    int status;
    proc_time_t proc_time;

    /* TODO report once per process in process table */
    while((pid = waitpid(-1, &status, WNOHANG)) > 1) {
        if(!WIFSIGNALED(status)) { /* TODO Fix this */
```

```
            /* Prepare stats to send */
            proc_time.pid = pid;
            proc_time.delta_millis = 0;
            proc_time.was_background = TRUE;

            /* Report! */
            if(-1 == write(proc_time_pipe[WRITE_SIDE], &proc_time, sizeof(proc_time))) {
                perror("write");
                exit(-1);
            }
        } else {
            /* Nothing to do, maybe next time!? */
            printf("Signal %d terminated process %d\n", WTERMSIG(status), pid);
        }
    }
    return;

    /* Reset signal handler, might not be needed? */
    if(SIG_ERR == signal(SIGCHLD, child_listener)) {
        perror("signal");
        exit(-1);
    }
}

/* Makes sure Ctrl-C in child process does not kill miniShell
 * at the moment also ignores if it happens in 'prompt' mode */
void sig_handler(int sig) {
    if(SIGINT == sig) {
        /* Do nothing, possible to know if sent during child process? */
    }
}

/*
 * This program lists environment variables. The results can be filtered to
 * only show lines containing parameter regex. The listing is done using 'less'
 * unless environment variable PAGER is set to another pager. If less is not
 * existing, more is used.
 *
 * Usage: digenv [parameter list]
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>

#define READ_SIDE 0
#define WRITE_SIDE 1

/*
 * Redirects listen_pipe to stdin, send_pipe to stdout
```

```
 * If an argument set to null, that argument will be ignored
 */
void setup_pipe(int *listen_pipe, int *send_pipe);

/* Does 'waitpid(pid, ...)' then checks exit status and quits if abnormal */
void checkpid(char *name, pid_t pid);

int main(int argc, char **argv, char **envp) {
    /* Declaration of all variables to be used */
    int return_value;
    int printenv_grep_pipe[2], grep_sort_pipe[2], sort_pager_pipe[2];
    int printenv_pid, grep_pid, sort_pid, pager_pid;
    int status;
    char *pager;

    /*
     * Let environment variable 'PAGER' decide how results should be listed. If
     * not set, use 'less'
     */
    pager = getenv("PAGER");
    if (pager == NULL) {
        pager = "less";
    }

    /* Set up all pipes */
    return_value = pipe(printenv_grep_pipe);
    if(return_value == -1) { return -1; }
    return_value = pipe(grep_sort_pipe);
    if(return_value == -1) { return -1; }
    return_value = pipe(sort_pager_pipe);
    if(return_value == -1) { return -1; }

    /* Create all childs */
    /* Run printenv and pipe output to grep */
    printenv_pid = fork();
    if(printenv_pid == 0) {

        /* Pipe output to grep process */
        setup_pipe(NULL, printenv_grep_pipe);

        /* Change process image */
        execlp("printenv", "printenv", NULL);

        /* We only get here if something went wrong */
        exit(0);
    }
    else if(printenv_pid == -1) {
        exit(-1);
    }

    /* Listen to printenv, grep, and pipe output to sort */
```

```
grep_pid = fork();
if(grep_pid == 0) {

    /* pipe to stdin */
    setup_pipe(printenv_grep_pipe, grep_sort_pipe);

    /* Change process image */
    if(argc == 1) {
        /* If no arguments, send '.' to grep */
        execlp("grep", "grep", ".", NULL);
    } else {
        char arg1[] = "grep";
        argv[0] = arg1;
        execvp("grep", argv);
    }

    /* We only get here if something went wrong */
    exit(-1);
}
else if(grep_pid == -1) {
    exit(-1);
}

/* Close first pipe to let sort start */
close(printenv_grep_pipe[READ_SIDE]);
close(printenv_grep_pipe[WRITE_SIDE]);

/* Listen to grep, sort, and pipe output to pager */
sort_pid = fork();
if(sort_pid == 0) {

    /* pipe to stdin */
    setup_pipe(grep_sort_pipe, sort_pager_pipe);

    /* Change process image */
    execlp("sort", "sort", NULL);

    /* We only get here if something went wrong */
    exit(-1);
}
else if(sort_pid == -1) {
    exit(-1);
}

close(grep_sort_pipe[READ_SIDE]);
close(grep_sort_pipe[WRITE_SIDE]);

/* Listen to grep, sort, and pipe output to pager */
pager_pid = fork();
if(pager_pid == 0) {
```

```
        /* pipe to stdin */
        setup_pipe(sort_pager_pipe, NULL);

        /* Change process image */
        execlp(pager, pager, NULL);

        /* If 'less' fails, try 'more' */
        execlp("more", "more", NULL);

        /* We only get here if something went wrong */
        exit(-1);
    }
    else if(pager_pid == -1) {
        exit(-1);
    }

    close(sort_pager_pipe[READ_SIDE]);
    close(sort_pager_pipe[WRITE_SIDE]);

    /* Wait for all processes and examine the nature of their exit */
    checkpid("printenv", printenv_pid);

    /* Special case: grep returns with 1 if no lines matched */
    if(waitpid(grep_pid, &status, 0) == -1) {
        perror("waitpid");
        exit(-1);
    }
    if(!WIFEXITED(status)) {
        fputs("grep process did not exit normally", stderr);
        exit(-1);
    } else {
        switch(WEXITSTATUS(status)) {
            case 0:
                /* One or more lines was selected */
                break;
            case 1:
                /* No lines were selected */
                break;
            default:
                fputs("grep exited with error", stderr);
                exit(-1);
        }
    }

    checkpid("sort", sort_pid);
    checkpid("pager", pager_pid);

    return 0;
}

/* Does 'waitpid(pid, ...)' then checks exit status and quits if abnormal */
```

```
void checkpid(char *name, pid_t pid) {
    int status;
    if(waitpid(pid, &status, 0) == -1) {
        perror("waitpid");
        exit(-1);
    }
    if(!WIFEXITED(status)) {
        fputs(name, stderr);
        fputs(": process did not exit normally", stderr);
        exit(-1);
    } else {
        if(WEXITSTATUS(status)) {
            fputs(name, stderr);
            fputs(": did not exit with status 0", stderr);
            exit(-1);
        }
    }
}

/*
 * Redirects listen_pipe to stdin, send_pipe to stdout
 * If an argument set to null, that argument will be ignored
 */
void setup_pipe(int *listen_pipe, int *send_pipe) {
     /* listen_pipe to stdin */
     if(listen_pipe != NULL) {
         close(listen_pipe[WRITE_SIDE]);
         if(dup2(listen_pipe[READ_SIDE], STDIN_FILENO) == -1) {
             fputs("Pipe couldn't be redirected", stderr);
             exit(-1);
         }
         close(listen_pipe[READ_SIDE]);
     }

     /* send_pipe to stdout */
     if(send_pipe != NULL) {
         close(send_pipe[READ_SIDE]);
         if(dup2(send_pipe[WRITE_SIDE], STDOUT_FILENO) == -1) {
             fputs("Pipe couldn't be redirected", stderr);
             exit(-1);
         }
         close(send_pipe[WRITE_SIDE]);
     }
}

/* These are a collection of helper function to minishell. Explanations can be
 * read before each function definition */
#include "helpers.h"
#include <stdlib.h>
#include <string.h>

char *strtok(char*, const char*);
```

```
/* Takes the string the user inputted and creates an array of the tokens in it */
char **args_tokenized(char *input_string) {
    /* Assume no more than 32 space separated tokens was entered */
    char **args = malloc(sizeof(char**) * 32);
    char *token;
    int i;

    /* Ugly hack to be able to know which elements to free */
    for(i = 0; i < 32; i++) {
        args[i] = NULL;
    }

    /* Take one token at a time, allocate and add to 'args' */
    token = strtok(input_string, " \t");
    for(i = 0; token != NULL; i++) {
        args[i] = malloc(strlen(token) + 1);
        strcpy(args[i], token);
        token = strtok(NULL, " \t");
    }

    return args;
}

/* Frees all tokens in argument and finally the array itself */
void free_args(char **args) {
    int i;
    for(i = 0; args[i] != NULL; i++) {
       free(args[i]);
    }
    free(args);
}
```