Jörg Schilling

Bourne Shell, Korn Shell, POSIX und weiter

Fokus Fraunhofer

Alte Shells unter UNIX

- Um 1970: Thompson Shell
  - If … goto als externe Kommandos, mit lseek(0)
  - Keine Variablen $0, $1, …
  - /etc/glob für wildchart Zeichen in Filenamen
- 1975 Mashey Shell als Patch auf Thompson Shell (osh)
- 1976 erste Version des Bourne Shell auch als Patch
  - Shell als Sprache
  - Variable Subst., Signal Mgt., control flow

Shells auf BourneShell-Code Basis

- Bourne Shell seit 1976, 1979 erste verteilte Version
- Bourne Shell 1981 #, [!...], ${var:[-+?=]val}, set
- Bourne Shell 1983 shift
- Bourne Shell 1984 Funktionen, unset/echo/type, buitlin redirect
- Bourne Shell 1986 $@, 8-bit-clean, getopts, rekursive Funktionen
- Bourne Shell 1989 Job-control
- Bourne Shell 2006 Schily Bourn Shell (bosh) viele Erweiterungen
- Korn Shell 1983 auf Basis der Bourne Shell Source
- Korn Shell 1986 mit eingebautem History Editor
- Korn Shell 1988 erweiterte Version, erkennbare Änderungen

Shells auf Bourne-/Kon-Shell-Code Basis

- Korn Shell 1993 Neue Version weitere Änderungen
- Korn Shell 1997 erste Version mit Source
- Korn Shell 2001 erste echte OpenSource Version
- **Dtksh** Korn Shell '93 mit CDE Builtins kann GUI als Shell skript bauen
- **Tksh** Korn Shell '93 mit TCL/TK Builtins kann GUI als Shell skript bauen

Shells auf Basis der BourneShell Syntax

- Bash 1989
- Ash (Almquist Shell) 1989 BSD Ersatz für BourneShell
- Dash (Debian Almquist Shell) 2004
- Pdksh (Public Domain ksh) 1988
- MirKorn Shell „mksh“ (pdksh Variante) 2003
- Zsh (als „zsh“ nicht Bourne Shell / POSIX kompatibel)
- Busybox von Bruce Perens Viele Builtins, Bourne Shell ähnlich auf Basis von ash, entstand aber erst nach ksh mit vielen Builtins

Unterschiede Bourne Shell, Schily, ksh

- Init Skripte, z.B. $HOME/.kshrc für interaktive Shells
- Pipe Aufbau Reihenfolge und Eltern Kind Verhältnis
- Variablen Zuweisungsreihenfolge
- Variablen Zuweisung (lebensdauer) vor Builtins
- Syntax Fehlerbehandlung bei Builtins
- History Editor
- Aliases
- Umask Parameter -S / chmod like Mask
- $PWD und andere Variablen

Unterschiede Bourne Shell, Schily, ksh

- ${.sh.*} Variablen
- Eingebaute Zeitmessungen
- pushd/popd/dirs (nur bosh)
- dosh Builtin (nur bosh)
- export/readonly Features
- Killpg (nur bosh)
- set -o longopt
- test Features
- ~ (Tilde Expansion)

- for *name* [in *word* ...] do *list* done
- case *word* in [*pattern* [| *pattern*]) *list* ;;] ... esac
- if *list* then *list* [elif *list* then *list*] ... [else *list*] fi
- while *list* do *list* done
- until *list* do *list* done
- (*list*)
- { *list*;}
- *name* () { *list*;}

- for (([*expr1*] ; [*expr2*] ; [*expr3*])) ;do *list* ;done (nicht POSIX)
- select *vname* [in *word* ...] ;do *list* ;done (nicht POSIX)
- ((*expression*)) (nicht POSIX)
- [[*expression*]] (nicht POSIX)
- time [*pipeline*] (nicht POSIX als Teil der ksh Syntax)
- ! *cmd*
- $(*cmd*)
- $((*expression*))

- **Bourne Shell: /etc/profile + ~/.profile bei login**
- **Bosh: /etc/profile + ~/.profile (login), /etc/sh.rc + /.shrc (i)**
- **Ksh93: /etc/profile + ~/.profile (login), /etc/ksh.kshrc + ~/.kshrc (i)**
- **Bei allen: kein .logout Skript, da es `trap cmd EXIT` gibt**

Pipe Aufbau, Eltern Kind Verhältnis

- Kommandointerpretation nicht im Standard vorgegeben
- Einfaches Beispiel: `cmd1 | cmd2`
- Bourne Shell: cmd1 ist Kind von cmd2, cmd2 ist Kind von sh
- Ksh93: beide Kommandos sind Kinder vom ksh
- Beispiel mit Builtin: `cmd1 | read var`
- Bourne Shell: read im Kind von sh, cmd1 ist Kind von read
- Ksh93: read läuft im ksh, cmd ist Kind von ksh
- Schily Bourne Shell: ähnlich wie ksh93
- Konsequenzen? Antwort aus dem Auditorium

Variablenzuweisungen

- `var1=val1 var2=val2 cmd` Kommando mit temporären (privaten) Environment Variablen
- Im statischen Fall kein Problem...
- Aber: `var1=val1 var2=$var1 cmd`
- Beim Bourne Shell ist var2 hier leer!
- Fix ist in ksh und bosh

Temporäre Variablen + Builtin Kommandos

- `var=val cmd` Erzeugt Kind, das var zuweist und cmd ruft
- `CDPATH=val cd` ist naheliegend aber problematisch
- „cd“ ist ein eingebautes Kommando und muß es sein
- Die Variablenzuweisung erfolgt daher im Shell selbst
- Beim Bourne Shell wirken alle Variablenzuweisungen vor Builtin- Kommandos auf den ganzen Shell nach
- Bei ksh und bosh wird dies bei den meisten Kommandos verhindert. Ausnahme: POSIX „special builtins“
- POSIX: bei „special builtins“ kommen vars in den Shell

Nomentklatur zu Builtins

- Einige Kommandos müssen im Shell implementiert sein damit sie funktionieren (z.B. cd)
- Der usprüngliche Bourne Shell hatte wenige Builtins
- Später kamen viele dazu
- Unübliches Verhalten ist nicht akzeptabel für Kommandos, denen man nicht ansieht daß sie im Shell implementiert sind (z.B. das ksh-builtin `uname`)
- POSIX führt „special builtins“ ein, anderes Verhalten OK
- Liste der „special builtins“ enthält nicht „cd“, „alias“, …
- Neu SUSv7 tc2: „intrinsic commands“ ohne anderes Verhalten

Syntax Fehlerbehandlung bei Builtins

- Änderungen am Shell um das Verhalten verständlich zu machen: Externe Kommandos <-> Builtins
- Bei externen Kommandos führt jeder Fehler maximal zu einem Exit Code != 0
- Klassische Methode im Bourne Shell: fatale Fehler in Builtins führen bei interaktivem Shell zu longjmp() vor Prompt-Ausgabe, sonst (wenn nicht interaktiv) zu exit
- Ksh: weitgehende Angleichung der Builtins an externe Kommandos – außer „special builtins“
- Bosh: ähnlich wie ksh
- POSIX: „command <special-builtin>“ kein spezial verh.

History

- Der ursprüngliche Bourne Shell hat keine History
- Ksh: Ursprünglich Builtin Erweiterungen zur Listenverarbeitung im Bourne Shell, daher auch keine History
- Ksh ab ca. 1983 History mit Hilfe des Kommandos `fc` editierbar durch externen vi.
- Später ab 1986: eingebauter History Editor in ksh
- Schily Bourne Shell (bosh) ab 2006 mit dem History Editor bsh (Berthold) Konzept/Prototyp 1982, erste Implementierung unter UNOS in bsh 1984. Emuliert: ved

Aliase

- Bourne Shell ursprünglich ohne Aliase
- Ksh führt um 1984 Aliase ähnlich zum csh ein
- Schily Bourne Shell: ab 2012 Aliase ähnlich dem Kommando Interpreter von UNOS (1980) mit persistenten Aliases und directory-spezifischen Aliases (POSIX++)
- Ksh: Parameterisierte Aliase wie beim csh sind nicht möglich
- Bosh: Parameterisierte Aliase mit Hilfe des Builtins „dosh“ das mini-Shell-Skripte emuliert

export/readonly

- Der Bourne Shell versteht `export VAR` bzw. `readonly VAR`
- Ksh, bosh und POSIX erlauben auch:
  - `export VAR=value ...`
  - `readonly VAR=value ...`
- sowie:
  - `export -p`
  - `readonly -p`
- Um eine Ausgabe zu bekommen, die wieder als Shell-Kommando nutzbar ist
  - `export -p > file`
  - `. file`

- Das builtin „umask“ im Bourne Shell versteht nur oktale Werte
  - Natchteil: Die „Maske“ ist das Inverse der Zugriffs-rechte und daher schwer verständlich
- Ksh führt `umask -S` ein, Ausgabe wie chmod, ähnlich zu `ls`
- Bosh folgt dem Beispiel, denn es wird auch durch POSIX gefordert
- Auch `umask u=rwx,g=rx,o=rx` ist als Ersatz zu `umask 022` möglich

- Der ursprüngliche Bourne Shell kennt nur wenige Shell Variablen: SHELL, PATH, HOME, IFS, MAIL, MAILCHECK, MAILPATH, PS1, PS2, SHACCT
- Ksh führt viele neue Variablen ein, einige davon werden in POSIX übernommen: ENV, LINENO, PPID, PS4, PWD
- Bosh übernimmt alle POSIX Erweiterungen
- Ksh und bosh: TIMEFORMAT für „time“ Formatierung
- Bosh: zusätzlich eingebautes Timing für alle Kommandos mit `set -o time`

- Bourne Shell: Verwendung von Variablen die mit einem Punkt anfangen führen zu einem longjmp() vor den interaktiven Prompt, bzw. zum Exit des ganzen Shells
- Ksh und bosh: ${.sh.*} Variablen für Sonderzwecke
- Bosh: Einfühfung von .sh.* um 32 Bit Exit Code zu liefern
- Alle anderen Shells: nur die unteren 8 Bit des Exit Codes der Kinder sind sichtbar obwohl es seit 1989 waitid() gibt

Zeitmessungen

- Klassiche UNIX Methode `/usr/bin/time kommando`
- Ksh implementiert „time“ als Teil der Syntax
- Bosh implementiert eingebautes Timing für alle Kommandos und „time“ als reserved Word wie ksh
- Ksh und bosh deaktivieren das eingebaute „time“, wenn es mit einer Option aufgerufen wird

- Bourne Shell (historisch): nur „cd“ ist verfügbar
- Bourne Shell 1989 von Svr4: CDPATH Variable wird unterstützt
- Ksh: ähnlich zum Bourne Shell von Svr4 aber „cd“ ist „logisch“ und Optionen -L / -P (default ist -L)
- Bosh: zusätzlich `pushd/popd` und `dirs`
  - Verwalten einen Directory „stack“
  - Optionen -L / -P (default ist -P)

Parameterisierte Alises

- Csh versteht: `alias lm 'ls -l \!* | more'`
  - \!* wird durch die aktuelle Argument Liste expandiert
- Bourne Shell kennt keine Aliase
- Ksh versteht nur einfache Aliase ohne Parameter
- Bosh kennt das Builtin „dosh“, das aus dem UNOS Kommmando Interpreter von 1980 abgeleitet wurde:
  - z.B: `alias lm='dosh '\''ls -l “$@“ | more'\'' lm'`
  - Ein „einzeiliges eingebautes Skript“ dosh ermöglicht die Parameterisierung

Komplexe Aliase verständlich editieren

- Wie im Vorigen Beispiel ersichtlich: die ksh/POSIX Syntax für aliase kann unlesbar werden
- Bosh kennt daher ein Verfahren zur Bearbeitung von Aliases im „RAW“ Modus:
  - `alias -R` Zeigt Aliase im RAW Modus an
  - Unser Beispiel im RAW Modus gezeigt:

```
#pb lm        dosh 'ls -l "$@" | more' lm
```

  - Wird plötzlich lesbar
- Zum Editieren: `set -o hashcmds` einschalten
- Dann `#lh lm` eingeben und Cursor hoch....

Lange Optionen

- Der Bourne Shell kennt nur wenige Optionen wie `-v`
- Ksh führt lange Optionen ein und POSIX übernimmt
- Auflisten der langen Optionen mit: `set -o / set +o`
- Statt `set -v` einschalten nun auch mit `set -o verbose`
- Bosh übernimmt dies, da es in POSIX ist
- Bosh kann zusätzlich lange Optionen mit getopts(1)
  - z.B. --lang als Alias zu -l mit optstring: „l(lang)“
  - aber auch --lang ohne entsprechende kurze Option
  - und -lang, wenn „optstring“ mit „()“ anfängt

Test Features

- Da „test“ seit Mitte der 1980er ein eingebautes Kommando ist, bestimmt der Shell die Features von „test“
- Bourne Shell hatte kaum neue Möglichkeiten eingebaut
- Ksh kennt vieles mehr (z.B. -e file) daher aufpassen wenn es portabel sein soll
- Bosh kennt alle ksh test Features

Abkürzungen für das Heimatverzeichnis

- Bourne Shell kennt nur $HOME
- Ksh führt zusätzlich ~ ein
  - ~/ → Eigene Homedirectory (POSIX)
  - ~joe/ → Homedirectory vom User „joe“ (POSIX)
  - ~+ → Aktuelles Verzeichnis ($PWD)
  - ~- → Voriges Verzeichnis ($OLDPWD)
- Bosh implementiert dies wie ksh

Ksh for- Schleife

- **Die ksh Erweiterung `for ((i=0; i < 10, i++))  ; do list; done` ist zwar im Bash verfügbar aber nicht POSIX**
- **Daher auch nicht in Bosh**

Select Kommando

- Das „select“ Kommando des ksh ist nicht in POSIX aber in vielen anderen ksh alike Shells
- Bourne Shell kennt es nicht
- Bosh kennt es noch nicht

Ausdrücke an beliebiger Stelle

- ((expression)) ist in ksh und bash aber nicht POSIX
- Bourne Shell kennt es nicht
- Bei anderen POSIX kompatiblen Shells gibt es:
  - $((expression))
- Achtung:
  - ((expression)) von ksh ist Teil der Syntax
  - $((expression)) ist eine Makro-Expansion
  - POSIX: $((expression)) oder $((cmd)), ksh erkennt es am Zusammenhang (Arithmetik vs. Sub-Shell)

- [[expression]] ist in ksh und bash aber nicht POSIX
- Bourne Shell kennt es nicht
- Bei anderen POSIX kompatiblen Shells gibt es:
  - „test“ und „[“
- Achtung:
  - [[expression]] von ksh ist Teil der Syntax
  - [expression] ist ein eingebautes Kommando

! cmd

- ! cmd Negiert den Exit Code (POSIX)
- Daher in vielen Shell verfügbar
- Nicht im Bourne Shell
- Aber in Bosh

Kommando Ersetzung

- Bourne Shell kennt: `` `cmd` `` um in der Kommandozeile die Ausgabe von „cmd“ einzubauen
- Das ist aber geschachtelt ähnlich komplex wie die ksh Alias-Definitionen
- Ksh führte daher $(cmd) ein und POSIX übernimmt
  - Das ist allerdings sehr schwer zu parsieren und benötigt einen rekursiven Parser
  - Korrekt implementiert nur in ksh93, bosh, bash-4, mksh

Arithmetische Ersetzung

- Nicht verfügbar im Bourne Shell
- Ksh: $((expression)) in der Kommandozeile wird ersetzt
- Noch nicht verfügbar in bosh, aber bald...
- Warnung Leerzeichen zwischen den Klammern, falls einer Kommandoersetzung in einem Subshell gemeint ist
- Ksh macht Plausibilitätsbetrachtungen aber das ist nicht portabel und kann nicht alles finden:
  - $((ls)) ist ein gültiger arithmetischer Ausdruck! (0)

Weitere ksh Erweiterungen

- Als der Bourne Shell 1979 entstand, gab es kein Pattern-Matching in libc
- Der Bourne Shell verwendet die eigene Funktion „g-match()“ die „*“, „?“ und „[...]“ unterstützt
  - Verwendet wird sie bei Dateinamen und bei case
- Ksh verwendet inzwischen teilweise Regular expressi-ons
- Ksh93 unterstützt „**“ für einen ganzen Directorybaum, wenn die Shell option -G (auch set -o globstar) gesetzt ist
- Ksh kennt viele Stringverarbeitungsfunktionen

Weitere ksh Erweiterungen

- Ksh kennt viele Stringverarbeitungsfunktionen:
  - ${parameter%word} Remove smallest suffix
  - ${parameter%%word} Remove largest suffix
  - ${parameter#word} Remove smallest prefix
  - ${parameter##word} Remove largest prefix
  - ${#parameter} Anzahl der Buchstaben in parameter
  - Sowie weitere non-POSIX Funktionen für substrings

Übung 1

- Erklären Sie warum
  - `$shell -c 'date | read VAR; echo $VAR'`
- Nur bei $shell == ksh, bosh und zsh das Datum ausgibt, nicht aber bei bash, dash, sh
- Entwickeln Sie ein Kommando daß das Datum bei allen Shells der Variable VAR zuweist

Antwort 1

- Bei ksh läuft das „read“ nicht in einem Subprozess
- Das portable Kommando ist:
  - sh -c 'VAR=`date`; echo $VAR'

Übung 2

- **Das Kommando `VAR1=bla VAR2=$VAR1 /usr/bin/env` listet eine leere Variable VAR2 wenn ein klassischer Bourne Shell verwendet wird. Warum passiert das und gibt es eine Lösung mit der die erwartete Ausgabe bei allen Shells nutzbar wird?**

Antwort 2

- **Portabel kann man nur VAR1 und VAR2 auf separaten Kommandozeilen setzen**
- **Dann werden aber VAR1 und VAR2 Bestandteil des Haupt-Shell-Prozesses**

Übung 3

- Welchen Wert hat die Variable FOO nach Ausführen des Kommandos

```
FOO=bar pwd
```

- In den diversen Shells?
- Warum unterscheiden sich die Ergebnisse?

Antwort 3

- Bei allen Shells, die Builtins POSIX-konform implementieren, hat `VAR=val pwd` keine Auswirkungen auf den eigentlichen Shell-Prozess
- Grund: Historisch hat sich das bei allen Builtins immer auf den Shell ausgewirkt, es wurde aber mit ksh88 geändert und von dort in POSIX übernommen

Übung 4

- Lesen Sie die Man Pages von Bourne Shell, bash, ksh, bosh und suchen Sie nach einer Möglichkeit bei Expansion der Variablen FOO folgenden Text zu erzeugen:
  - „gesetzt“ nur wenn $FOO nicht leer ist, sonst $FOO
  - „leer“ nur wenn $FOO leer ist, sonst $FOO
  - „fehlt“ nur wenn $FOO nicht existiert, sonst $FOO
- Hinweis: die Erklärung befindet sich im Abschnitt „Parameter substitution“
- Ist dies mit allen Shells möglich?
- Liefern alle Shells brauchbare Man Pages?

Antwort 4

- FOO=irgendwas
- ${FOO:+gesetzt}
- FOO=““
- ${FOO:-${FOO+leer}}
- unset FOO
- ${FOO-fehlt}
- Die Expansion ist mit allen Shells möglich wenn sie mindestens kompatibel zum Bourne Shell Stand 1981 sind

Übung 5

- Das Kommando `sh -c 'test -R && echo bla'` gibt „bla“ aus
- Erklären Sie warum
- Das Kommando `sh -c 'test 2 -GT 3; echo foo'` gibt mit einigen Shells „foo“ aus, aber nicht mit allen
- Erklären Sie warum

Antwort 5

- Weil „-R“ ein String ist, dessen Länge > 0 ist
- Weil -GT ein illegaler Operator ist gibt es beim historischen Bourne Shell ein exit() des ganzen Shells, bei POSIX Kompatibilität jedoch lediglich exit() mit Exit-Code != 0 für das `test` Kommando

- Das Kommando:

```
sh -c 'exit 1234567890' ; echo $?
```

- Gibt 210 aus, Erklären Sie warum nicht 1234567890
- Aus welchem Shell und unter welchen Umständen schaffen Sie das erwartete Ergebnis zu erzielen, wenn Sie dort das angegebene Kommando aufrufen?
- Mit:

```
ksh -c 'exit 1234567890' ; echo $?
```

klappt das nie, woran könnte das liegen?

Antwort 6

- 210 ist 12345678980 & 0xFF
- Wenn aus dem bosh mit „set -o fullexitcode“ gearbeitet wird und man auf einem modernen BS ist, klappt es
- Ksh maskiert schon den Parameter des eingebauten exit Kommandos mit 0xFF bevor das eigentliche exit() ausgeführt wird

Übung 7

- **Schreiben Sie ein kleines ksh Skript, das ein Menü mit 3 Auswahlmöglichkeiten ausgibt.**

```
select i in wer wo was; do
   echo Selektiert: $i  aus Nummer: $REPLY
   break
done
```

- **Schreiben Sie dieses ksh-Skript:**

```
farbe=braun-gelb-blau
if [[ $farbe == *gelb* ]]; then
        echo “mit gelb“
fi
```

- **So um, daß es auch mit dem Bourne Shell das gewünschte Ergebnis liefert**

```
farbe=braun-gelb-blau
case $farbe in
*gelb*)
        echo “mit gelb“
        ;;
esac
```

Arbeitsweise der POSIX Standardisierung

- Entwicklung der UNIX Varianten beobachten
- Tendenzen erkennen
- Vorhandene Implementierungen dokumentieren
- Features, die im Widerspruch zu vorher Vorhandenem sind, dürfen nicht standardisiert werden
- Bisherige Ausnahmen:
  - POSIX: getpgrp() / setpgrp() ist in Konflikt zu BSD
  - Drepper: fexec() und getline() sind in Konflikt zu UNOS und libschily

Probleme mit POSIX und dem Shell

- Normalerweise standardisiert POSIX das was am Markt ausreichend einheitlich verfügbar ist
- Es gibt aber viele Shell Implementierungen mit vielen zu einander inkompatiblen Erweiterungen
- Daher muß hier die Initiative von der Standardisierung ausgehen und Einfluß auf alle Shells nehmen
- Die normale Methode, Alles teilweise inkompatible als „unspecified“ zu bezeichnen wäre unbenutzbar
- Daher werden sinnvolle Shell Erweiterungen aufgegriffen und durch das POSIX Komitee neu formuliert

Planungen für POSIX Issue 8

- Unterstützung für die ksh $'...' Expansion
  - C-ähnliche String Escape Darstellung, z.B. $'\n'
  - Unicode Zeichen mit $'\uxxxx'
- Reservierter neuer POSIX Namensraum
  - Auflisten der inkompatiblen Builtins
  - POSIX Flags für set(1)
  - Prüfen oder aktivieren von erweiterten Features
- 32 Bit exit Codes mittels waitid()
  - Vermutlich mit Hilfe einer neuen Variablen $/
- siginfo_t Werte in „trap“ Kommandos verfügbar machen

Suchpfade für eingebaute Kommandos

- Alle Shells haben eigene Builtin-Kommandos
- Viele der Kommandos entsprechen normalen UNIX Kommandos mit leichten Abweichungen
- Normale Shell-Regel: ein Builtin hat immer Vorrang
- PATH ermöglicht bei separaten Binaries eine Auswahl
- Für die ksh93 Integration in OpenSolaris wurde PATH für Builtins gefordert und implementiert
- Bei ksh93 mit dem Kommando „builtin“:
  - PATH Zuordnung für jedes Builtin ändern
  - Einzelne Builtins deaktivieren oder reaktivieren

Suchpfade für eingebaute Kommandos

- Die meisten ksh93 Builtins sind dem Pfad `/usr/ast/bin` zugeordnet.
- `/usr/ast/bin` ist nicht im normalen PATH
- Einige wenige Kommandos sind `/usr/bin` zugeordnet
  - Allerdings nur auf Solaris
  - Ein solches Kommando auf Solaris ist `getconf` wegen POSIX `getconf PATH`
- Ein unbedarfter Nutzer sieht diese eingebauten Kommandos nicht

Danke!

URL: http://cdrtools.sf.net/Files/