Systemy operacyjne

Projekt programistyczny nr 1

„Powłoka uniksowa"

Termin oddawania: 14 grudnia 2024

Przed przystąpieniem do prac programistycznych należy zapoznać się z następującymi materiałami:

- The Linux Programming Interface: 34.5 – 34.7
- Advanced Programming in the Unix Environment: 9.6 – 9.9
- The GNU C Library: Job Control[1]

Wskazówka: Programowanie na ostatnią chwilę jest **bardzo złym** pomysłem. Daj sobie trochę czasu na przemyślenie koncepcji!

Zamieszczanie rozwiązań

Rozwiązania będą zbierane i oceniane przy pomocy systemu GitHub Classroom. Rozwiązania nadsyłane innymi środkami, np. pocztą elektroniczną, nie będą sprawdzane! Prowadzący dostarczy odnośnik do wygenerowania prywatnej wersji repozytorium, w którym należy umieścić rozwiązanie przed upływem terminu oddawania. Rozwiązanie należy umieścić w gałęzi «`master`» swojego repozytorium. W gałęzi «`feedback`» musi się znajdować **niezmodyfikowana** wersja gałęzi «`master`» głównego repozytorium.

Autor zadania zastrzega, że dostarczona binarna wersja rozwiązania może posiadać drobne usterki. Dodatkowo początkowa lista testów automatycznych może nie być pełna. W związku z tym może zajść potrzeba zsynchronizowania repozytorium rozwiązania studenta z repozytorium głównym zadania. W katalogu swojego repozytorium należy wykonać poniższy szereg poleceń, będąc przygotowanym na rozwiązywanie konfliktów:

```
$ git checkout feedback
$ git pull https://github.com/ii-ask/so-shell.git
  «tu należy rozwiązać potencjalne konflikty»
$ git push
$ git checkout master
$ git merge feedback
```

Prowadzący będzie oceniał zmiany plików w prośbie o połączenie (ang. *pull request*) o nazwie «`Feedback`». Inne prośby będą przez sprawdzającego ignorowane. W zakładce „*Files changed*" mają być widać **wyłącznie** zmiany, które są częścią rozwiązania i zostały przygotowane przez studenta.

Zadaniem studenta jest uzupełnienie brakujących fragmentów programu zawartych w wierszach począwszy od dyrektywy «`#ifdef STUDENT`» do najbliższego wystąpienia dyrektywy «`#endif /* !STUDENT */`». Rozwiązanie, w którym zmodyfikowano fragmenty kodu leżące poza wyznaczonymi miejscami, zostanie automatycznie **odrzucone** przez sprawdzarkę!

Sprawdzanie rozwiązań

Po wypchnięciu zmian do repozytorium zostaną automatycznie uruchomione testy poprawności rozwiązania przy użyciu GitHub Actions. Rozwiązania niepoprawnie sformatowane lub generujące błędy albo ostrzeżenia w trakcie kompilacji będą odrzucane. Formatowanie kodu będzie sprawdzane programem «`clang-format`».

[1] `https://www.gnu.org/software/libc/manual/html_node/Job-Control.html`

Oprócz pewnej liczby punktów przyznawanych za zaliczenie testów, rozwiązanie będzie sprawdzane przez prowadzącego grupę zajęciową. Wykonujący **inspekcję kodu** (ang. *code review*) przydzieli punkty za organizację i czytelność kodu. Jeśli będziesz: starannie wybierać nazwy procedur i zmiennych, dbać o przejrzystość struktury programu i prostotę przepływu sterowania, dodawać komentarze do fragmentów, których zrozumienie wymaga większego wysiłku – to możesz liczyć na pełną liczbę punktów.

Po zebraniu wszystkich utworzonych repozytoriów zostanie przeprowadzony test na unikatowość rozwiązań nadesłanych przez studentów – tj. będziemy sprawdzać podobieństwo między rozwiązaniem studenta, a innym rozwiązaniem z bieżącego roku lub poprzednich lat. W przypadku wykrycia znacznego podobieństwa prowadzący ćwiczenio-pracownię podejmą decyzję o skierowaniu sprawy do komisji antyplagiatowej z konsekwencjami wymienionymi w „*zasadach zaliczania zajęć*" dostępnych na stronie przedmiotu w SKOS.

Lekser i parser

Lekser dzieli ciąg znaków na tokeny i zwraca tablicę wskaźników do ciągów znakowych. Specjalne tokeny zastępuje wskaźnikami o wartościach numerycznych (np. «T_BGJOB», «T_OUTPUT», ...). Na przykład: ciąg znaków `"grep foo < in.txt | wc -l &"` zostanie przetworzony do tablicy wartości typu «token_t»: `{"grep", "foo", T_INPUT, "in.txt", T_PIPE, "wc", -l", T_BGJOB, T_NULL}`. Tablicę tokenów można modyfikować w trakcie przetwarzania polecenia.

Implementacja parsera i interpretera wyrażeń niestety nie są od siebie oddzielone. Zakładamy, że «shell» akceptuje wyłącznie poprawnie uformowane wyrażenia według poniższej gramatyki (w formacie dla programu bison). Nie musisz się przejmować zachowaniem powłoki dla źle uformowanych wyrażeń.

```
%start program
%%
program          : pipe_sequence '&'
                 | pipe_sequence
                 ;
pipe_sequence    : command
                 | pipe_sequence '|' command
                 ;
command          : cmd_words cmd_suffix
                 | cmd_words
                 ;
cmd_words        : cmd_words WORD
                 | WORD
                 ;
cmd_suffix       : io_redirect
                 | cmd_suffix io_redirect
                 ;
io_redirect      : '<' WORD
                 | '>' WORD
                 ;
```

Zadania

Zadanie to grupa procesów utworzonych w wyniku przetwarzania pojedynczego polecenia powłoki. Zadanie może składać się z wielu procesów i wtedy nazywamy je potokiem. Zadania dzielimy na pierwszoplanowe (ang. *foreground*) i drugoplanowe (ang. *background*).

Powłoka zawsze oczekuje na zakończenie zadania pierwszoplanowego zanim będzie gotowa do wykonania kolejnego polecenia. Jednym z zadań powłoki jest przypisanie terminala sterującego do grupy procesów zadania pierwszoplanowego. Dzięki temu sterownik terminala wysyła, między innymi, sygnały «SIGINT» i «SIGTSTP» w wyniku naciśnięcia klawiszy «CTRL+C» i «CTRL+Z». Po zakończeniu zadania (stan FINISHED)

lub przeniesieniu go w tło (zadanie drugoplanowe) należy oddać terminal sterujący grupie procesów powłoki. Zadanie pierwszoplanowe ma zarezerwowany slot 0 w tablicy zadań «jobs».

Zadania drugoplanowe można utworzyć na dwa sposoby: albo dopisując znak «&» na koniec polecenia, albo zatrzymując zadanie naciskając klawisze «CTRL+Z». W pierwszym przypadku polecenie jest aktywne (stan RUNNING), natomiast w drugim zatrzymane (stan STOPPED). Powłoka może nadzorować wiele zadań drugoplanowych. Z użyciem procedury «watchjobs» powłoka wyświetla zakończone zadania drugoplanowe przed pokazaniem znaku zachęty (ang. *prompt*).

Zarządzanie zadaniami

Powłoka rozpoznaje kilka wbudowanych poleceń. Procedura «builtin_command» zwraca wartość ujemną, jeśli polecenie nie zostało rozpoznane jako wbudowane. Dla nas najważniejsza jest implementacja poleceń służących do zarządzania zadaniami (ang. *job control*). Parametr w nawiasach kwadratowych jest opcjonalny. Jeśli nie zostanie podany, to zostaje wybrane zadanie o najwyższym numerze.

- `fg [n]`: zmienia zatrzymane lub uruchomione zadanie drugoplanowe na pierwszoplanowe,
- `bg [n]`: zmienia stan zadania drugoplanowego z zatrzymanego na aktywne,
- `kill %n`: uśmierca procesy należące do zadania o podanym numerze,
- `jobs`: wyświetla stan zadań drugoplanowych.

W przypadku polecenia «kill» występuje konflikt nazw z poleceniem zewnętrznym. Jeśli pierwszy argument do «kill» nie zaczyna się od znaku %, to «do_kill» zwraca wartość ujemną.

Polecenia

Powłoka wykonuje polecenia zewnętrzne i wewnętrzne w ramach zlecanych jej zadań. Poniżej widnieje kilka przykładów istotnie różnych poleceń akceptowanych przez powłokę:

- «`cd ..`»: polecenie wewnętrzne «cd»,
- «`cd .. &`»: j.w. ale jako zadanie drugoplanowe w podprocesie (możliwe, ale nie ma sensu),
- «`ls -l`»: polecenie zewnętrzne uruchomione w podprocesie i monitorowane dopóki podproces się nie zakończy lub nie zostanie zatrzymany,
- «`ls -l &`»: polecenie zewnętrzne uruchomione jako zadanie drugoplanowe, powłoka od razu przechodzi do wyświetlenia znaku zachęty,
- «`cd .. | ls -l`»: zadanie pierwszoplanowe, obydwa polecenia są uruchomione w osobnych podprocesach mimo, że pierwsze jest wewnętrzne, a drugie zewnętrzne,
- «`cd .. | ls -l &`»: zadanie drugoplanowe, reszta jak wyżej,
- «`ls -l | wc -l`»: zadanie pierwszoplanowe, procesy muszą być prawidłowo połączone potokiem.

Przekierowania

Zadaniem procedury «do_redir» jest przetworzenie i usunięcie tokenów odpowiedzialnych za przekierowania. Wartość «n» zwracana z «do_redir» to liczba tokenów, które zostaną przekazane do execve(2). Argumenty «intputp» i «outputp» wskazują na miejsce, w którym zostaną zapisane deskryptory plików otwarte przez procedurę «do_redir». Ciąg tokenów {"grep", "foo", T_INPUT, "test.in", T_OUTPUT, "test.out"} (ntokens=6) zostanie przetworzony do {"grep", "foo", T_NULL, T_NULL, T_NULL, T_NULL} (n=2). W wyniku wykonania procedury «do_redir» deskryptor, na który wskazuje «inputp» i «outputp», odpowiada plikowi «test.in» otwartemu do odczytu i plikowi «test.output» otwartemu do zapisu.

W przypadku użycia przekierowania do pliku w zadaniu będącym potokiem zachowanie jest niezdefiniowane, np. «grep foo test.in > test.out | wc -l».

Wycieki deskryptorów plików

Po utworzeniu zadania powłoka powinna mieć otwarte tylko cztery deskryptory plików: «stdin», «stdout», «stderr» i «tty_fd», co można sprawdzić przy pomocy polecenia wydanego w zewnętrznej powłoce:

```
bash$ ls -l /proc/$(pgrep shell)/fd
(...) 0 -> /dev/pts/2
(...) 1 -> /dev/pts/2
(...) 2 -> /dev/pts/2
(...) 3 -> /dev/pts/2
```

W trakcie tworzenia przekierowań i potoków należy zadbać o to, żeby do podprocesów nie trafiały niepożądane deskryptory plików. Innymi słowy powłoka ma przekazywać do podprocesów wyłącznie deskryptory plików «stdin», «stdout» i «stderr». Można to sprawdzić wydając poniższe polecenia, przy czym deskryptor 3 to katalog otwarty przez polecenie «ls».

```
# ls -l /proc/self/fd
(usunięto)
# ls -l /proc/self/fd | cat
(usunięto)
# echo | ls -l /proc/self/fd
(usunięto)
# echo | ls -l /proc/self/fd | cat
(...) 0 -> pipe:[662015934]
(...) 1 -> pipe:[662015935]
(...) 2 -> /dev/pts/2
(...) 3 -> /proc/13200/fd
```

Zadania rozgrzewkowe

Wykonanie poniższego zestawu zadań jest zupełnie opcjonalne. Jednakże Autor zadania poleca to jako ćwiczenie mające na celu wdrożenie się w strukturę projektu.

Zadanie 1. Wykonywanie zewnętrznego polecenia przez powłokę uniksową ma dwa warianty. Gdy w ścieżce do pliku występuje znak ukośnika «/» to zakładamy, że użytkownik podał ścieżkę względną i uruchamiamy execve(2) bezpośrednio. W przeciwnym przypadku musimy odczytać zawartość zmiennej środowiskowej «PATH» przechowującej katalogi oddzielone znakiem dwukropka «:». Każdą ścieżkę katalogu sklejamy z nazwą polecenia (przyda Ci się procedura «strapp» z pliku «lexer.c») i wołamy «execve». Jeśli polecenie nie zostanie znalezione na dysku to «execve» wraca z błędem.

Uzupełnij ciało procedury «external_command» z pliku «command.c» zgodnie z powyższym opisem.

Wskazówka: Do przetwarzania ciągów znakowych użyto funkcji strndup(3) i strcspn(3).

Zadanie 2. Procedura «do_job» w pliku «shell.c» przyjmuje tablicę tokenów, zawierającą polecenie do uruchomienia i przekierowania, oraz rodzaj tworzonego zadania (pierwszoplanowe lub drugoplanowe). Najpierw sprawdza czy podane polecenie należy do zestawu poleceń wbudowanych (ang. *builtin command*). Jeśli nie, to przechodzi do utworzenia zadania i wykonania polecenia zewnętrznego. Po utworzeniu podprocesu i nowej grupy procesów, zadanie jest rejestrowane z użyciem «addjob» i «addproc». Jeśli utworzono zadanie pierwszoplanowe, to należy poczekać na jego zakończenie przy pomocy «monitorjob».

Uzupełnij ciało procedury «do_job» – wykorzystaj funkcje opakowujące wywołania: sigprocmask(2), sigsuspend(2), setpgid(2) i dup2(2). Pamiętaj, że uruchomione zadanie musi prawidłowo reagować na sygnały: «SIGTSTP», «SIGTTIN» i «SIGTTOU», a powłoka musi zamykać niepotrzebne deskryptory plików.

Wskazówka: Jednym z zadań funkcji «monitorjob» jest konfigurowanie terminala przy pomocy tcsetpgrp(3).

Zadanie 3. Powłoka uniksowa dzieli zadania na pierwszoplanowe (ang. *foreground job*) i drugoplanowe (ang. *background job*). Jednocześnie może istnieć tylko jedno zadanie pierwszoplanowe, natomiast zadań drugoplanowych (polecenie zakończone znakiem «&») może być wiele. Zapoznaj się z kodem odpowiedzialnym za obsługę zadań – znajduje się on w pliku «`jobs.c`». Przeczytaj i wyjaśnij co robią procedury «`addjob`», «`watchjobs`», «`jobdone`» i «`killjob`». Zadanie pierwszoplanowe ma zawsze numer 0.

Uzupełnij procedurę obsługi sygnału «`SIGCHLD`». Dla każdego zakończonego dziecka należy znaleźć odpowiedni wpis w tablicy «`jobs`» i wpisać mu kod wyjścia. Wykorzystaj nieblokujący wariant `waitpid(2)`.