GNU Bourne-Again Shell

Eine Einführung

Michele Corazza, Fabian Becker

FH Giessen-Friedberg

18. Oktober 2010

Die GNU-POSIX-Shell BASH

- Kommandozeileninterpreter
- *eine* von vielen Shells
- nur ein Programm..

Funktionen der Bash

- ausführen von Programmen
  - einzeln
  - mehrere gleichzeitig
  - mit Abhängigkeiten..

Operatoren

- „;“ sequenziell: ls ; pwd
- „&&“ ausführen bei Erfolg: g++ pipe.cc -o pipe && ./pipe
- „||“ ausführen bei Misserfolg: test -d dir || mkdir dir

Interne Bash Kommandos

intern

„cd“ wechselt das Verzeichniss

extern

„pwd“ zeigt den aktuellen Verzeichnisspfad

Test

„whereis <Programm>“
whereis cd =>cd:
whereis pwd =>pwd: /bin/pwd

Exit Codes

Konventionen

- 0 = Alles OK
- >0 = Fehler

Ausgabe des letzten Exit Codes

```
echo $?
```

Weitere Shell Funktionen - Shellprogrammierung

- Shebang #!/bin/sh
- `if, while, for, case`

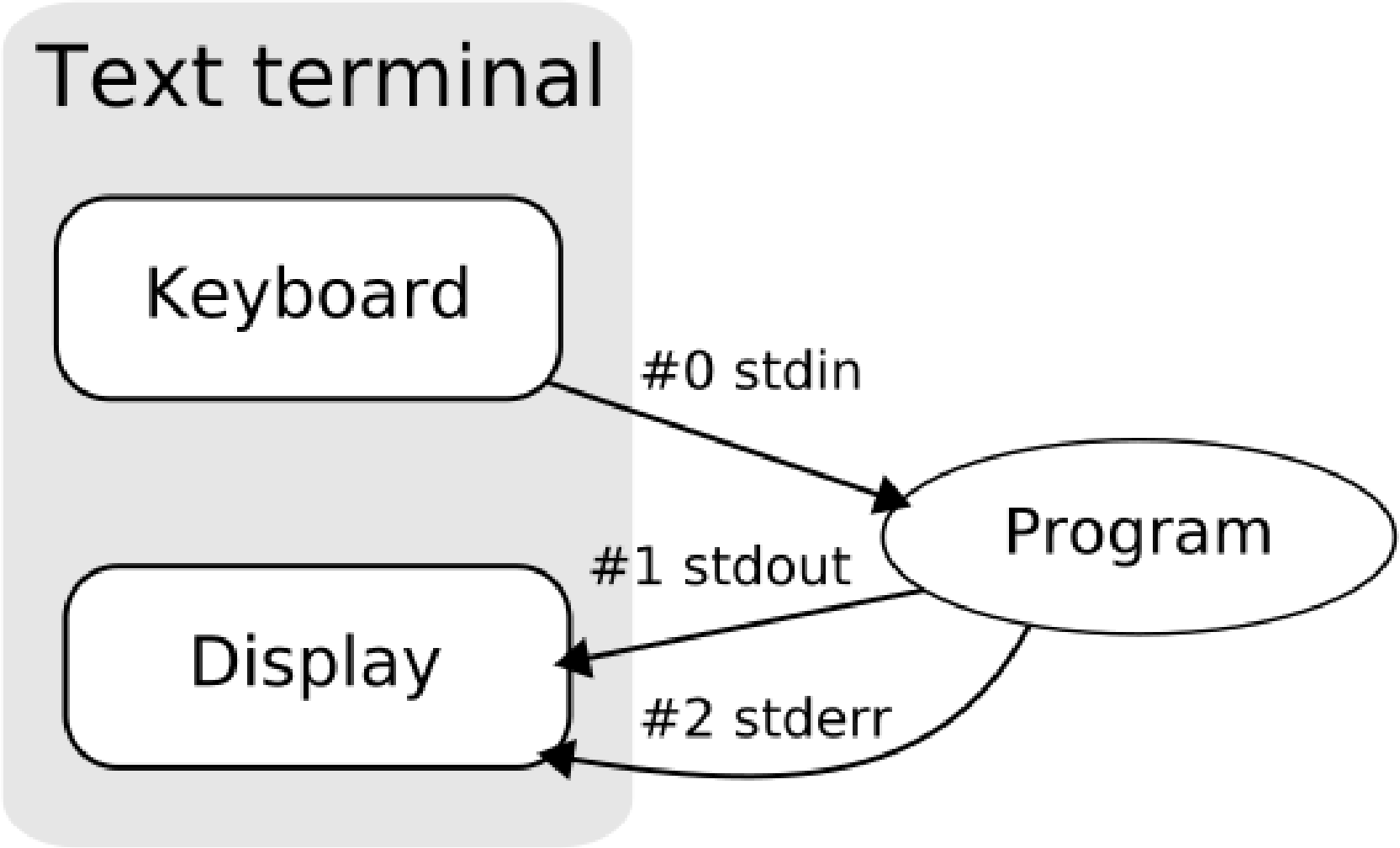
Warum Shell Scripting?

- Automatisierte Verarbeitung von Daten (Batch)
- Filtern

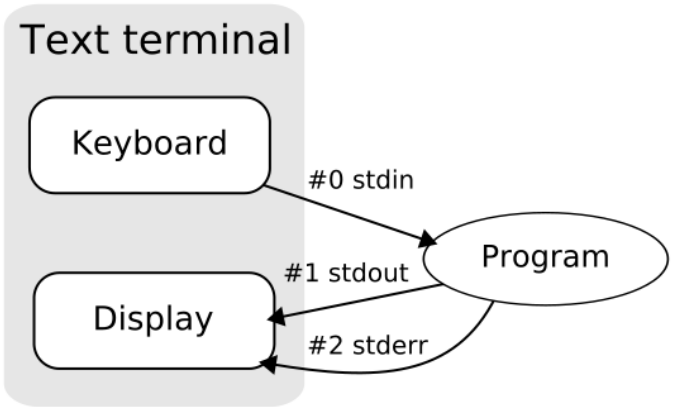
Weitere Shell Kommandos

$1 $2 $3 Erster Zweiter Dritter Übergabeparameter
$# Anzahl der Übergabeparameter

Ein / Ausgabe Ströme



Ein / Ausgabe Umlenkung



Warum?

- Fehlerausgabe in externe Datei lenken
- Ausgabe des Programmes unterdrücken
- Eingabe des Programmes per Datei (automatische Verarbeitung)

Ein / Ausgabe Umleitung in der Shell

- Standard Ausgabe Umlenken >
- Standard Eingabe Umlenken <
- Standard Fehler Umlenken 2>

Datei erstellen / anfügen

Beispiel: ls /home > dateixy
an Datei anfügen: ls /home >> dateixy

Ein / Ausgabe Umleitung in C++

- Ein / Ausgabe Streams sind Filedeskriptoren (0,1,2)
  - Unix/Linux alles ist eine Datei
- werden als normale `ints` behandelt
- Umlenkung per `dup/dup2`

`dup2` Syntax

```
int dup2(int oldfd, int newfd)
```

Rückgabewert: der neue Filedeskriptor

Pipeline

Unnamed Pipes

- Standard Pipes in C/C++
- haben keinen Filedeskriptor
- kommen zum Einsatz wenn ein Kindprozess in die Pipe schreiben soll

Named Pipes

- haben einen globalen Verzeichnis Eintrag
- werden immer dann gebraucht wenn die Prozesse unabhängig voneinader sind (kein Fork())

pipe C-Call

```
int fd[2];
pipe(fd);
```

`fd[0]` lesen, `fd[1]` schreiben

Systemcall fork()

Eigenschaften von fork()

- Erstellt eine Kopie des Prozesses

Systemcall fork()

Eigenschaften von fork()

- Erstellt eine Kopie des Prozesses
- Aufrufender Prozess wird "parent"genannt, die Kopie child

Systemcall fork()

Eigenschaften von fork()

- Erstellt eine Kopie des Prozesses
- Aufrufender Prozess wird "parent"genannt, die Kopie child
- fork() liefert im parent die PID des Kindprozesses, im Kindprozess 0 zurück

Systemcall fork()

Eigenschaften von fork()

- Erstellt eine Kopie des Prozesses
- Aufrufender Prozess wird "parent"genannt, die Kopie child
- fork() liefert im parent die PID des Kindprozesses, im Kindprozess 0 zurück

Kann fork() fehlschlagen? Wenn ja, wann?

Fehlschlagen von fork()

fork() kann fehlschlagen, wenn..

- nicht genügend freier Arbeitspeicher verfügbar ist (EAGAIN)

Fehlschlagen von fork()

fork() kann fehlschlagen, wenn..

- nicht genügend freier Arbeitspeicher verfügbar ist (EAGAIN)
- das Prozesslimit des Benutzers erreicht wurde (EAGAIN)

Fehlschlagen von fork()

fork() kann fehlschlagen, wenn..

- nicht genügend freier Arbeitspeicher verfügbar ist (EAGAIN)
- das Prozesslimit des Benutzers erreicht wurde (EAGAIN)
- die nötigen Kernelstrukturen nicht allokiert werden konnten (ENOMEM)

In diesen Fällen liefert fork() -1 im parent zurück und es wird kein Kindprozess erstellt. errno wird entsprechend gesetzt.

Beispiel

Listing 1: Einfaches fork() Beispiel

```
 1 #include <unistd.h>
 2 #include <stdlib.h>
 3 #include <stdio.h>
 4 #include <sys/types.h>
 5 int main(void) {
 6 pid_t fork_pid;
 7 switch(fork_pid=fork()) {
 8 case -1: printf("Fehler bei fork!\n");
 9 exit(1);
10 case 0: printf("Kind: PID=%d\n", getpid());
11 printf("Kind: Eltern-PID=%d\n", getppid());
12 break;
13 default: printf("Elternproz.: Kind-PID=%d\n", fork_pid);
14 }
15
16 exit(0);
17 }
```

Ausgabe

fork() kann fehlschlagen, wenn..

Elternproz.: Kind-PID=18426
Kind: PID=18426
Kind: Eltern-PID=18425

Kopiervorgang

Was passiert beim Kopieren?

Kopiervorgang

Was passiert beim Kopieren?

- Variablenwerte sind identisch

Kopiervorgang

Was passiert beim Kopieren?

- Variablenwerte sind identisch
- Gleicher Programmzähler

Kopiervorgang

Was passiert beim Kopieren?

- Variablenwerte sind identisch
- Gleicher Programmzähler
- Gleiche Dateideskriptoren (!)

Kopiervorgang

Was passiert beim Kopieren?

- Variablenwerte sind identisch
- Gleicher Programmzähler
- Gleiche Dateideskriptoren (!)
- Gleiche Zugriffsrechte / Eigentümer

Ausführen eines neuen Programms

exec()

Ausführen eines neuen Programms

exec()

- Mit den exec-Funktionen wird ein neues Programm ausgeführt

Ausführen eines neuen Programms

exec()

- Mit den exec-Funktionen wird ein neues Programm ausgeführt
- Kein neuer Prozess! Das alte Programm wird im Speicher ersetzt.

Ausführen eines neuen Programms

exec()

- Mit den exec-Funktionen wird ein neues Programm ausgeführt
- Kein neuer Prozess! Das alte Programm wird im Speicher ersetzt.
- Keine Rückkehr zum aufrufenden Programm.

Ausführung von execl()

execl()

Ausführung von execl()

- int execl(const char *path, const char *arg, ...); // unistd.h

execl()

Ausführung von execl()

- int execl(const char *path, const char *arg, ...); // unistd.h
- Aufruf: execl(„ls“, „ls“, „-al“, „/home“, (char *)NULL);

execl()

Ausführung von execl()

- int execl(const char *path, const char *arg, ...); // unistd.h
- Aufruf: execl(„ls“, „ls“, „-al“, „/home“, (char *)NULL);
- Variable Anzahl an Parametern!

execl()

Ausführung von execl()

- int execl(const char *path, const char *arg, ...); // unistd.h
- Aufruf: execl(„ls“, „ls“, „-al“, „/home“, (char *)NULL);
- Variable Anzahl an Parametern!
- Aufruf scheitert vermutlich, da execl nur im lokalen Verzeichnis nach ls sucht.

execlp()

execlp()

execlp()

execlp()

- Bei execl() muss immer der gesammte Pfad angegeben werden.

execlp()

execlp()

- Bei execl() muss immer der gesammte Pfad angegeben werden.
- int execlp(const char *file, const char *arg, ...);

execlp()

execlp()

- Bei execl() muss immer der gesammte Pfad angegeben werden.
- int execlp(const char *file, const char *arg, ...);
- Sucht zusätzlich noch im PATH.

Listing 2: exec-Funktionen

```
1 #include <unistd.h>
2 extern char **environ;
3 int execl(const char *path, const char *arg0, ... /*,
4 (char *)0 */);
5 int execv(const char *path, char *const argv[]);
6 int execle(const char *path, const char *arg0, ... /*,
7 (char *)0, char *const envp[]*/);
8 int execve(const char *path, char *const argv[],
9 char *const envp[]);
10 int execlp(const char *file, const char *arg0, ... /*,
11 (char *)0 */);
12 int execvp(const char *file, char *const argv[]);
```

Warten auf den Kindprozess

Die Funktion waitpid()

- pid_t waitpid(pid_t pid, int status, int options);
- Warten auf Subprozessterminierung
- Durch Optionen lässt sich abfragen ob ein Subprozess terminiert ist ohne dabei zu blockieren.
- Durch Makros aus wait.h lassen sich Terminierungsstatus abfragen.

Abfragen des Terminierungsstatus

- waitpid(kind_pid, &kind_status, 0);
- WIFEXITED(kind_status) $\Rightarrow$ Normal Terminiert?
- WIFSIGNALED(kind_status) $\Rightarrow$ Terminiert durch Signal?
- WTERMSIG(kind_status) $\Rightarrow$ Welches Signal? (numerisch)

Zombies!

Was sind Zombies?

- Prozess hat Ausführung beendet
- Eintrag in der Prozesstabelle ist noch vorhanden!

Zombies!

Was sind Zombies?

- Prozess hat Ausführung beendet
- Eintrag in der Prozesstabelle ist noch vorhanden!

Wie wird man Zombies los?

- Abfrage des Terminierungsstatus durch wait/waitpid
- Kann mit Signals (SIGCHLD) vereinfacht werden.