Temat projektu: Prosty interpreter poleceń

Przygotowali: Jakub Bartnicki, Łukasz Filimoniuk, Krzysztof Horodeński	Prowadzący : dr Inż. Wojciech Kwedlo
Temat projektu: Prosty interpreter poleceń	Grupa pierwsza Informatyka i ekonometria

Treść zadania:

[10p] Zaprojektuj i zaimplementuj prosty interpreter poleceń. Interpreter pobiera ze standardowego wejścia pojedynczy wiersz. Następnie dokonuje prostej analizy wiersza dzieląc go na słowa separowane spacjami. Pierwsze słowo jest nazwą programu, który należy uruchomić (wykorzystując zmienną PATH), a pozostałe są argumentami. Shell uruchamia program i standardowo czeka na zakończenie jego pracy, chyba że ostatnim słowem jest znak & co powoduje uruchomienie programu w tle, jak w normalnym shellu bash. Shell kończy pracę 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 scieżkę do programu shella). Dodatkowe opcje to:

a) **[6p]** możliwość przekierowania standardowego wyjścia polecenia przy pomocy >>
b) **[9p]** możliwość tworzenia potoków o dowolnej długości przy pomocy znaku |
c) **[9p]** historia poleceń - shell przechowuje (w zewnętrznym pliku w katalogu domowym użytkownika - tak że historia powinna "przetrwać" zakończenie shella) dokładną treść 20 poleceń, a wysłanie sygnału SIGINT powoduje wyświetlenie historii na standardowym wyjściu

main.c

```
#include
<stdio.h>
            #include <stdlib.h>
            #include <string.h>
            #include <unistd.h>
            #include <errno.h>
            #include <stdbool.h>
            #include <signal.h>
            #include "src/history.h"
            #define OK       0
            #define NO_INPUT 1
            #define TOO_LONG 2
            #define INPUT_MAX_LENGTH 200
            #define RUN_PATH "./"
            // obsługa sygnału - wypisuje historię
```

```
void handleSigint(int sig) {
    showHistory();
}
// funkcja wypisująca znak '>' oraz wczytująca linijkę tekstu z terminala,
wartość zwracana do tablicy znaków text
static int getLine (char sign, char *text, size_t textSize) {
    char character;
    if (sign != '\0') putchar(sign);
    if (strlen(fgets(text, textSize + 1, stdin)) <= 1) return NO_INPUT;
    if (text[strlen(text) - 1] != '\n')
        while (((getchar()) != '\n') && (character != EOF)) return TOO_LONG;

    text[strlen(text) - 1] = '\0';
    return OK;
}
// funkcja usuwająca spacje na początku tablicy znaków text podanej jako
argument
char* removeSpaces(char* text) {
    char* string = (char*) malloc(INPUT_MAX_LENGTH * sizeof(char));
    int i, j = 0;
    strcpy(string, text);
    for(i=0; string[i]==' ' || string[i]=='\t'; i++);

       for(j=0; string[i]; i++)
       {
              string[j++] = string[i];
       }
       string[j]='\0';
       for(i=0; string[i] != '\0'; i++)
       {
              if(string[i] != ' ' && string[i] != '\t')
                     j = i;
       }
       string[j+1] = '\0';
    return string;
}
// funkcja sprawdzająca liczbę powtórzeń znaku '|' w podanej tablicy znaków i
zwracająca tę liczbę
int checkSignAmount(char* readed)
{
       int counter;
       int signAmount = 0;
       for (counter = 0; readed[counter] != '\0'; counter++) {
           if (readed[counter] == '|') signAmount++;
    }
       return signAmount;
```

```
}
// funkcja sprawdzająca czy podana tablica znaków jest literą, zwraca
True/False
bool isLetter(char character) {
    return ((character >= 'a' && character <= 'z') || (character >= 'A' &&
character <= 'Z'));
}
// funkcja sprawdzająca czy podane argumenty są identyczne, zwraca True/False
bool checkInput(char* option, char* input) {
    return !(strcmp(option, input));
}
// funkcja tworząca i zwracająca gotowe polecenie do wykonania na podstawie
podanej tablicy znaków
char* createCommand(char* readedCommand) {
    char* command = (char*) malloc(INPUT_MAX_LENGTH * sizeof(char));
    strcpy(command, RUN_PATH);
    strcat(command, removeSpaces(readedCommand));
    return command;
}
// funkcja uruchamiająca podaną w argumentach komendę
void runCommand(char* command) {
    if (system(createCommand(command))) perror("Command not found");
}
// funkcja zamykająca interpreter
void exitProgram() {
    puts("exitting...");
    exit(0);
}
// funkcja oddzielająca polecenia oddzielone znakiem '|', wywołaniem funkcji
runCommand uruchamia te polecenia oddzielnie
void connectStreams(char* readed) {
    int partcount = checkSignAmount(readed) + 1;
    char *foundPipe, *newReaded;
    int i, j;
    newReaded = (char*) malloc(INPUT_MAX_LENGTH * sizeof(char));
    strcpy(newReaded, readed);
    char** commandArray = (char**) malloc(INPUT_MAX_LENGTH * sizeof(char));
    for (i = 0; i < partcount; i++)
        commandArray[i] = (char*) malloc(INPUT_MAX_LENGTH * sizeof(char));
    for(i = 0; (foundPipe = strsep(&newReaded, "|")) != NULL; i++)
        strcpy(commandArray[i], foundPipe);

    for(i = 0; i < partcount; i++) {
        char* command, *exitCommand;
        command = createCommand(commandArray[i]);
        exitCommand = commandArray[i];
```

```
            strtok(exitCommand, " ");
            if (checkInput(removeSpaces(exitCommand), "exit")) exitProgram();
            runCommand(command);
            free(exitCommand);
            free(command);
        }
        free(foundPipe);
        free(newReaded);
        free(commandArray);
}
// funkcja, która ma za zadanie sprawdzać czy podana funkcja występuje w liście dostępnych poleceń
void chooseOption(char* option, char* readed) {
    if (checkInput(option, "exit")) exitProgram();
    else if (checkInput(option, "sum")) runCommand(readed);
    else if (checkInput(option, "counter")) runCommand(readed);
    else if (checkInput(option, "help")) runCommand(readed);
    else if (checkInput(option, "rectangle")) runCommand(readed);
    else if (checkInput(option, "remainder")) runCommand(readed);
    else if (checkInput(option, "echo")) runCommand(readed);
    else printf("%s: command not found\n", readed);
}
// funkcja, w której wczytana zostaje komenda poprzez getLine(), i wydawane są polecenia uruchamiania poleceń i dodawania ich do historii poleceń
void readCommand(ListElement_type **headOfHistoryList, char* option, char* readed) {
    readed = (char*) malloc(INPUT_MAX_LENGTH * sizeof(char));
    int inputStatus = getLine('>', readed, INPUT_MAX_LENGTH * sizeof(char));
    if (inputStatus == NO_INPUT) return;
    if (inputStatus == TOO_LONG) {
        perror("Input is too long.");
        return;
    }
    option = (char*) malloc(INPUT_MAX_LENGTH * sizeof(char));
    strcpy(option, readed);
    if (!checkInput(option, "history") && !checkInput(option, "help") && !checkInput(option, "exit"))
        strtok(option, " ");
    strcpy(readed, removeSpaces(readed));
    if (checkSignAmount(readed) > 0) connectStreams(readed);
    else chooseOption(option, readed);
    addHistoryElement(&(*headOfHistoryList), readed);
    free(option);
    free(readed);
}
//funkcja główna programu, służy głównie do deklaracji najważniejszych
```

```
zmiennych i uruchamianiu pętli z poleceniem readCommand
int main(int argc, const char* argv[]) {
    ListElement_type** headOfHistoryList = (ListElement_type**)
malloc(sizeof(ListElement_type*));
    char* option;
    char* readed;
    if (argc > 1) perror("Unnecessary arguments have been ignored");
    signal(SIGINT, handleSigint);
    while (1) {
        readCommand(headOfHistoryList, option, readed);
    }
    free(headOfHistoryList);
    return 0;
}
```

history.c

```
#include
<stdlib.h>
    #include <stdio.h>
    #include <string.h>
    #include <unistd.h>
    #include "history.h"
    #define INPUT_MAX_LENGTH 200
    #define PATH "/home/history.txt"
    // funkcja wyświetlająca historię na standardowym wyjściu
    void showHistory() {
        FILE *historyFile = fopen(PATH, "r");
        if (historyFile == NULL) {
            perror("Unable to open file");
            return;
        }
        int character;
        while( (character = fgetc(historyFile)) != EOF && character != '\0')
           {
                  putchar(character);
           }
        putchar('\n');
        fclose(historyFile);
    }
    // funkcja zwracająca liczbę elementów podanej w argumentach listy
    jednokierunkowej
    int getHistorySize(ListElement_type *headOfHistoryList)
    {
```

```
    int counter = 0;
    if (headOfHistoryList == NULL) return counter;
    else {
        ListElement_type *current = headOfHistoryList;
            do {
            counter++;
            current = current->next;
            } while (current != NULL);
    }
    return counter;
}
// funkcja wpisująca do pliku z historią aktualną historię, jednocześnie
usuwa starą
void updateHistory(ListElement_type *headOfHistoryList) {
    FILE *historyFile = fopen(PATH, "w");
    ListElement_type *current;
    if (headOfHistoryList != NULL)
         current = headOfHistoryList;
    else return;

    int i = 1;
    while (current->next != NULL) {
        fprintf(historyFile, "%d. %s\n", i, current->data);
        current = current->next;
        i++;
    }

    fclose(historyFile);
}
// funkcja dodająca element do listy podanej jednokierunkowej
void addHistoryElement(ListElement_type **headOfHistoryList, char* text)
{
       if((*headOfHistoryList) == NULL) {
              (*headOfHistoryList) = (ListElement_type *)
malloc(sizeof(ListElement_type));
        (*headOfHistoryList)->data = (char*) malloc(sizeof(char) *
INPUT_MAX_LENGTH);
              strcpy((*headOfHistoryList)->data, text);
       (*headOfHistoryList)->next = NULL;
       } else {
        if (getHistorySize((*headOfHistoryList)) >= 21) {
            ListElement_type * newListHead = NULL;

            newListHead = (*headOfHistoryList)->next;
            (*headOfHistoryList) = newListHead;
        }
```

```
                    ListElement_type *current = (*headOfHistoryList);

                while (current->next != NULL) {
                    current = current->next;
                }
                current->next = (ListElement_type *
)malloc(sizeof(ListElement_type));
        current->next->data = (char*) malloc(sizeof(char) *
INPUT_MAX_LENGTH);
                strcpy(current->next->data, text);
                current->next->next = NULL;
            }
        updateHistory(*headOfHistoryList);
}
```

history.h

```
#ifndef
HISTORY_H_
            #define HISTORY_H_
            typedef struct ListElement {
                char* data;
                struct ListElement * next;
            } ListElement_type;
            void showHistory();
            void addHistoryElement(ListElement_type **headOfHistoryList, char* text);
            #endif
```

sum.c

```
#include
<stdlib.h>
            #include <stdio.h>
            #define typename(x) _Generic((x), int: "int")
            // Polecenie wypisujące sumę podanych argumentów (argv[>1])
            int main(int argc, const char* argv[]) {

                int i, sum = 0;
                char* type;
                if (argc == 1) {
                    perror("Please provide the command line arguments");
                }
                for (i = 1; i < argc; i++) {
```

```
        typeof(argv[i]) type = argv[i];

        sum += atoi(argv[i]);
    }
    printf("%d\n", sum);

    return 0;
}
```

help.c

```
#include
<stdlib.h>
#include <stdio.h>
// Polecenie wypisujące pomoc z informacją o dostępnych poleceniach
int main(int argc, const char* argv[]) {
    puts("  help - show this help list\n"                              \
         "  sum x [y z ...] - sum 2 (or more) numbers\n"               \
         "  counter x - countdown from x to 0\n"                       \
         "  remainder x y - show result of x % y \n"                   \
         "  rectangle x y - print rectangle (x - high, y - width)\n"   \
         "  echo sth - print text sth\n"                               \
         "  ctrl+C - show commands history\n"                          \
         "  exit - exit the interpreter"                               \
);

    return 0;
}
```

echo.c

```
#include
<stdlib.h>
#include <stdio.h>
#include <string.h>
// Polecenie wypisujące na standardowym wyjściu podane argumenty (argv[>0])
int main(int argc, const char* argv[]) {
    int i;
    if (argc == 1) perror("Add command line arguments");
    for (i = 1; i < argc; i++) printf("%s ", argv[i]);

    putchar('\n');
```

```
    return 0;
}
```

remainder.c

```
#include
<stdlib.h>
#include <stdio.h>
#include <string.h>
// Polecenie wypisujące resztę z dzielenia dwóch liczb (argv[1] % argv[2])
int main(int argc, const char* argv[]) {
    int a = atoi(argv[1]);
    int b = atoi(argv[2]);
    if (argc != 3) {
        perror("Please add 2 command line arguments");
        return 0;
    }

       printf("%d\n", a % b);

    return 0;
}
```

rectangle.c

```
#include
<stdlib.h>
#include <stdio.h>
#include <string.h>
// Polecenie "rysujące" prostokąt o podanych wymiarach (wysokość: argv[1],
szerokość: argv[2])
int main(int argc, const char* argv[]) {
    if (argc != 3) {
       perror("Please add 2 command line arguments");
       return 0;
       }
       int i, j;
       int high = atoi(argv[1]);
       int width = atoi(argv[2]);
       for (i = 0; i < high; i++) {
              for (j = 0; j < width; j++) printf("*");
```

```
                printf("\n");
            }
        return 0;
    }
```

counter.c

```
#include
<stdlib.h>
        #include <stdio.h>
        #include <string.h>
        #include <unistd.h>
        // Polecenie odliczające od podanego argumentu (argv[1]) do 0
        int main(int argc, const char* argv[]) {
            if ((argc > 3 && strcmp(argv[argc - 1], "&")) || (argc == 1 || argc > 2))
        {
                perror("Please provide the command line 1 argument\n");
                return 0;
            }
            int number = atoi(argv[1]);
            while (number > 0) {
                printf("%d\n", number);
                sleep(1);
                number--;
            }
            return 0;
        }
```

1.Algorytm wczytania komend ze standardowego wejścia, przerobienia ich tak, aby miały odpowiednią formę oraz otworzenia odpowiednich programów z ich użyciem.

a) W funkcji main() zostaje uruchomiona funkcja readCommand(), pobiera komendę poprzez wywołanie funkcji getLine(), w tej funkcji sprawdzamy czy występują potoki, jeśli tak wywołana jest funkcja connectStreams()
b) Funkcja connectStreams() rozdziela potok na określoną liczbę komend i wywołuje funkcję runCommand()/exitProgram() dla każdej z nich
c) Dla pojedynczego polecenia następuje wczytanie funkcji chooseOption(), która sprawdza rodzaj komendy
d) Funkcja chooseOption() wywołuje funkcję runCommand(), która wczytuje program lub exitProgram()

2.Algorytm zapisu do historii

a) W funkcji readCommand() po każdym wczytaniu komendy jest wywoływana funkcja addHistoryElement()
b) Funkcja addHistoryElement() dodaje do listy nowy element oraz pilnuje, aby na liście nie było ponad 20 komend przez usuwając najstarsze
c) W funkcji addHistoryElement() następuje wywołanie funkcji updateHistory(), która zapisuje aktualna historię do pliku history.txt

Opis funkcji:

1) readCommand() – To funkcja odpowiedzialna wywołanie komendy getLine() pobierającej tekst oraz odczytanie tej komendy, dla komendy z potokiem wywołuje funkcję connectStreams(), a dla normalnej komendy funkcję chooseOptions()
2) chooseOption() – To funkcja odpowiedzialna za sprawdzenie pierwszego słowa w komendzie i wywołanie odpowiedniej funkcji w zależności od tego słowa
3) addHistoryElement() – To funkcja odpowiedzialna za dodawanie poleceń do listy przechowującej historię oraz pilnuje, aby na liście nie było więcej niż 20 komend przez usuwając najstarsze
4) updateHistory() – To funkcja odpowiedzialna za zapisanie aktualnej historii do pliku historia.txt
5) showHistory() – To funkcja odpowiedzialna za wczytywanie historii z pliku historia.txt oraz wypisanie jej
6) getHistorySize() – To funkcja zwracająca liczbę elementów listy na której przechowywana jest historia
7) connectStreams() – To funkcja odpowiedzialna za rozdzielenie potoku na odpowiednią ilość komend
8) exitProgram() – To funkcja odpowiedzialna za wypisanie napisu exitting… w terminalu oraz wyjście z powłoki
9) runCommand() – To funkcja odpowiedzialna za uruchomienie funkcji system
10) createCommand() – To funkcja zwracająca komendę z dodanym ”./ ” na jej początku
11) checkInput() – To funkcja porównująca dwa stringi i zwracająca prawdę lub fałsz w zależności od wyników porównania
12) isLetter() – To funkcja sprawdzająca czy znak jest literą i zwracająca prawdę lub fałsz w zależności od wyników porównania
13) checkSignAmount() – To funkcja zwracająca ilość wystąpień znaku „|”
14) removeSpaces() – To funkcja zwracająca tekst z usuniętymi spacjami występującymi na początku argumentu
15) getLine() – To funkcja odpowiedzialna za pobranie i zwrócenie komendy oraz wypisanie w terminalu znaku „>”

Opis programów:

1) sum.o – Program drukujący w terminalu sumę podanych argumentów
2) help.o – Program wyświetlający informacje o dostępnych komendach w shellu
3) echo.o – Program drukujący swoje argumenty w terminalu
4) remainder.o – Program zwracający resztę z dzielenia pierwszego argumentu przez drugi
5) rectangle.o – Program rysujący prostokąt o wysokości i szerokości pobranej argumentów funkcji
6) counter.o – Program wypisujący liczby od podanej w argumencie do zera