Praktikum Betriebssysteme: Projekt 1 Client/Server Stream Socket Programmierung

Andreas Ruscheinski* Christian Delfs† Fabienne Lambusch‡

3. Juli 2013

Inhaltsverzeichnis

*Matr.-nr.: 211203494
†Matr.-nr.: 211204103
‡Matr.-nr.: 211203538

1 Problem - Aufgabenstellung

Das Ziel des 2. Projektes ist es eine Kommandozeile zu erstellen. Gesteuert wird diese über eine Konsole bzw. Shell. Dabei kann zwischen gewisse Funktionen, enthalten sein sollen, gewählt werden:

- die Prozessverwaltung (Erzeugung, Kontrolle und Terminierung)
- Interprozesskommunikation (Signale, Sockets und Pipes)

Um eine lauffähige Kommandozeile zu erstellen muss folgendes enthalten sein.

1. Es soll eine Eingabeaufforderung mit Eingabebearbeitung und Befehlshistorie erstellt werden.
2. Das Ausführen von Befehlen soll mit beliebigen Argumenten erfolgen.
3. Es soll möglich sein mehrere Befehle hintereinander schreiben zu können. Dies soll durch das Trennen der Befehele, mit einem Simikolon(;), möglich sein.
4. Befehle sollen standartmäßig im Vordergrund ausgeführt werden.
5. Wenn die Befehlseingabe beendet werden soll, geschieht dies mit Hilfe eines Und-Zeichen (&). Dieser Befehl soll dadurch im Hintergrund ausgeführt werden.
6. Die TastenKombination „Strg + C“soll keinen Einfluss auf Vordergrundprogramme oder auf die Kommandozeile selber haben. Jedoch sollen Vordergrundprogramme terminieren.
7. Der Umgang mit Signalen ist ebenfalls wichtig. Diese werden abgefangen und eine entsprechende Nachricht wird ausgegeben.
8. Die Ausgabe von aussagekräftigen Fehlermeldungen soll ebenfalls erfolgen. Beispielsweise bei Syntaxfehlern.
9. Es soll möglich sein, dass die Ein- und Ausgabe (I/O) in eine Datei umgeleitet werden kann. Hierbei ist zu beachten, dass dies beim Einlesen mit dem Kleiner-als-Zeichen (<) und beim Ausgeben durch das Größer-als-Zeichen (>) geschehen soll.
10. Escape-Sequenzen sollen mit einem Backslash (\) realisiert werden.
11. Freitexte werden anhand des einfachen Anführungszeichens (') erkannt.
12. Die Kommentierung wird mittels des Hash-Zeichens (#) realisiert.
13. Grundlegenede Befehle, die von der Kommandozeile ausgeführt werden müssen sind:
    a) „exit“, um die Konsole zu beenden.

b) „cd Pfad“, um das Arbeitsverzeichnis zu ändern.

14. Der Umgang mit Umgebungsvariablen soll ebenfalls unterstützt werden:
    a) Eine Variable wird mit „**setenv** Variable Wert“gesetzt.
    b) Eine Variable wird mit „**unsetenv** Variable “gelöscht.
    c) Eine Variable wird mit „**$** Variable oder **${**Variable**}**“mit einem Variablenwert ersetzt.

15. Option A
    Wir haben uns für die Option A entschieden. Hierbei soll es zusätzlich die Möglichkeit geben, dass der Nutzer Befehle in der Konsole durch das Zeichen „|“ verknüpfen kann. Außerdem sollen Pipes im Vorder- und Hintergrund ausgeführt werden. Hierbei ist die Ausgabe des vorherigen Befehls die Eingabe des darauffolgenden.

2 Vorbetrachtung - Theorie

2.1 Kommandozeile

Die Kommandozeile ist ein Eingabebereich für die Steuerung von Software. Wird von der Kommandozeile gesprochen, ist damit jedoch häufig die Konsole oder das Terminal gemeint. Dieses läuft im Allgemeinen in einem Textmodus ab. Die Kommandozeile wird von einer Shell oder einem Kommandozeileninterpreter gesteuert. Diese führen die entsprechenden Funktionen aus. Welche in der jeweiligen Software vorkommen ist abhängig vom Betriebssystem.
Die Befehle bzw. Kommandos besitzten oft die Form einer Zeichenkette. Diese werden über die Tastatur eingegeben. Bei vielen Konsolen ist es so, dass die Befehle auf englisch verfasst sind. Die Ausführung der Befehle wird meist direkt aus der Zeile durch zusätzlich angegebene Parameter gesteuert. Ein Kommandozeilenprogramm läuft somit typischerweise mit den gegebenen Parametern einmal ab, bevor eine erneute Befehlseingabe möglich ist. Ein automatisiertes Abarbeiten mehrerer Kommandos nennt man Stapelverarbeitung.

2.2 Shell

Die Unix-Shell oder kurz Shell bezeichnet die traditionelle Benutzerschnittstelle unter Unix oder unixoiden Computer-Betriebssystemen. Es ist also eine Software, die den Benutzer in einer bestimmten Form mit dem Computer verbindet. Der Benutzer kann in einer Eingabezeile Kommandos eintippen, die der Computer dann sogleich ausführt. Man spricht darum auch von einem Kommandozeileninterpreter.
In der Regel hat der Benutzer unter Unix die Wahl zwischen verschiedenen Shells. Die typischen Aufgaben der Shell sind die Bereitstellung von Kerneldiensten oder das Anbieten von Oberflächen für Systemkomponenten.

2.3 Prozessverwaltung

Ein Prozess, oder auch Task genannt, ist eine dynamische Folge von Aktionen, um ein Programm auszuführen. Die Prozessverwaltung ist ein Bestandteil von Betriebssystemen. Sie beschreibt die Tätigkeit des Betriebssystems, einzelnen Prozessen Systemresourcen zuzuweisen und wieder zu entziehen, der sogenannte Prozesswechsel. Auf einem Computer werden meist mehrere Prozesse ausgeführt und die Prozessverwaltung ist für die ordnungsgemäße Verwaltung dieser zuständig.

2.4 Interprozesskommunikation - Signale

Signale werden zur Steuerung von Prozessen verwendet. Wie ein Prozess auf ein bestimmtes Signal reagiert, kann entweder vom Entwickler festgelegt oder dem System überlassen werden.
Bei Signalen handelt es sich um asynchrone Ereignisse, die eine Unterbrechung auf der Prozessebene bewirken können. Signale kommen einer Interrupt-Anforderung gleich. Es handelt sich hierbei um ein einfaches Kommunikationsmittel zwischen zwei Prozessen. Diese Interprozesskommunikation ermöglicht es, dass ein Prozess einem anderen Prozess eine Nachricht senden kann. Die Nachrichten selbst ist eine einfache Ganzzahl. Es ist aber auch möglich ein Macro zu verwenden. Ein Macro ist ein symbolischer und aussagekräftiger Namen.
Verwendung finden Signale vorallem in der Synchronisation oder dem Beenden bzw. Unterbrechen von Prozessen. Ebenfalls möglich ist die Ausführung von vordefinierten Aktionen.
Es gibt drei Kategorien von Signalen:

- Systemsignale (Dies sind Signale, die Hardware- und Systemfehler kenntlich machen)
- Gerätesignale
- Benutzerdefinierte Signale

Um die Abarbeitung von Signalen zu realisieren, wird eine Prozesstabelle verwendet. Erhält ein Prozess ein bestimmtes Signal, dann wird dieses als Eintrag in einer solchen Tabelle hinterlegt.
Um auf die Signale reagieren zu können, muss dem Prozess Rechenleistung von der CPU gewährt werden. Ist dies der Fall, wird der Kernel aktiv und gibt Auskunft darüber, wie auf das Signal zu reagieren ist. Als Entwickler ist es möglich festzulegen, wie eine solche Reaktion sein soll.
Dies geschieht, indem das Signal abgefangen wird und den Prozess nicht erreicht. Dies erlaubt, dass auf diese Signale reagiert werden kann oder dass sie ggf. auch ignoriert werden. Ist nichts genaueres spezifiziert, dann wird die Standardaktion auf solche Signale ausgeführt. Eine mögliche Standardaktion könnte das Beenden des Prozesses sein. Ein ebenfalls mögliches Szenario ist das Erstellen eines Core-Dump – einem Speicherabbild des Prozessors, mit dem es möglich ist Fehlerfälle zu rekonstruieren.
Auslöser für Signale sind häufig Programmfehler oder der Benutzter selber. So ist es

möglich bei der Tastenkombination das entsprechende Signal abzufangen und gesondert darauf zu reagieren.
Mit dem Kommando kill ist es möglich einem oder mehreren Prozessen ein Signal zu senden. Das Kommando trap ist das Gegenstück zu kill. Hiermit ist es möglich, auf ein Signal zu reagieren.
Trifft beim ausführenden Script ein Signal ein, wird die Ausführung an der aktuellen Position unterbrochen und die angegebenen Befehle der trap-Anweisung werden ausgeführt. Danach wird mit der Ausführung des Scripts an der unterbrochenen Stelle wieder fortgefahren.

2.5 Pipes und ihre Implementierung

Die Pipes sind eine mit am häufigsten verwendete Interprozesskommunikations-Technik. Es ist ein gepufferter uni- oder bidirektionaler Datenstrom zwischen zwei Prozessen. Die Abarbeitung erfolgt nach dem First-In-First-Out Prinzip. Bei beliebiger Anzahl an Prozessen ist die Ausgabe des vorherigen Prozesses die Eingabe des darauffolgenden.
Um dieses Interprozesskomunikationsprinzip zu implemeniteren, verwendet man einen Filedescriptor. Standartmäßig ist die Eingabe eines Programmes „stdin“und die daraus resultierende Ausgabe „stdout“.
Um diese mit einander zu verknüpfen, müssen sie umgeleitet werden. An dieser Stelle kommt der Filedescriptor ins Spiel. Dieser legt fest, wohin die Ein- und Ausgaben geleitet werden. Unter Unix besteht ein solcher Filedescriptor aus einem Array mit zwei Feldern, jeweils einem für Ein- und Ausgabe. Die Werte sind die entsprechenden Adressen.
Um die Pipe zu konstruieren, muss die eine Seite geschlossen werden, sowie die andere Seite mit dem Filedescriptor der Standardausgabe verbunden werden. Indem die zusammengehörigen Ein- und Ausgabefelder aufeinander gelegt werden wird die Arbeitsweise der Pipe realisiert.
Ist es für die Bearbeitung eines Befehls in einer Stufe der Pipeline notwendig, dass ein Befehl, der sich weiter vorne in der Pipeline befindet, zuerst abgearbeitet wird, so spricht man von Abhängigkeiten. Diese können zu Konflikten führen. Konflikte erfordern es, dass entsprechende Befehle am Anfang der Pipeline warten, was „Lücken“ in der Pipeline erzeugt. Dies führt dazu, dass die Pipeline nicht optimal ausgelastet ist und der Durchsatz sinkt. Daher ist man bemüht, diese Konflikte so weit wie möglich zu vermeiden.
Ein Kontrollflusskonflikt, ist ein Konflikt, bei dem die Pipeline abwarten muss, ob ein bedingter Sprung ausgeführt wird oder nicht. Die Anzahl Kontrollflusskonflikte lässt sich durch eine Sprungvorhersage reduzieren. Hierbei wird spekulativ weitergerechnet, bis feststeht, ob sich die Vorhersage als richtig erwiesen hat. Im Falle einer falschen Sprungvorhersage müssen in der Zwischenzeit ausgeführte Befehle verworfen werden (pipeline flush), was besonders bei Architekturen mit langer Pipeline viel Zeit kostet.

2.6 Vor- und Nachteile von Pipes

Vorteile von Pipes:

- Es handelt sich um ein recht einfaches gut umsetztbares Konzept.

- Mit Pipes ist es möglich sehr starke Steigerungen der Verarbeitungsgeschwindigkeit zu erreichen.
- Der Gewinn durch Pipelining ist umso größer, je höher die Anzahl der Befehle zwischen Kontrollflussänderungen ist. Da die Pipeline erst nach längerer Benutzung unter Volllast wieder geflusht werden muss.

Nachteile von Pipes:

- Die Prozessrichtung geht nur in eine Richtung.
- Es befinden sich sehr viele Befehle gleichzeitig in Bearbeitung.
- Im Falle eines Pipeline-Flushes müssen alle Befehle in der Pipeline verworfen und die Pipeline anschließend neu gefüllt werden. Dies bedarf des Nachladens von Befehlen aus dem Arbeitsspeicher oder dem Befehlscache der CPU, so dass sich hohe Latenzzeiten ergeben, in denen der Prozessor untätig ist.

2.7 Vorder- und Hintergrundprozesse

Prozesse können einzeln (synchron) oder parallel nebeneinander (asynchron) ausgeführt werden. Wenn ein synchrones Kommando ausgeführt wird, wartet die Shell bis der Befehl abgearbeitet worden ist, bevor sie weitere Eingaben akzeptiert. Dies bezeichnet man auch als einen Prozess im Vordergrund laufen zu lassen. Asynchrone Kommandos laufen ab, während die Shell weitere Kommandos ausführt. Dieses Kommando läuft dann im Hintergrund.
Der wesentliche Unterschied zwischen Vor- und Hintergrundprozessen besteht folglich darin, dass bei Hintergrundprozessen der Nutzer die Konsole noch nutzen kann, um anderweitige Befehle auszuführen. Dies ist nur dann sinnvoll, wenn der Prozess eine längere Zeit braucht, keine/kaum Eingaben abfragt und nichts/wenig im Terminal ausgibt. Beispiele für solche Prozesse wären langwierige Suchprozesse, das Klonen von Partitionen oder ähnliche.

2.8 Prozesswechsel

Ein Prozess wird über einen PCB (process control block) verwaltet, der alle relevanten Informationen enthält und über den er identifiziert wird. Ein PCB ist ein Datensatz mit folgenden Inhalten:

- eine eindeutige Prozessnummer
- Startadresse und Größe des Codebereichs
- Startadresse und Größe des Datenbereichs
- Kopie aller CPU-Register (Prozesszustandsinformation)
- Priorität des Prozesses (sofern vorhanden)

- Zeiger auf den nächsten PCB
- weitere betriebssystemspezifische Informationen

Bei jedem Prozesswechsel wird der Zustand eines Prozesses gesichert. Ein Prozesswechsel findet bei einem Interrupt statt. Der Zustand eines Prozesses besteht aus den aktuellen Inhalten der CPU-Register. Die Sicherung dieser Daten findet im PCB statt.
Nach Abarbeitung der Interrupt-Service-Routine wird der Zustand des aktuellen Prozesses aus den im PCB gespeicherten Daten wiederhergestellt. Falls die Interrupt-Service-Routine einen Prozesswechsel durchführt, wird der Zustand dieses neuen Prozesses wiederhergestellt.
Die Prozesse selbst merken nichts davon, dass sie zwischenzeitlich unterbrochen wurden. Wenn die Prozesswechsel schnell genug stattfinden, bemerken es auch die Anwender nicht.

3 Praktische Umsetzung

3.1 Erste Schritte

Als erstes war es notwendig die Arbeitsweise des vorgegebenen readline Beispiels und des Parsers zu verstehen. Es wurde schnell deutlich, dass dieser einen gewissen Teil der Arbeit abnehmen würde. Nun musste herausgefunden werden, wie dieser korrekt angesprochen werden kann. Solange die Shell laufen soll, befindet sich der Programmablauf innerhalb der main-Methode in einer while Schleife. Der Logische Ausdruck ist solange erfüllt, bis der Befehl „exit“diesen auf true setzt und das Programm deshalb terminiert.
Die Eingabe des Benutzers in die Shell wird im Parser mit einer Struktur namens cmds behandelt. In der Shell benutzen wir als Namen der Variablen „command“.
Durch „line = readline(prompt)“wird die Eingabe des Nutzers in die Variable line gelesen. Der Parser soll nun mit Eingabe der Variable line parsen, durch den Befehl „command = parser_parse(line)“wird dieser aufgerufen.
Man erhält nun durch die Struktur cmds die Möglichkeit einer Fallunterscheidung nach Art des Kommandos, also ob exit, change directory, Umgang mit Environment Variablen, ein Programm oder eine Pipe eingegeben wurde.
Mit Semikolon abgetrennte, sequentielle Eingaben werden in dieser Struktur ebenfalls schon so getrennt, dass auf den nächsten Befehl mittels "command->nextßugegriffen werden kann. Sequentielle Befehle erforderten zur Behandlung somit nur noch eine while-Schleife. Einige Aufgaben, wie z.B. Kommentarfunktion oder Escape-Sequenzen, wurden durch den Parser bereits realisiert.

3.2 Bibliothek readline

Um die Shell ordnungsgemäß ausführen zu können, wird die readline Bibliothek vorausgesetzt.
Diese Programmbibliothek stellt Funktionen zum Bearbeiten von Zeilen zur Verfügung.

So lässt sich beispielsweise die History anlegen und verwalten. Außerdem wird so die Benutzereingabe entgegengenommen.

3.3 Prozessverwaltung

Fork() ist der einzige Weg, um unter POSIX einen neuen Prozess zu erzeugen. Er erzeugt eine exakte Kopie des aufrufenden Prozesses, einschließlich aller Dateideskriptoren, der Register etc. Nachdem fork beendet ist, laufen der ursprüngliche und der neu erzeugte Prozess (Eltern- und Kindprozess) getrennt voneinander weiter. Der Rückgabewert von fork ist entweder die Null, wenn er im neu erzeugten Kindprozess abgefragt, oder gleich der Prozessnummer des Kindprozesses im Elternprozess.
Zum Beenden eines Prozesses wird exit() benutzt, da vor dem eigentlichen Beenden dafür gesorgt wird, dass alle Dateien geschlossen werden.
Die Funktionen der exec-Familie wurden in unserem Projekt an mehreren Stellen verwendet. Sie finden Anwendung, indem mit fork() ein neuer Kindprozess erzeugt und mit Aufruf einer exec-Funktion durch das neue angegebene Programm, komplett ersetzt wird. Außerdem bewirkt ein exec-Aufruf auch keine Änderung der PID, da ja kein neuer Prozess erzeugt wird, sondern eben nur die Segmente des aktuellen Prozesses durch die Segmente des neuen Prozesses überschrieben werden. Es gibt sechs verschiedene exec-Funktionen, die sich nur geringfügig anhand der Parameter voneinander unterscheiden. Wir verwenden execvp() und execvpe().
Wenn auf einen Prozess gewartet werden muss erfolgt dies folgender maßen: Die Funktion wait() wartet, bis sich ein Kindprozess beendet, und gibt dessen PID als Rückgabewert zurück. Bei Fehler wird -1 zurückgegeben. Die Funktion waitpid() hingegen wartet auf einen bestimmten Kindprozess mit der in pid als erstes Argument angegebenen Prozess-ID. Mit waitpid() lassen sich Zombie-Prozesse verhindern.

3.4 I/O Umleitungen

Input von einer Datei und Output in eine Datei werden wie andere Programme mittels fork() und exec behandelt. Der entscheidende Unterschied liegt hier jedoch dabei, dass nach dem Starten eines eigenen Prozesses durch fork() der Standard In (sdtin) bzw. Standard Out (stdout) durch eine Datei festgelegt wird. Dazu haben wir die Funktion freopen genutzt, die nach Angabe des Dateipfades als String den I/O direkt umleitet. Den Pfad erhält man mittels der vorgegebenen Struktur über command->prog.input bzw. output. Nach Ausführung des Befehls wird der Standard In/Out zurückgesetzt, so dass es nicht nötig ist ein fclose(..) zu benutzen.

3.5 Pipes

Für die Implementierung der Pipes wird zu Beginn eine Pipe erstellt. Diese besteht aus einem Eingang und einen Ausgang für die Prozesskommunikation. Für die effektive Nutzung der Pipes ist es notwendig einen Kindprozess zu erzeugen, welcher mittels der Pipe mit dem Elternprozess kommuniziert. Für die Interaktion schreibt der Kindprozess

etwas in die Pipem, worauf der Elternprozess den Input für die Weiterverarbeitung des Programmes nutzt. Vor der Ausführung der Systembefehle wird im Kindprozess mittels dup2 der gesamte Stdout auf den Eingang der Pipe gelegt, wel.cher im Elternprozess, durch Verknüpfung von Stdin und dem Ausgang, weiter behandelt wird

4 Nutzerhandbuch

4.1 Inbetriebnahme

Als Betriebssystem wird eine Linux Distribution vorrausgesetzt. Um das C Programm zu kompilieren, führt der Nutzer im Terminal „make install“ aus. Wichtig ist, dass er sich im richtigen Verzeichnis befindet - in dem sich auch das Skrip „makefile“ befindet. Nun wird mit „./shell“ die Shell gestaret. Es ist außerdem darauf zu achten, dass die Bibliotheken -lreadline -lncurses vorhanden sind.

4.2 Ausführen der Kommandozeile

Bei der Eingabe von nicht korrekt eingegebenen Befehlen, erscheint eine Fehlermeldung.
Mit [Strg] + [c] beenden Sie einen laufenden Prozess.
Mit Pfeiltaste-oben und -unten können Sie zwischen zuvor eingegeben Befehlen wechseln.
Die Konsole legt für jede Eingabe eine sogenannte History an. Dies geschieht selbst dann, wenn die Eingabe nicht korrekt war. So ist es möglich Tippfehler zu korrigieren.
Eingaben innerhalb eines Befehls, die Sie mit Apostrophs eingrenzen oder einen Backslash voraus stellen, 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.

4.3 Übersicht über Kommandobefehle (Manpage-Stil)

4.3.1 exit

Der Befehl exit beendet die Komandozeile und der Nutzer kehrt in die Konsole zurück in der die Komandozeile ausgeführt wurde.

4.3.2 cd

Der Befehl cd steht für „Change Directory“. Folglich ist es möglich mit diesem Befehl in ein existierendes Verzeichnis zu wechseln. Existiert kein Verzeichnis mit dem entsprechend eingegebenen Parameter wird eine Fehlermeldung ausgegeben. Wenn anstelle eines Parameters zwei Punkte eingegeben werden (cd ..) wird in das übergeordnete Verzeichnis gewechselt.

4.3.3 setenv

Mit diesem Befehl können Umgebungsvariablen gesetzt oder verändert werden.

4.3.4 unsetenv

Dieser Befehl ist das Gegenstück zu setenv und ermöglicht das löschen von Umgebungsvariablen. Im Anschluss wird der Speicher wieder freigegeben.

4.3.5 &-Befehl

Befindet sich am Ende eines Befehls das &-Zeichen, so wird der Prozess im Hintergrund gestartet. Die Tastenkombination [Strg] + [c] beendet jedoch nur Vordergrundprozesse.

4.3.6 <-Befehl

Das <-Zeichen ermöglicht es, dass der auszuführende Befehl aus einer Datei geladen wird. Einlesen eines Parameters für einen angegebenen Befehl aus Datei: Befehl < Datei.

4.3.7 >-Befehl

Auch hier findet eine Interatkion mit Dateien statt. Es wird das Ergebnis eines Prozesses in eine Datei geschrieben.

4.3.8 | (Pipe)

Wie oben beschrieben erlaubt der Pipe-Befehl, dass mehrere Prozesse hintereinander ausgeführt werden. Wie bei beispielsweise bei „Befehl1 | Befehl2 | Befehl 3 | ... | Befehl n“können beliebig viele Befehle hintereinander ausgeführt werden, wobei Ausgaben vom (i)-ten die Eingabe des (i+1)-ten sind.

4.4 Formale Spezifizierung der Konsole (EBNF)

Befehl = zweiBefehle | hintergrundBefehl | ausDateiBefehl | inDateiBefehl | pipeBefehl | kommentarBefehl | Kommando

stdBefehl = zweiBefehle | ausDateiBefehl | inDateiBefehl | pipeBefehl | Kommando

zweiBefehle = stdBefehl + „;“+ Befehl

hintergrundBefehl = stdBefehl + „&“

ausDateiBefehl = stdBefehl + „<“+ „Datei“

inDateiBefehl = stdBefehl„>“+ „Datei“

pipeBefehl = stdBefehl + „|“+ Kommando

kommentarBefehl = stdBefehl + „#“+ Kommentar

Kommando = „exit“| „cd“+ Pfad | „setenv“+ Variable + Wert | „unsetenv“+ Variable | Programm | Systemprogramme

Kommentar, Pfad, Variable, Programm = String

Wert = Int