Dokumentation

Command Shell in Unix/Linux

mit Verwendung von **Pipes**

Betriebssysteme Projekt 2

Universität Rostock

Sven Berger, 209204572
Robert Engelke, 209204290
Stephan Rudolph, 209206017
John Trimpop, 209206264

Inhalt

1. Einleitung

1.1 Aufgabenstellung

Im Rahmen des Modul Praktikum Betriebssysteme für den Studiengang Bachelor Informatik der Universität Rostock, sollten zwei Gruppenprojekte, als Realisierung eines C-Programms, durchgeführt werden. Diese Dokumentation bezieht sich auf das zweite der beiden Projekte. Die Hauptaufgabe des Projekts 2 bestand darin, eine eigene Konsole für die Kommunikation verschiedener Prozesse auf einer Unix/Linux-Oberfläche zu realisieren. Für eine gezieltere Umsetzung dieser Aufgabe wurden einige Grundfunktionen von den Praktikumsleitern vorgegeben. (Eine detaillierte Auflistung und Beschreibung dieser Grundfunktionen können Sie in den Rubriken 2.2, 2.3, 3.2 und 3.3 nachlesen).
Des Weiteren durfte sich die Gruppe für eine von zwei vorgegebenen Unteraufgaben entscheiden: *a) Job Control* oder *b) Pipes*. Wir haben uns für die Umsetzung der *Pipes* entschieden. (Ebenfalls detailliert nachzulesen in den Menüpunkten 2.5 und 3.3).

1.2 Nutzen des Programms

Die fertige Konsole lässt sich auf jedem Linux Betriebssystem ausführen und bedienen. Sie dient jedoch hauptsächlich als Lernprojekt für die Entwickler, um mit verschiedenen Unix Systemkommunikationen vertraut gemacht zu werden, wie etwa Prozessen, Prozesskontexten oder unterschiedlichen Methoden der Prozesskommunikation.
Als Konsole mit wenigen Funktionen könnte sie durchaus verwendet werden, jedoch müsste sie, um beispielsweise mit einem Linux Terminal mithalten zu können, noch einigen Erweiterungen unterzogen werden.

1.3 Systemumgebung und Kompatibilität

Für die Implementierung unserer Konsole haben wir unter anderem verschiedene Editoren verwendet und den GCC (GNU C Compiler). Als Systemumgebung wurde die aktuelle Ubuntu-Linux Distribution benutzt, sowie eine Linux Terminal Emulation für Windows namens Cygwin, welches die gleichen Pakete für den GCC verwendet.
Das Projekt ließe sich demnach mit Hilfe des GCC compilieren und auf jeder Linux Distribution ohne Probleme ausführen, sofern diese über eine korrekte Installation des GCC verfügt.

2. Entwicklerdokumentation

2.1 Planung

Zunächst war es wichtig, den Bearbeitungshorizont der verschiedenen gegebenen Aufgaben zu erfassen. Da der Parser vorgegeben war, wurde schnell klar, dass viele der genannten Soll Bedingungen mit weniger Aufwand verbunden waren. Aus diesem Grund bestand der erste Schritt darin, die Implementierungsaufgaben in drei Bereiche zu verteilen:

1. Das korrekte Einbinden und Ansprechen des fertigen Parsers,
2. Die Handhabung der Back- und Foregroundjobs und
3. Die Realisierung der Pipes aus Teilaufgabe a).

Die Verteilung dieser Punkte innerhalb der Gruppenmitglieder erfolgte demnach:

Sven Berger und John Trimpop	**Punkt 1**
Stephan Rudolph	**Punkt 2**
Robert Engelke	**Punkt 3**

Vor der Implementierungsphase sollten jedoch noch einige wichtige Architekturpunkte abgehandelt werden, welche anhand von UML-Diagrammen im nächsten Menüpunkt genauer erläutert werden.

2.2 Architektur und Diagramme

Um eine effiziente Planung durchführen zu können, stellten wir uns zunächst die Frage, auf welche Diagrammarten wir zugreifen könnten.
Da C eine rein imperative Programmiersprache ist, haben wir auf die Verwendung von Klassendiagrammen verzichtet. Auch das Verwenden eines Use-Case-Diagramms erschien uns, da der Benutzer der fertigen Software nur eine Person sein wird, als nicht sinnvoll.
Um die reinen Fähigkeiten der Konsole und einen groben Übersicht über ihre Funktionsweise zu erlangen entschieden wir uns deshalb als Erstes für ein Zustandsdiagramm (s. Abb. 1, Seite 5).
Hier ist die Unabhängigkeit der Fore- und Backgroundjobs klar zu erkennen. Ein Backgroundjob kann beispielsweise nicht durch Betätigen der Tasten STRG+C beendet werden und wird zwangsläufig ausgeführt. Ebenfalls wichtig ist, dass die History auch für fehlerhafte Eingaben angelegt wird und, dass das Programm nach jeder Eingabe zum Ausgangspunkt zurückgelangt: Die Prompt.

Als ein weiteres Diagramm für die Darstellung der Funktionsweise der Pipes hielten wir ein Datenflussdiagramm als passend (s. Abb 2, Seite 5).

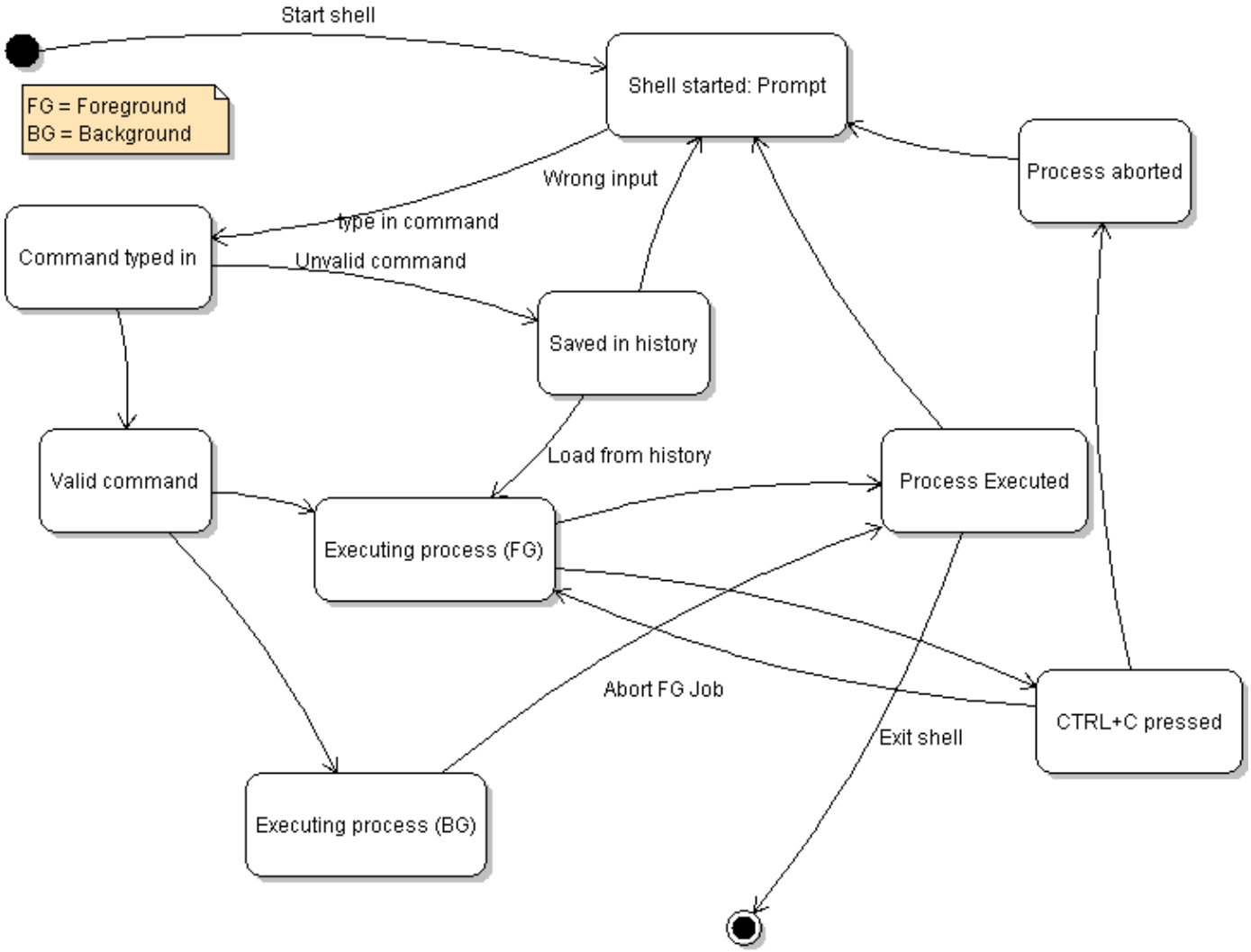

Abbildung 1: Zustandsdiagramm: Programm

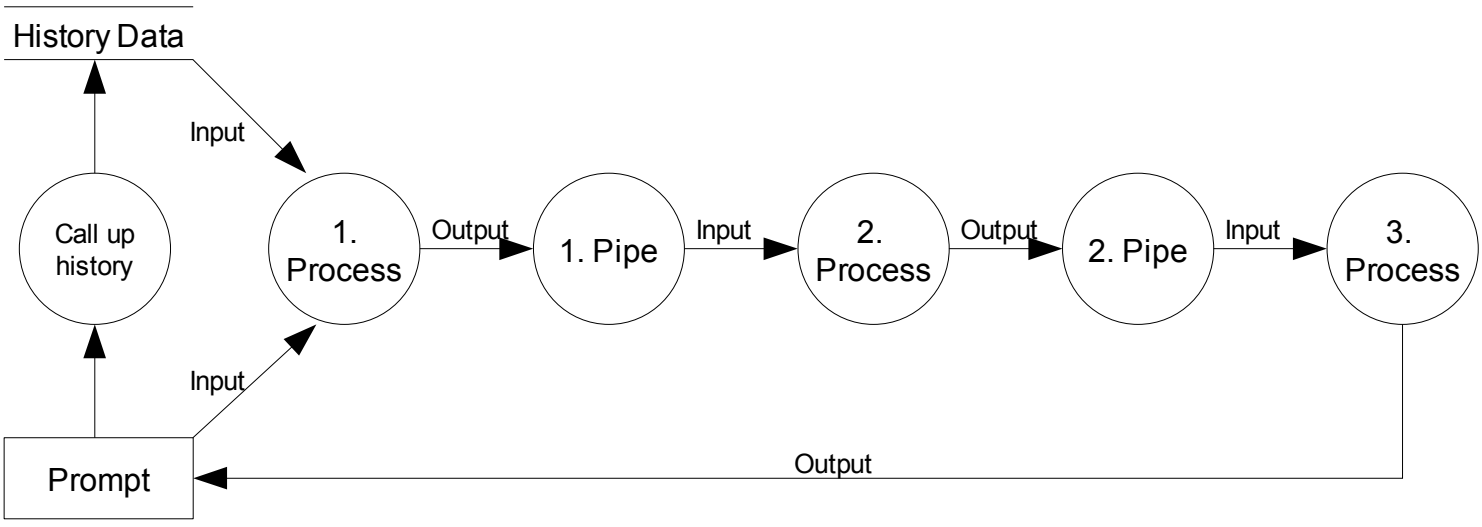

Abbildung 2: Datenflussdiagramm: 3-Prozess-Beispiel

2.3 Implementierung

Nach der Planung konnten wir zur Implementierungsphase übergehen.
Der erste Schritt dieser Phase, war die Zusammenhänge des Parsers zu verstehen. Mithilfe der beiliegenden HTML-Dokumentation und ein wenig Geduld, war dies auch nicht weiter schwer. Wichtig war es nun, die korrekte Art und Weise herauszufinden, wie der Parser angesprochen wird um beispielsweise den einfachsten der Konsolenbefehle umzusetzen: Den Exit-Befehl.
Die Eingabe in die Konsole wird innerhalb des Parsers mit der Charvariabeln `cmd` gesteuert. Um die Eingabe aus unserem Programm nun durch den Parser zu schicken, ist die Codezeile `cmd = parser_parse(line)` nötig (wobei line unsere Eingabezeile beschreibt).
Mit der Methode `parser_parse()` können nun alle nötigen Befehlsfälle abgehandelt werden. Dies geschieht mit dem Pointer der Variabeln `cmd` auf den Token `kind`, für die Art der Eingabe. Die verschiedenen Fälle haben wir danach mit simplen if-else-Bedingungen in einer While-Schleife abgefragt. Für den Exit-Befehl wäre dies zum Beispiel: `if(cmd->kind == EXIT)` [...].
Die While-Schleife wird verlassen, sobald eine Eingabe nicht den Bedingungen entspricht und der Parser damit nichts anfangen kann. Eine Fehlermeldung erscheint und die Prompt wird erneut gestartet.

Der zweite Schritt bestand nun darin, die verschiedenen Arten der Eingaben korrekt verarbeiten zu lassen. Mit Hilfe einiger praktischer C-Bibliotheken, und verschiedenen fertigen Methoden konnte dies an einigen Stellen sogar sehr unproblematisch durchgeführt werden. Exemplarisch lässt sich etwa `chdir()` nennen, womit das aktuelle Ordnerverzeichnis geändert wird, oder `execl()`, mit Hilfe dessen sich Textdateiverwaltungen steuern lassen.

Die Realisierung der Fore- und Backgroundjobs, beziehungsweise der Pipes war eine schwerwiegendere Aufgabe. Diese werden in den beiden herauf folgenden Unterkapiteln näher erläutert.

2.4 Fore- und Background Prozesse

Der wesentliche Unterschied zwischen Vor- und Hintergrundprozessen besteht darin, dass bei Hintergrundprozessen der Nutzer die Konsole noch nutzen kann, um anderweitige Befehle auszuführen.
Wird die `execl()` Funktion aufgerufen, wird der Prozess im Vordergrund ausgeführt.
Um die Funktion jedoch im Hintergrund auszuführen, muss ein neuer „Thread“ gestartet werden. Dies geschieht mittels `fork()`. Bei dem Funktionsaufruf wird der aktuelle Prozess gespiegelt. Dabei werden alle Argumente übergeben.
Das einzige was sich ändert ist die Prozess-ID, die für den Kindprozess erzeugt wird, welcher somit als eigenständiger Prozess vom Betriebssystem geführt wird. Durch den Rückgabewert der `fork()`-Methode kann daraufhin entschieden werden, welche Zweige im Eltern- und welche im Kindprozess ausgeführt werden.

So können die Background-Prozesse ausgeführt werden und das Hauptprogramm trotzdem weiterlaufen. Die Ausgabe, dass ein Background-Prozess fertig ist, erfolgt ebenfalls in diesen Zweigen. Die Kindprozesse müssen dann mittels der `exit()`-Funktion beendet werden, da die Prozesse sonst als sogenannte „Daemon-Prozesse“ im Hintergrund weiterlaufen.

2.5 Realisierung Teilaufgabe a): Pipes

In der Informatik beschreibt der Begriff „Pipe“ einen gepuffert uni- oder bidirektionalen Datenstrom zwischen zwei Prozessen nach dem First-In-First-Out Prinzip. Dies bedeutet, dass der Output des ersten Prozesses sofort als Input des zweiten dient. Falls mehrere Pipes hintereinander stehen, wird dieses Muster fortgeführt: Der Output des zweiten Prozesses, welcher den Output des ersten als Input verwendete, wird als Input des dritten verwendet und so weiter.
Für die Implementierung wird dies mithilfe eines Filedescriptors gemacht. Standardmäßig ist für jedes Programm die Eingabe Stdin und die Ausgabe Stdout. Diese müssen entsprechend umgeleitet werden.
Der Filedescriptor gibt an, wohin die Ein- und Ausgabe geleitet wird. Unter Unix ist der Filedescriptor ein Array aus zwei Integer-Werten. Der eine Wert ist die Adresse der Eingabe und der andere die entsprechende Ausgabeadresse. Für das Programm muss entsprechend die eine Seite mittels `close(file_descriptor[0])` geschlossen werden, sowie die andere Seite mittels `dup2(file_descriptor[1], 1)` mit dem Filedescriptor der Standardausgabe verbunden werden.
Für das sequentiell nächste Programm muss entsprechend die Eingabeseite von der Standardeingabe auf den Filedescriptor gelegt werden.

3. Benutzerdokumentation

3.1 Erste Schritte

Um ein erfolgreiches Starten unserer Konsole zu gewährleisten, muss zuallererst die *shell.c* Datei mit einem über die benötigten Libraries verfügenden GCC compiliert werden (dies ist bei den meisten Linux Distributionen standardmäßig der Fall).
Bevor sie die Shell allerdings compilieren können, müssen Sie jedoch sicherstellen, dass sich alle relevanten Dateien (*shell.c, parser.c* und *parser.h*) in ein und demselben Ordner befinden. Starten Sie nun Ihren Terminal und geben sie folgende Befehlszeile ein:

```
gcc shell.c -o makefile.o parser.c
```

Beachten Sie hier, dass makefile.o ein beliebig gesetzter Name der anschließend auszuführenden Datei ist. Starten Sie jetzt das Programm mit der Zeile:

```
./makefile.o
```

Wenn alles glatt gegangen ist, sollte innerhalb Ihrer Konsole nun eine neue Zeile namens SuperMegaShell: /home/USERNAME > erscheinen. Im Verzeichnis steht bei Ihnen anstelle von USERNAME der aktuelle angemeldete Benutzer Ihres Betriebssystems.

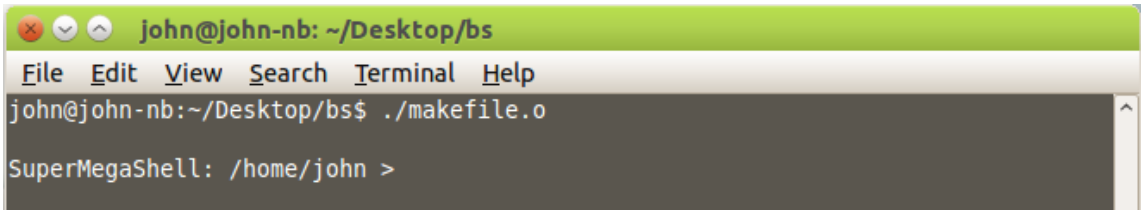

Abbildung 3: Startbildschirm

3.2 Informationen über Befehle

Unsere Konsole ist jetzt in der Lage jegliche integrierte Befehle auszuführen (Sehen Sie dazu Punkt 3.3 für eine Auflistung und Erklärung aller Befehle).
Diese werden grundsätzlich in der Anfangszeile der Konsole eingegeben und mit der Entertaste bestätigt und damit ausgeführt.
Falls ein Befehl nicht korrekt eingegeben worden ist, erscheint eine Fehlermeldung. Mit `STRG+C` beenden Sie einen laufenden Prozess.
Mit Pfeiltaste-oben und -unten können Sie zwischen zuvor eingegeben Befehlen wechseln, da die Konsole für jede Eingabe eine sogenannte History anlegt (auch wenn die Eingabe nicht korrekt war). Eingaben innerhalb eines Befehls, die Sie mit Apostrophs eingrenzen oder einen Backslash voraus stellen (also zum Beispiel `'test'` oder `\test`), werden geschützt und nicht weiter ausgeführt. Ein Rautesymbol (`#`) erstellt einen Kommentar, was bedeutet, dass hinter diesem Zeichen stehende Eingaben bis zum Ende der Zeile ignoriert werden.

Außerdem können Sie mit Hilfe von Semikolons verschiedene Befehle hintereinander ausführen (also zum Beispiel `befehl1 ; befehl2`).

3.3 Auflistung aller weiteren Befehle

`exit`
Beendet die Konsole. Sie können diese erneut starten, indem Sie den erzeugten Makefile wieder ausführen.

`cd` (Change Directory)
Wechselt das derzeit ausgewählte Verzeichnis. Standardmäßig ist dieses /home/USERNAME. Geben Sie ein gültiges Verzeichnis (getrennt von `cd` mit einer Leerzeile) ein. Stellen Sie sicher, dass die Verzeichnisse vor dem Wechseln existieren. Bei falscher Eingabe erscheint eine Fehlermeldung. `cd ..` wechselt in das übergeordnete Verzeichnis.

`setenv X Y / unsetenv X`
`Setenv` definiert eine Umgebungsvariable mit der in künftigen Prozessen gearbeitet werden kann. `X` ist hierbei der Name der Variabeln und `Y` ihr Wert. `unsetenv` entfernt den Wert wieder aus dieser Variable.

`&`-Befehl
Mit Hilfe eines `&`-Zeichens können Sie Prozesse, die bei einem normalen Start im Vordergrund laufen, im Hintergrund starten. Dies bedeutet, dass ein Prozess im Hintergrund weiterläuft, sie aber dennoch einen neuen Befehl ausführen können. Beachten Sie, dass `STRG+C` nur Vordergrundprozesse beenden kann.

`< datei`
Durch Verwendung von < mit Nachstellung einer Leerzeile und eines gültigen Textdateinamens innerhalb eines Befehls, wird der Input dieses Befehls aus dem angegebenen Textdokument gelesen.

`> datei`
Hier erfolgt die Verwendung anders herum, als obige Beschreibung: Das Ergebnis des Befehls wird in ein angegebenes Textdokument hineingeschrieben.

$-Befehl
Was soll das, das check ich nicht.

Pipes (`|`)
Sogenannte Pipes lassen sich mit Hilfe des Mittelstrichs (`|`) angeben. Hierbei handelt es sich um an eine Aneinanderreihung verschiedener Befehle. Das Ergebnis des ersten Befehls dient als Eingabe des zweiten, das des Zweiten wiederum als Eingabe des Dritten und so weiter. Der letzte Befehl gibt das Endergebnis aus.
Beispiel: (`befehl1 | befehl2 | befehl3`)

4. Abschluss und Bewertung

Schnell war zu erkennen, dass dieses Projekt, im Gegensatz zu Projekt 1, tiefer in die Realisierung Betriebssystem ähnlicher Prozesse dringt, und damit weniger mit der Programmiersprache als solches zu tun hat.
Im Großen und Ganzem war dieses Projekt wichtig, um einen richtigen Umgang mit Prozessen und Prozesskommunikation zu erlernen.

Ohne den vorbereiteten Parser und importierte Bibliotheken, hätte der Zeitaufwand jedoch den Erwartungshorizont eines Bachelor Informatik Studenten überschritten.

Dennoch konnten wir leider an manchen Stellen den gestellten Anforderungen nicht entsprechen.
Zwar ist ein Grundgerüst des Programms gut erkennbar und manche kleinere Prozesse wurden umgesetzt, jedoch konnten wir ein wichtiges Merkmal nicht umsetzen: Die Trennung der Fore- und Backgroundjobs.
Pipes wurden zwar fertig implementiert, haben es aufgrund mangelnder Einbindung im Befehlsabruf, jedoch nicht bis in die Testphase geschafft.
Die History wurde fast vollständig umgesetzt, allerdings manuell und ohne Zuhilfenahme der fertigen Library „Readline“, womit man einige Zeit hätte sparen können.