Sprawozdanie końcowe UXP

Skład zespołu

Wiktor Michalski
Robert Ostoja-Lniski
Wojciech Sitek
Maciej Szulik

Treść zadania

Treść: Napisać prosty interakcyjny interpreter poleceń (shell) posiadający minimum następujące funkcje:

1) możliwość uruchamiania programów w postaci potoku arbitralnej długości “proga | progb | progc”
2) możliwość przekierowania we/wy do plików “>”, “<”
3) obsługa kodu powrotu jako zmiennej shella $?
4) podstawowe operacje wbudowane: cd, pwd, echo, obsługa separatora ; (średnik)
5) obsługa cytowania ‘...’ (pojedyncze zwykłe cudzysłowy)
6) obsługa zmiennych shella i środowiskowych (np. PATH; HOME; itp.); poprzez konstrukcję “=” oraz “export”
7) pozostałych funkcji tworzących shell-a (np. konstrukcje warunkowe i sterujące, rozwijanie wzorców plików, …) nie trzeba realizować, chyba że wynikają z zadanego wariantu (niżej)
8) w przypadku wątpliwości co do zachowania i składni należy bazować na shell “sh/bash”

Komunikację między procesami w potoku zrealizować jako:

- W12 - potoki nazwane.

Dodatkowe wymaganie:

- W22 - możliwość uruchamiania potoków w tle i przełączenia pierwszy / drugi plan.

Interpretacja treści zadania

Zadanie polega na zaprojektowaniu oraz zaimplementowaniu programu interaktywnego realizującego podstawowe funkcje shella. Program ten będzie umożliwiał wpisywanie przez użytkownika poleceń takich jak cd, pwd, echo, a także będzie obsługiwał odpowiedni zbiór operatorów (“`|`”, “>”, “<”, `$?`, =, ‘ ‘). Ponadto na dodatkowe cechy tego programu składają się: zaimplementowana realizacja potoków nazwanych, a także przełączanie procesów między pierwszym i drugim planem (z wykorzystaniem poleceń `fg`, `bg` i operatora `&` do uruchomienia procesu w tle).

Składnia poleceń pokrywa się ze składnią basha. Aby polecenie mogło być interpretowane należy je zakończyć klawiszem “enter”, lub znakiem “ ; “. Jeśli polecenie będzie posiadało błędy składniowe, program wyświetli użytkownikowi odpowiedni komunikat.

Opis funkcjonalny oraz api użytkownika

Użytkownikowi po uruchomieniu programu pokazuje się krótka informacja na temat projektu (nazwa projektu, przedmiotu, semestr studiów oraz imiona i nazwiska autorów projektu). Poniżej znajduje się wiersz określający nazwę użytkownika korzystającego z programu, symbol ‘@’ oraz aktualny katalog, w którym się znajduje. Całość jest zakończona znakiem zachęty oznaczającym, że program oczekuje podania polecenia.

Użytkownik może korzystać zarówno z poleceń zewnętrznych i wewnętrznych.

Polecenia wewnętrzne obsługiwane przez program to “cd”, “export” oraz “exit”

1. Polecenie “cd”:
    a. Polecenie cd może być podane bez żadnego dodatkowego argumentu, wtedy przenosi użytkownika do katalogu domowego.
    b. Polecenie z innym argumentem przenosi do podanej (względnej lub bezwzględnej) lokalizacji, o ile określa ona poprawną ścieżkę
2. Polecenie “export”
    a. Polecenie export może przyjmować jako parametr zmienną, która była wcześniej zapisana w lokalnej hashmapie, co ustawi ją globalnie w środowisku.
    b. Polecenie export może przyjmować jako parametr przypisanie postaci `klucz=wartosc` - wtedy nie ma potrzeby wcześniejszego zapisania jej lokalnie.
    c. Użycie polecenia export bez żadnego parametru wypisze listę zmiennych środowiskowych.
3. Polecenie “exit”
    a. Umożliwia opuszczenie programu

Polecenia zewnętrzne obsługiwane przez program są wywoływane poprzez funkcję systemową execvp, dzięki czemu użytkownik może korzystać z programów do których ścieżki są określone w zmiennej PATH.

1. Korzystanie z operatora | umożliwia tworzenie potoków arbitralnej długości.
    a. Wpisanie operatora | bez argumentów powoduje wypisanie informacji, że nie można zinterpretować tego polecenia.
    b. Użytkownik może tworzyć potoki praktycznie nieograniczonej długości (ograniczeniem jest maksymalna długość polecenia)
2. Użytkownik może korzystać z operatów > oraz <
    a. Jest możliwe korzystanie z operatora < i > w jednym poleceniu
3. Operacja cytowania ‘ ‘ jest obsługiwana. Wszystko co znajduje się w symbolach ‘ ‘ jest interpretowane jako znak tekstowy. Program uwzględnia konkatenację wyrażeń

otoczonych przez symbole ‘ ‘ i wyrażeń nie posiadających cudzysłowów np. echo ‘abc’efg wyświetli abcefg.

4. Użytkownik może zakończyć polecenie poprzez podanie operatora ; lub wpisując klawisz enter. W przeciwnych przypadkach dane podane przez użytkownika będą traktowane jako kolejne argumenty danego polecenia. np cat Makefile echo abd będzie interpretowany jako wywołanie programu cat z argumentem Makefile, następnie echo i na koniec abd.

Przenoszenie procesów w tło i przywracanie z tła (funkcja niezaimplementowana):

1. Użytkownik ma do dyspozycji 5 mechanizmów związanych z obsługą tła:
    a. operator & umieszczany na końcu grupy komend (job) służący do uruchomienia w tle poprzez stworzenie grupy procesów
    b. polecenie jobs wyświetlające listę zadań (jobs) wykonywanych w tle
    c. polecenie fg <id> przywracające zadanie o id na pierwszy plan (sterowanie terminalem), lub samo fg, by przywrócić ostatnie z zadań
    d. polecenie bg <id> wznawiające zatrzymane zadanie o id w tle, lub samo bg, by przywrócić ostatnie z zadań
    e. CTRL-Z by zatrzymać aktywnie wykonywane zadanie działające na pierwszym planie

Korzystanie z potoków nazwanych

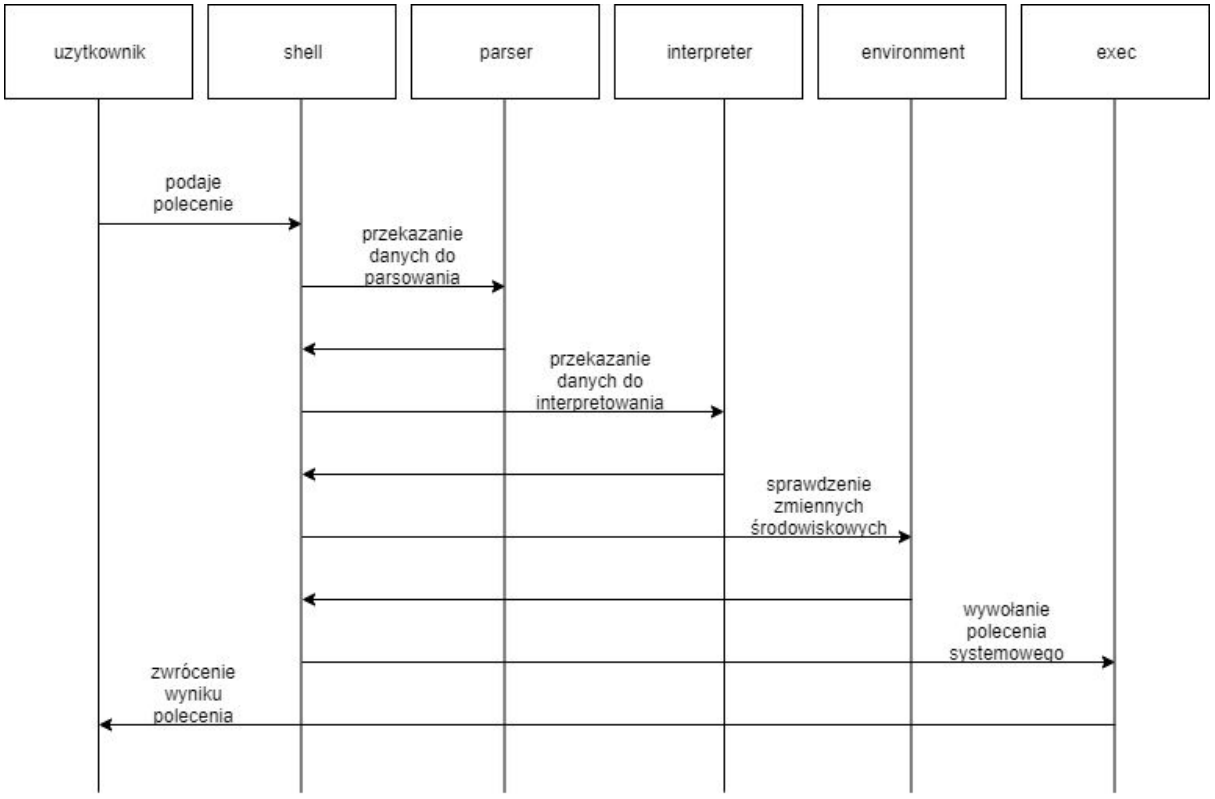
Z perspektywy użytkownika nie jest widoczne korzystanie z potoków nazwanych. Korzystając z symbolu | dostanie utworzony odpowiedni plik specjalny przy pomocy polecenia mkfifo, a procesy będą w stanie się komunikować ze sobą przy użyciu deskryptorów do zapisu i odczytu.

Ponadto, ponieważ obsługiwane są polecenia zewnętrzne użytkownik ma dostęp do polecenia mkfifo (o ile istnieje /usr/bin/mkfifo) dzięki czemu może jawnie korzystać z potoków nazwanych wykorzystując do tego zaimplementowane w programie operatory > oraz <.

Dodatkowe funkcje programu:
Program uwzględnia przypadki szczególne podczas wpisywania poleceń.

1. Użytkownik podając otwierający cudzysłów, którego nie zamknie przed wciśnięciem klawisza enter, nie wprowadza polecenia do programu. Widoczny jest symbol “>”, w wierszu poleceń dopóki nie zostanie podany zamykający cudzysłów.
2. Obsługiwany jest znak \ na końcu linii. Dzięki niemu użytkownik może wprowadzać kolejne symbole w następnej linii, które zostaną dołączone do tych w poprzedniej z pominięciem znaku “\”.

Rysunek poniżej przedstawia komunikację między modułami.



Podział programu na moduły

1. Main.cpp
    a. W klasie Main.cpp inicjalizowany jest program i uruchamiany
2. Shell.cpp
    a. Jest to klasa umożliwiająca odbieranie poleceń od użytkownika i przekazywanie ich od parsera.
    b. Umożliwia blokowanie sygnałów dla programu.
    c. W niej zaimplementowanie są metody odpowiedzialne za wyświetlanie użytkownikowi znaku zachęty.
    d. Rozpatruje przypadki szczególne (np wpisanie pustego polecenia, podanie cudzysłowu bez zamknięcia itp) dzięki czemu do parsera przekazywane są wyrażenia możliwe do przetwarzania.
    e. Po odebraniu pogrupowanych logicznie danych przekazuje je do klasy odpowiedzialnej za wykonywanie programów.
3. Parser.cpp
    a. Jest to klasa zajmująca się podziałem wejścia od użytkownika na prace (jobs), a następnie dla każdej pracy na tokeny.
    b. Parser przyjmuje input w postaci stringa i dzieli go na wektor stringów jobs przy użyciu separatora “;” (ignoruje go wewnątrz cudzysłowów).
    c. Parser dzieli każdy wyekstraktowany job na tokeny. Odbywa się to przy użyciu separatora spacji (za wyjątkiem spacji wewnątrz cudzysłowów). Operatory shella również stają się tokenami (|, &, <, >).
4. Interpreter.cpp
    a. Jest to klasa, która zajmuje się interpretacją przesłanych tokenów w obrębie jednego stringa reprezentującego joba. Wynikiem jej działania jest struktura Job.
    b. Interpreter dzieli zbiór tokenów na komendy (separatorem dla komend jest operator |).
    c. Interpreter dla każdej komendy sprawdza czy chcemy przekazać standardowe wejście i wyjście, zapamiętuje odpowiednio nazwy plików.
    d. Interpreter sprawdza czy ostatnim tokenem jest &, co oznacza chęć uruchomienia joba w tle.
    e. Interpreter zwraca przygotowaną do wykonania strukturę Job.
5. Exec.cpp

    a. Klasa Exec zajmuje się obsługą wykonania danego joba.
    b. Klasa wykonuje kolejne komendy zawarte w obrębie danego joba.
    c. Exec rozpoznaje czy dana komenda jest wbudowana i jeśli tak wykonuje odpowiednią funkcję.
    d. Jeśli job zawiera więcej niż jedną komendę, oznacza to, że należy zbudować potok. Klasa tworzy odpowiednią liczbę potoków nazwanych.
    e. Klasa zleca wykonanie komend, w zależności od wejścia użytkownika ustawia przekierowania do plików lub do potoków.
    f. Po wykonaniu wszystkich komend klasa usuwa potoki nazwane.
6. Environment.cpp
    a. Klasa Environment robi operacje na kolejnych tokenach z Parsera oraz przekazuje je do Interpretera.
    b. Klasa zajmuje się zarządzaniem zmiennymi środowiskowymi: wczytywanie, zapisywanie lokalnie w hashmapie, funkcja export.
    c. Zapisywanie zmiennych funkcją export odbywa się w całości w klasie Environment - tokeny nie są wtedy przekazywane dalej.
    d. Jeśli funkcja export nie otrzymała dodatkowych parametrów (czyli użytkownik chce wypisać zmienne środowiskowe), to obsługa polecenia jest przekazana dalej.

Zarys koncepcji implementacji

Język programowania:

1. Do implementacji programu będą wykorzystywane języki C i C++.
    a. Użyta zostanie biblioteka C POSIX library (m.in z plikami nagłówkowymi unistd.h, pthread.h, semaphore.h, fcntl.h). Dzięki niej będzie możliwy wybór funkcji realizujących dane polecenie.
    b. Język C++ posłuży do stworzenia klas i obiektów w celu łatwiejszej implementacji danych modułów i komunikacji między nimi, zostaną wykorzystane najnowsze standardy (C++11, C++14, C++17).

Dodatkowe biblioteki:

1. Biblioteka Boost.Process ułatwi tworzenie procesów i komunikację między nimi. Będzie szczególnie przydatna do realizacji wymagania W22 - uruchamiania potoków programów w tle i przełączanie pierwszy / drugi plan.
2. Biblioteka Boost.Filesystem będzie wspomagać sprawdzanie poprawności danych wprowadzonych przez użytkownika. Posiada funkcje sprawdzające ścieżki do plików i do katalogów.

Notacja BNF

```
Opis operatorów specjalnych:
<PIPE_OPERATOR> ::= “|”
<OUT_OPERATOR> ::= “>”
<IN_OPERATOR> ::= “<”
<BG_OPERATOR> ::= “&”
<SMC_OPERATOR> ::= “;”

Opis symboli podstawowych:
<LETTER> ::=
a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|
A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

<DIGIT> ::= 0|1|2|3|4|5|6|7|8|9

<SYMBOL> ::= “.” | “_” | “/” | “-”

Opcja złożonych wyrażeń:
<NUMBER> ::= <DIGIT>
           | <NUMBER> <DIGIT>

<ANY_CHAR> = <SYMBOL> | <DIGIT> | <LETTER>
<QUOTE_UNDONE> ::= <ANY_CHAR> | <QUOTE_UNDONE> <ANY_CHAR>
<QUOTE> ::= “‘“ <QUOTE_UNDONE> “‘“

<WORD> ::= <LETTER>
         | <WORD> <LETTER>
         | <WORD> <SYMBOL>


<WORDS> ::= <WORD> | <WORDS> <WORD> | <QUOTE> | <WORDS>
<QUOTE>

<ENV_SET_OPERATOR> ::= “=”
<ENV_SET> ::= <WORD><ENV_SET_OPERATOR><WORD>

<SUBCOMMAND> ::= <ENV_SET> | export <ENV_SET> | export <WORD>
| <WORDS>

<REDIRECTION_IN> ::= <IN_OPERATOR> <WORD>
<REDIRECTION_OUT> ::= <OUT_OPERATOR> <WORD>
<REDIRECTION> ::= <REDIRECTION_IN> | <REDIRECTION_OUT>
```

```
<COMMAND> ::= <SUBCOMMAND>
| <SUBCOMMAND> <REDIRECTION>
| <SUBCOMMAND> <REDIRECTION_IN> <REDIRECTION_OUT>

<PIPE> ::= <COMMAND><PIPE_OPERATOR><COMMAND> | <PIPE>
<PIPE_OPERATOR><COMMAND>

<JOB> ::= <COMMAND> | <PIPE> | <COMMAND> <BG_OPERATOR> |
<PIPE> <BG_OPERATOR>

<JOB_SEPARATOR> ::= ";"
<JOBS_LIST> ::= <JOB> | <JOB_SEPARATOR><JOBS_LIST><JOB>
```

Testy i metodyka testowania

W celu przetestowania programu przeprowadzono szereg testów manualnych. Za zachowanie wzorcowe uznano zachowanie interpretera polecenie bash.

1. Zagnieżdżone uruchamianie shella z przekazywaniem zmiennych środowiskowych i wyjściem funkcją exit.
2. Blokowanie sygnałów z poziomu shella, np. CTRL-C.
3. Uruchamianie potoków arbitralnej długości, "`cat Makefile | grep PHONY | grep cpp`".
4. Przekierowanie standardowego wyjścia do pliku, np. "`echo test > plik`".
5. Przekazanie standardowego wejścia do pliku.
6. Obsługa cytowania: wszystko zawarte wewnątrz cudzysłowów traktowane jako jeden token, np. "`echo '| < > &'`".
7. Obsługa separatora ";" (dzielenie na joby), np. "`echo 1; echo 2;`".
8. Zakończenie wiersza symbolem \
9. Wieloliniowy output, np. "`echo '`

   `a'`".
10. Obsługa kodu powrotu, np. "`ls nieznanykatalog; echo $?`" powinno zwrócić 2.
11. Wywoływanie poleceń z argumentami w postaci zapisanych wcześniej zmiennych środowiskowych na przykładzie "ls -l -h".

```
wiktor@/home/wiktor/Desktop/UNIX_projekt/build>
chyba=ls;fajnie=-l;dziala=-h;$chyba $fajnie $dziala
total 1,9M
-rwxrwxr-x 1 wiktor wiktor 1,8M sty 28 07:37 bash
-rw-rw-r-- 1 wiktor wiktor  15K sty 28 07:37 CMakeCache.txt
drwxrwxr-x 5 wiktor wiktor 4,0K sty 28 07:37 CMakeFiles
-rw-rw-r-- 1 wiktor wiktor 1,5K sty 28 07:37 cmake_install.cmake
-rw-rw-r-- 1 wiktor wiktor  12K sty 28 07:37 Makefile
```

12. Wywoływanie poleceń składających się wyłącznie ze zmiennych środowiskowych na przykładzie komendy “ls”.

```
wiktor@/home/wiktor/Desktop/UNIX_projekt/build> un=l;ix=s;$un$ix
bash  CMakeCache.txt  CMakeFiles  cmake_install.cmake  Makefile
```

Pliki pomocnicze:

1. W celu zwiększenia efektywności debugowania został wykorzystany pomocniczy plik zbierający logi
2. Jest on obsługiwany przez plik nagłówkowy Logger.h
3. W tym pliku znajduje się definicja funkcji, która wpisuje do pliku ~/shell.log komunikaty podane przez użytkownika
4. Dodatkowo każdy wiersz w logu zawiera czas systemowy.
5. Projekt jest przenośny i budowany za pomocą pliku CMakeLists.txt i uruchamiany przy użyciu pomocniczego skryptu