Bonusprojekt Shell ohne Standardbibliothek

Dr. Matthias Frank, Dr. Matthias Wübbeling

Ausgabe Mittwoch, 20. Dezember 2023
Abgabe bis Ende der vorletzten Vorlesungswoche
Freitag, 26. Januar 2024, 23:59 Uhr
Vorführung in der letzten Vorlesungswoche vom 29. Januar bis zum 2. Februar 2024

Die Aufgaben auf diesem Bonuszettel sind freiwillig, das heißt, die Punkte erhöhen nicht die Zulassungsgrenze. Alle Punkte, die Sie hier erhalten, fließen aber trotzdem in Ihre Punktzahl für die Zulassung mit ein.

Sie können sich die Bearbeitung der Aufgaben zeitlich frei einteilen. Die Abgabe ist in der vorletzten Vorlesungswoche fällig, damit die Vorstellung in der letzten Vorlesungswoche erfolgen kann.

Einige der benötigten Konzepte werden erst im weiteren Verlauf der Vorlesung behandelt. Teilweise wird es auch noch Aufgaben zu diesen Konzepten geben, so dass es sinnvoll sein kann, mit der Bearbeitung etwas zu warten.

Geben Sie die Bonusaufgabe in Ihrem normalen Gruppen-Repo in dem Ordner **blatt15** ab. Sie müssen die Aufgaben nicht einzeln implementieren, sondern können alles gemeinsam abgeben; geben Sie in einer **README**-Datei aber bitte an, welche der Aufgaben Sie implementiert haben.

Hinweis: Sollten Ihnen Teile der vorherigen Shell-Aufgaben fehlen, finden Sie auf der Vorlesungswebsite sowie in dieser PDF-Datei eine Library **mystdshell.a** mit allen Funktionen der bisherigen Shell-Aufgaben (respektive **realloc.a** nur für die `realloc`-Funktion der Speicherverwaltung) sowie die entsprechende Header-Datei. Um ihre eigene Shell gegen **mystdshell.a** zu linken, reicht dann beispielsweise folgender clang-Aufruf (**realloc.a** analog):

```
clang -o shell shell.c mystdshell.a
```

Setzen Sie dies in Ihrer Makefile korrekt um.

Bonusaufgabe I. Shell-Eingabe (6 Punkte)

Ein wichtiger Anteil einer Shell ist ihr Parser. In dieser Aufgabe wird das Eingabemodul einer extrem simplen UNIX-Shell realisiert

1. Zeileneingabe: Deklarieren Sie in einer Headerdatei **mygetline.h** und implementieren Sie in einer Quelltextdatei **mygetline.c** die folgende Funktion:
   `ssize_t mygetline(char **buf, size_t *buflen, int fd)` – Liest eine Zeile aus dem gegebenen Dateideskriptor und speichert sie in einen dynamisch allokierten Puffer an `*buf`, dessen Größe in `*buflen` liegt. Die Zeile endet bei einem Zeilenvorschub-Zeichen (`'\n'`) oder an einem End of File (welches durch `myread` signalisiert wird, da es in diesem Fall 0 zurückgibt). Die Ausgabe enthält ein eventuelles Zeilenvorschub-Zeichen sowie ein abschließendes Nullbyte. Die Länge der Zeile (einschließlich eines eventuellen Zeilenvorschubs und ausschließlich des Nullbytes) wird zurückgegeben; falls ein Fehler auftritt, wird -1 zurückgegeben. Der Puffer wird mit Ihrem `mymalloc` bzw. `myrealloc` allokiert und muss mit `myfree` freigegeben werden. `*buflen` muss vor dem Aufruf auf die Länge des Puffers gesetzt werden und wird durch `mygetline` entsprechend aktualisiert. Falls vor dem Aufruf `*buf = NULL` und `*buflen = 0` ist, wird ein passender Puffer von `mygetline` allokiert.

   > Hinweis: Sie können mittels `myread` Zeichen lesen. Um Zeichen nach dem Zeilenende nicht zu „verschlucken“, müssen sie die Eingabe in einer Schleife Zeichen für Zeichen einlesen. Um unnötigen Speichermanagement-Aufwand zu vermeiden, sollten Sie etwa, wenn immer der Puffer zu klein für die Zeile wird, ihn auf das Doppelte seiner bisherigen Größe vergrößern.

2. Zeilenzerlegung: Deklarieren Sie in einer Headerdatei **shell.h** und implementieren Sie in einer Quelltextdatei **shell.c** die folgende Funktion:
   `char **shell_split(char *input)` – Zerlegt die gegebene Eingabe an Leerraum-Zeichen in einzelne nichtleere „Wörter“ und gibt ein nullterminiertes Array von Zeigern an die Anfänge der Wörter zurück. Im Fehlerfall gibt dies `NULL` zurück. Um die Enden der Wörter zu markieren, fügt diese Funktion Nullbytes in den übergebenen String ein. Die Leerraum-Zeichen, an denen getrennt wird, sind: Tabulator (`'\t'`), Zeilenvorschub (`'\n'`), Vertikaltabulator (`'\v'`), Seitenvorschub (`'\f'`), Wagenrücklauf (`'\r'`), und Leerzeichen (`' '`). Der zurückgegebene Puffer ist mittels `mymalloc` allokiert und muss mit `myfree` freigegeben werden.

   > Hinweis: Da Sie die Anzahl der Wörter a priori nicht kennen, müssen Sie den Puffer gegebenenfalls dynamisch umallokieren.

Bonusaufgabe II. Shell – Version 1.0 (8 Punkte)

In dieser Aufgabe wird schließlich eine erste tatsächliche Shell realisiert. Dazu holen wir zuerst etwas nach, was in vorherigen Aufgaben ausgelassen wurden.

1. Zahlenkonversion: Es ist für die Shell nützlich, Zahlen für den Benutzer darstellen zu können. Diese Funktionalität setzen wir hier um. Deklarieren Sie dazu in **mystdlib.h** und implementieren Sie in **mystdlib.c** die folgenden Funktionen:
   - `int mystrtoint(const char *input, int *result)` – Konvertiert die gegebene Eingabe, die eine ganze Zahl im Dezimalsystem darstellt, in eine Ganzzahl und gibt diese zurück. `input` ist der Eingabestring, das Ergebnis wird nach `result` geschrieben. Gibt die Anzahl der erfolgreich interpretierten Zeichen zurück, oder 0, falls gar keine Zeichen interpretiert werden konnten. Die Eingabe besteht aus einem optionalen Vorzeichen (`+` oder `-`), gefolgt von einer Folge aus Ziffern (`0`, . . . , `9`), in der mindestens eine Ziffer vorkommen muss.
   - `char *myinttostr(char *output, int value)` – Konvertiert die gegebene Zahl `value` in das Dezimalsystem, schreibt den resultierenden String nach `output` und gibt `output` zurück. Das Format der Ausgabe folgt dem der Eingabe von `mystrtoint`, außer dass kein führendes `+` erzeugt wird. Ein abschließendes Nullbyte wird eingefügt. Der Fall, dass `output` nicht groß genug ist, um das Ergebnis zu speichern, muss nicht extra behandelt werden.

2. Prozesserzeugung: Erweitern Sie ihre Dateien **shell.h** und **shell.c** um eine Deklaration und Implementierung der folgenden Funktionen:
   - `pid_t run_command(char **argv, int no_fork)` – Erzeugt einen Kindprozess aus der Datei an `argv[0]` mit der an `argv` übergebenen Kommandozeile. `argv` ist ein durch einen Nullzeiger terminiertes Array aus nullterminierten Strings. Falls `no_fork` nicht `NULL` ist, wird der Befehl stattdessen im aktuellen Prozess ausgeführt. Gibt die Prozess-ID des Kindprozesses zurück oder 0, falls `no_fork` nicht `NULL` war (und die Funktion überhaupt zurückkehrt). Im Fehlerfall soll ein negativer Wert zurückgegeben werden.
   - `int run_cmdline(char **argv)` – Ruft `run_command` mit der gegebenen Kommandozeile und einem zweiten Argument von 0 auf, und – falls ein Prozess erzeugt wurde – wartet, bis er sich beendet. Gibt 0 zurück oder einen negativen Wert, falls ein Fehler auftritt.

   Verwenden Sie für das Erzeugen von Kindprozessen das aus den Vorlesungsfolien bekannte `fork`-`exec`-Verfahren (nur mit `execve`) und für das Warten auf den Kinderprozess den `mywait4`-Systemaufruf. Für das Auslösen der Systemaufrufe sollten Sie Ihre vorher definierten Wrapperfunktionen verwenden.

   > Hinweis: Der `execve`-Systemaufruf erwartet ein weiteres `run_command` nicht übergebenes Argument (die Umgebungsvariablen in `**envp`). Sie können diese beziehen, indem Sie eine Variable `extern char **environ` (in der Quelltextdatei!) deklarieren und diese als letzten Parameter an `execve` übergeben.

3. Shell: Implementieren Sie nun die `main`-Funktion der Shell. Diese soll in einer Schleife:
   - Einen Prompt, also ein Dollarzeichen (`$`) gefolgt von einem Leerzeichen, mittels `mywrite` auf `STDERR` ausgeben.
   - Eine Eingabezeile mittels `mygetline` von `STDIN` einlesen.
   - Die Zeile mit `shell_split` bzw. `shell_split_ex` in Worte zerlegen.

- Die so ermittelte Kommandozeile mit `run_cmdline` ausführen.

Achten Sie auf Sonderfälle: So sollte sich die Shell etwa bei einem End of File (erkennbar dadurch, dass `mygetline` 0 zurückgibt) beenden, und leere Eingabezeilen sollten ignoriert werden. Außerdem sollen Sie Ressourcenverbrauch (die von `mygetline` und `shell_split` zurückgegeben Puffer müssen freigegeben werden) und Fehlerbehandlung (der Benutzer kann etwa einen nicht existierenden Befehl angeben) beachten. Falls Sie von irgendeiner Operation einen Fehlercode bekommen, können Sie ihn mittels `myinttostr` in eine lesbare Form umwandeln und dem Benutzer anzeigen.

Bonusaufgabe III. Shell Builtins (2 Punkte)

Eine Shell enthält typischerweise einige eingebaute Befehle (builtins). Manche Befehle müssen sogar in die Shell eingebaut sein, da sie ansonsten gar nicht realisierbar sind. In dieser Unteraufgabe führen wir einen exemplarischen eingebauten Befehl in unsere Shell ein.

Ergänzen Sie den Code Ihrer `run_command`-Funktion so, dass der folgende Befehl abgefangen und wie beschrieben ausgeführt wird:

- `cd dir` – Wechselt das aktuelle Arbeitsverzeichnis („change directory“) der Shell zu `dir`. Es muss genau ein Argument angegeben werden (die Kommandozeile muss also genau zwei Worte enthalten, `cd` und das Argument), ansonsten schlägt der Befehl mit einer Fehlermeldung fehl.

Sie sollten, um die Funktion tatsächlich zu bewerkstelligen, Ihre `mychdir`-Funktion verwenden. `cd` wird stets innerhalb der Shell selbst ausgeführt (als ob `no_fork` immer gesetzt wäre).

Bonusaufgabe IV. Shell mit Pipes (8 Punkte)

In dieser Aufgabe ergänzen wir die Shell um Piping, d. i. das Verknüpfen der Standardaus- und -eingaben von Befehlen.

Erweitern Sie die Funktion `run_cmdline` um Piping-Unterstützung. Dazu soll die Funktion wie folgt verfahren:

1. Falls kein Pipe-Zeichen (senkrechter Strich; |) als einzelnes Wort in der Eingabe vorkommt, wird so verfahren wie bisher. Es wird also ganz normal das Programm ausgeführt, und die Bearbeitung der Eingabe ist abgeschlossen.

2. Falls ein Pipe-Zeichen vorkommt, wird stattdessen wie folgt vorgegangen:
   a) Teile die Kommandozeile an Pipe-Zeichen-Worten in Teilbefehle auf.
   b) Führe die einzelnen Teilbefehle in Kinderprozessen aus (auch wenn sie Builtins sind), wobei jeweils benachbarte Paare von Prozessen mit Pipe-Objekten verknüpft werden: Die Standardausgabe (Dateideskriptor 1) des ersten Prozesses wird zum Schreib-Ende der ersten Pipe, die Standardeingabe (Dateideskriptor 0) des zweiten Prozesses wird zum Lese-Ende der ersten Pipe, etc.
   c) Warte, bis alle Kinderprozesse beendet sind.

Erzeugen Sie die Pipes mit dem Systemaufruf `pipe`; Sie sollten dazu Ihre Wrapperfunktion `mypipe` verwenden. Achten Sie darauf, dass alle überflüssigen Dateideskriptoren geschlossen werden; insbesondere müssen die Kinderprozesse und die Shell die jeweils nicht benötigten Enden der Pipe schließen und die Kinderprozesse müssen die Enden der Pipe mittels `dup2` an die jeweils entsprechenden Stellen setzen (und dann die alten Deskriptoren der Pipe-Enden ebenfalls schließen), sowie auf Fehlerbehandlung.