Wydział Informatyki Politechniki Białostockiej Przedmiot: Systemy operacyjne	Data: 13.05.2024.
Ćwiczenie nr. 2 Temat: Projekt 1 - Prosta powłoka tekstowa (shell) Grupa PS 7 1. Michał Sienkiewicz 2. Jakub Wyszyński	Prowadzący: dr Marcin Koźniewski Ocena:

Treść

Prosta powłoka tekstowa (shell):

Powłoka pobiera ze standardowego wejścia pojedynczy wiersz. Następnie dokonuje prostej analizy wiersza dzieląc go na słowa separowane spacjami. Pierwsze słowo jest nazwa programu który należy uruchomić (wykorzystując zmienną PATH), a pozostałe sa argumentami. UWAGA: procesy uruchamiać należy poprzez wywołania systemowe z rodziny exec (execl, execlp, execvp itd.) - inne rozwiązania (np. przekierowanie odpowiedzialności do systemowej powłoki) nie będą uznawane. Shell uruchamia program i standardowo czeka na zakończenie jego pracy, chyba ze ostatnim słowem jest znak & co powoduje uruchomienie programu w tle, jak w normalnym shellu bash. Shell kończy prace gdy otrzyma znak końca pliku. Dzięki temu możliwe jest przygotowanie prostych skryptów, które można uruchamiać z wiersza poleceń bash-a, jeżeli pierwsza linia skryptu ma postać #!/home/student/moj_shell (gdzie po ! podaje się ścieżkę do programu shella). Shell powinien również implementować funkcję zmiany katalogu roboczego (w stylu cd) **(12p)**. Dodatkowe opcje to:

- możliwość przekierowania standardowego wyjścia polecenia przy pomocy » **(6p)**
- możliwość tworzenia potoków o dowolnej długości przy pomocy znaku | **(8p)**
- historia poleceń - shell przechowuje (w zewnętrznym pliku w katalogu domowym użytkownika - tak ze historia powinna ”przetrwać” zakończenie shella) dokładna treść 20 poleceń, a wysłanie sygnału SIGQUIT powoduje wyświetlenie historii na standardowym wyjściu. **(8p)**

Sposób uruchamiania

W celu uruchomienia naszej powłoki shell znajdując się w katalogu z projektem wykonujemy Makefile wpisując w terminalu polecenie „make”. Następnie po skompilowaniu naszego programu uruchamiamy go za pomocą polecenia ./main.

Testowanie projektu

W celu przetestowania powłoki shell:

- W celu wykonania danego programu w tle wpisujemy np. „sleep 5 &”, te polecenie domyślnie powoduje oczekiwanie przez X sekund jednakże wywołanie go w tle spowoduje brak oczekiwania.
- W celu sprawdzenia czy shell zakończy pracę po otrzymaniu końca pliku wystarczy w terminalu kliknąć „Ctrl+D”.
- W celu sprawdzenia możliwości uruchamiania prostych skryptów wystarczy w powłoce bash za pomocą polecenia „sh s.sh” odpalić skrypt załączony razem z programem. Skrypt ten zawiera na początku tzw. Shebang (#!) powodujący wywołanie go w wybranej powłoce shell. (Wymagana będzie zmiana ścieżki do shella).

- W celu sprawdzenia funkcji zmiany katalogu roboczego (w stylu cd) wystarczy wpisać polecenie np. „cd /ścieżka_do_katalogu”.
- W celu sprawdzenia przekierowania standardowego wyjścia polecenia przy pomocy >> należy wywołać jakieś polecenie np. ls -l, a następnie wpisać po nim >> nazwa_pliku.format (np. ls -l >> out.txt).
- W celu sprawdzenia możliwości tworzenia potoków o dowolnej długości przy pomocy znaku | można wykonać polecenie np. „ls -l | grep tekst” . Polecenie to wypisze tylko pliki z „tekst” w nazwie. Można również użyć większej ilości poleceń np. wc – „ls -l | grep main | wc”.
- W celu sprawdzenia historii poleceń należy dać shellowi sygnał SIGQUIT (Ctrl+\).

Zaimplementowane funkcjonalności

1. Historia Poleceń: Zapisywanie historii poleceń do pliku history.txt w katalogu domowym użytkownika.

```
void display_history() {
    char *home_directory = getenv("HOME");
    if (home_directory == NULL) {
        fprintf(stderr, "Error: HOME environment variable is not set.\n");
        return;
    }

    char history_file[MAX_COMMAND_LEN];
    snprintf(history_file, sizeof(history_file), "%s/history.txt",
home_directory);

    FILE *file = fopen(history_file, "r");
    if (file) {
        char line[MAX_COMMAND_LEN];
        while (fgets(line, sizeof(line), file)) {
            printf("%s", line);
        }
        fclose(file);
    } else {
        perror("Unable to open history file");
    }
}
void handle_sigquit(int sig) {
    printf("\n");
    display_history();
    printf("$ ");
    fflush(stdout);
    signal(SIGQUIT, handle_sigquit); // Ponowne ustawienie obsługi SIGQUIT
}
```

```
// W funkcji main:
signal(SIGQUIT, handle_sigquit);

char *home_directory = getenv("HOME");
        if (home_directory == NULL) {
            fprintf(stderr, "Error: HOME environment variable is not set.\n");
            continue;
        }

        char history_file[MAX_COMMAND_LEN];
        snprintf(history_file, sizeof(history_file), "%s/history.txt",
home_directory);

        FILE *history_file_ptr = fopen(history_file, "a");
        if (history_file_ptr) {
            fprintf(history_file_ptr, "%s", command_line);
            fclose(history_file_ptr);
        } else {
            perror("Unable to open history file");
        }
```

2. Obsługa Potoków: Możliwość wykonywania potokowych poleceń, np. ls, | grep. (|)

```
const char pipe_delim[] = "|";

// Fragmenty kodu…

int arg_count = 0;
token = strtok(command_line, pipe_delim);
while (token != NULL) {
       args[arg_count++] = token;
       token = strtok(NULL, pipe_delim);
}
args[arg_count] = NULL;


// Fragmenty kodu…

int prev_read_end = STDIN_FILENO;
        for (int i = 0; i < arg_count; ++i) {
            int pipefd[2];
            if (i < arg_count - 1) {
                pipe(pipefd);
            }

            char *cmd_args[MAX_ARGS];
```

```
            char *cmd_token = strtok(args[i], delim);
            int cmd_arg_count = 0;
            while (cmd_token != NULL) {
                cmd_args[cmd_arg_count++] = cmd_token;
                cmd_token = strtok(NULL, delim);
            }
            cmd_args[cmd_arg_count] = NULL;

            pid_t pid = fork();
            if (pid == 0) {
                dup2(prev_read_end, STDIN_FILENO);
                if (i < arg_count - 1) {
                    dup2(pipefd[1], STDOUT_FILENO);
                    close(pipefd[0]);
                } else {
                    if (output_redirected) {
                        int output_fd = open(output_file, O_WRONLY | O_CREAT |
O_APPEND, 0644);
                        if (output_fd == -1) {
                            perror("open");
                            exit(EXIT_FAILURE);
                        }
                        dup2(output_fd, STDOUT_FILENO);
                        close(output_fd);
                    }
                }
                execvp(cmd_args[0], cmd_args);
                perror("execvp");
                exit(EXIT_FAILURE);
            } else if (pid < 0) {
                perror("fork");
                exit(EXIT_FAILURE);
            }

            if (i < arg_count - 1) {
                close(pipefd[1]);
                prev_read_end = pipefd[0];
            }
        }
```

3. Przekierowanie Wyjścia: Obsługa przekierowania wyjścia do pliku (>>).

```
const char output_redirect_delim[] = ">>";
int output_redirected = 0;
char *output_file = NULL;

char *output_redirect = strstr(command_line, output_redirect_delim);
        if (output_redirect != NULL) {
```

```
            *output_redirect = '\0';
            output_redirect += strlen(output_redirect_delim);
            output_redirected = 1;
            output_file = strtok(output_redirect, delim);
        }
```

4. Zmiana Katalogu: Obsługa polecenia cd do zmiany bieżącego katalogu.

```
if (strncmp(command_line, "cd", 2) == 0) {
            char *new_dir = strtok(command_line + 2, delim);
            if (new_dir == NULL) {
                fprintf(stderr, "cd: missing argument\n");
            } else {
                if (chdir(new_dir) != 0) {
                    perror("cd");
                }
            }
            continue;
        }
```

5. Shell kończy prace gdy otrzyma znak końca pliku.

```
if (fgets(command_line, sizeof(command_line), stdin) == NULL) {
            if (feof(stdin)) {
                break;
            } else {
                perror("fgets");
                exit(EXIT_FAILURE);
            }
        }
```

6. Uruchamianie skryptów przy zastosowaniu shebang (#!).

```
void execute_script(char *script_path) {
    char *interpreter = "./main";
    char *script_args[] = {interpreter, script_path, NULL};
    execvp(script_args[0], script_args);
    perror("execvp");
    exit(EXIT_FAILURE);
}


// W funkcji main:
if (argc > 1){
    execute_script(argv[1]);
```

```
    return 0;
}
```

7. Dzielenie wiersza na słowa tj. nazwa programu oraz argumenty.

```
int arg_count = 0;
token = strtok(command_line, pipe_delim);
while (token != NULL) {
       args[arg_count++] = token;
       token = strtok(NULL, pipe_delim);
}
args[arg_count] = NULL;
```

8. Procesy uruchamiane przez wywołania systemowe z rodziny exec.

```
// Wykonanie komendy
                execvp(cmd_args[0], cmd_args);
```

9. Ostatni znak w poleceniu & powoduje wykonanie procesu w tle bez czekania na jego zakończenie.

```
char *rn_background = strstr(command_line, background_delim);
        if (rn_background != NULL) {
            *rn_background = '\0';
            rn_background += strlen(background_delim);
            run_in_background = 1;
            printf("Running command in background.\n");
        }

// Fragmenty kodu…

// Fragment do czekania na zakończenie w przypadku braku znaku &
int status;
      if (!run_in_background) {
      waitpid(pid, &status, 0);
      }
```

Opis funkcji display_history()

Funkcja display_history() służy do odczytu i wyświetlania historii poleceń z pliku tekstowego history.txt. Jest wywoływana po naciśnięciu kombinacji klawiszy „Ctrl + \” powodującej wygenerowanie sygnału SIGQUIT. Funkcja nie zwraca wartości.

- Pobiera ścieżkę do katalogu domowego użytkownika za pomocą funkcji getenv("HOME") i przypisuje ją do wskaźnika home_directory.
- Sprawdza, czy zmienna home_directory została prawidłowo pobrana. Jeśli nie, wypisuje komunikat błędu na standardowe wyjście błędów (stderr) i kończy działanie funkcji.

- Tworzy ścieżkę do pliku historii poprzez połączenie home_directory i nazwy pliku history.txt za pomocą funkcji snprintf().
- Otwiera plik historii do odczytu za pomocą funkcji fopen().
- Jeśli plik został otwarty poprawnie, czyta kolejne linie z pliku za pomocą funkcji fgets() i wyświetla je na standardowym wyjściu (stdout) za pomocą funkcji printf().
- Zamyka plik po zakończeniu operacji czytania.
- Jeśli plik nie został otwarty poprawnie, funkcja wypisuje komunikat błędu związany z problemem otwarcia pliku za pomocą funkcji perror().