

# GNU/Linux – Eine Einführung

---

Computer- und Mediensicherheit  
FH Hagenberg  
Wintersemester 2004/2005

# Agenda

---

- Information über das System
- SSH Suite
- Daemonen
- Shell
- Shellprogrammierung

# Disktools

---

# nützliche Hilfsprogramme

---

- ❑ `df` – freier Plattenspeicher
- ❑ `du` – belegter Plattenspeicher
- ❑ `locate` – findet Dateien mit Datenbank
- ❑ `updatedb` – aktualisiert DB für `locate`
- ❑ `find` – findet Dateien (langsam)
- ❑ `which` – findet den Pfad zu ausführbaren Dateien

# Prozesse

---

# Prozesse

---

- jeder Prozess hat eindeutig ID
  - bleibt ihm sein Leben lang erhalten
  - Prozess 1 ist init-Prozess
  
- jeder Prozess (außer init) hat einen "Vater" – jener Prozess, der ihn aufgerufen hat

# Informationen über Prozesse

---

- ps
  - Snapshot der gerade laufenden Prozesse
  - viele verwirrende Optionen
  - ps auxf meist ausreichend
- top
  - Prozessinformationen interaktiv und vollbild
  - liefert auch CPU- und Speichernutzung

# Prozesscontrol

---

- `kill -SIGNAL PID`
  - sendet Signal an Prozess
    - KILL – sofort beenden
    - TERM – ordentlich beenden
    - HUP – spezielle Bedeutung bei Serverp.
  
- `killall name`
  - sendet Signal an alle Prozesse mit Namen *name*

# Daemons

---

- sind Serverprozesse
- stand für Disk Access and Execution Monitor
- laufen immer im Hintergrund
- Ein- und Ausgabe nur über IPC
  - meist über Netzwerk
- Beispiel: Hauptbestandteil von Apache ist httpd, der HTTP Daemon

# Daemons (2)

---

- werden meist von Initskripts gestartet (beim Booten)
- laufen oft während der gesamten Laufzeit des Systems
- werden nicht von Shell-Jobcontrol erfasst
- direkte Kinder von `init`

# Kontrolle über die Unterwelt

---

- da Daemons nicht im Terminal laufen, geschieht Steuerung über Signale
- Signale werden mit `kill -SIGNAL pid` gesendet
- PID oft in `/var/run/daemonname.pid` abgelegt, damit Initskripte die Prozesse leicht finden können

# Signale

---

## □ TERM

- "beende dich" (nett)
- Prozess führt meist Aufräumarbeiten durch

## □ KILL

- "du wirst beendet" (nicht nett)
- Prozess wird gar nicht mehr ausgeführt, kann daher auch keine Aufräumarbeiten durchführen

# Signale

---

## □ HUP

- "User legt auf, beende dich"
- ursprünglich für Shell-Hintergrund-Jobs
- bei Daemons heute meist Bedeutung "lies deine Konfiguration neu"

## □ ALARM

- Bedeutung variiert nach Daemon
- wird nicht von allen erkannt

# Cron

---

# Cron

---

- ermöglicht die periodische Ausführung von Programmen
- läuft als Daemon im Hintergrund
- wird über Konfigurationsdateien (crontab) angepasst
  - eine systemweite Datei
  - je eine für jeden User (wenn nötig)
- mailt Ausgabe an Benutzer

# /etc/crontab

---

- für systemweite Programme

```
# Zeitpunkt    Benutzer    Programm
0 * * * *      root        /usr/sbin/updatedb
1 3 * * *      root        /usr/sbin/logrotate
15 4 * * 6      root        /usr/sbin/programm
```

- je nach verwendetem Cron-Programm möglicherweise HUP nötig

# crontab -e

---

- öffnet Benutzer-crontab in Texteditor
- installiert diese nach Beendigung richtig

```
# Zeitpunkt Programm
0 * * * * /home/cms04/bin/erinnerung
0 18 * * * echo "Zeit zum Essen"
```

# Statusinformationen

---

# System allgemein

---

- `uname`
  - zeigt Systemtyp, Kernel inkl. Version
  
- `uptime`
  - wie lange das System ohne Unterbrechung läuft
  - Auslastung des Systems

# Hauptspeicher

---

- free
  - belegter/freier Hauptspeicher
  - wieviel davon für Festplattencache genutzt wird

	total	used	free	shared	buffers	cached
Mem:	62080	61304	776	0	6132	43488
-/+ buffers/cache:		11684	50396			
Swap:		155224	16			

# Das /proc Pseudodateisystem

---

- Information über laufende Prozesse
  - `/proc/PID/cmdline` – Kommandozeile
  - `/proc/PID/exe` – Link auf Datei
- Informationen über den Kernel
  - `/proc/version` – Versionsnummer
- Informationen über das System
  - `/proc/cpuinfo`
  - `/proc/meminfo`
- Einstellungen für den Kernel

# Netzwerk

---

- netstat
  - zeigt aktive Verbindungen an
  - netstat -t – nur TCP
  - netstat -u – nur UDP
  - netstat -l – listening Ports (Server)
  
- arp
  - welche MAC-Adresse gehört zu welcher IP-Adresse?

# Der Bootvorgang im Detail

---

# Bootvorgang

---

1. Spannungsversorgung eingeschaltet
2. BIOS
3. Bootloader
4. Kernel
5. Init
6. Runscripts
7. Shell/X-Windows

# BIOS – Basic Input/Output System

---

- auf Mainboard-ROM gespeichert
- führt Selbsttest durch (POST)
- initialisiert Hardware
  - belegt Interrupts vor
  - fährt Festplatten hoch
  - ...
- lädt Bootsektor von 1. Festplatte

# Bootloader

---

- kleines Programm, da Bootsektor nur beschränkte Kapazität
- zuständig für das Laden des Kernels
- muss bereits Dateisystem verstehen
- verschiedenen Bootoptionen
  - werden dem Kernel mitgeteilt
  - auch verschiedene Kernel möglich

# Kernel

---

- Kern des Betriebssystems
- initialisiert Hardware wirklich
  - detektiert vorhandene Hardware autom. (sofern Treiber/Support einkompiliert)
  - kümmert sich nicht um BIOS-Settings
- ruft `/sbin/init` auf
  - benötigt dazu Angabe der `/`-Partition
  - anderer Pfad möglich mit Bootoption (z.B. `init=/bin/bash`)

# init

---

- Urvater aller Prozesse
- hat PID 1
- startet Programme nach /etc/inittab
  - Runscripts
  - getty – stellt I/O für virtuelle Terminals bereit, auf denen dann die Shell läuft
- kann direkte Kinder am Leben halten

# Runscripts

---

- mehrere Skripte, jeweils für eine bestimmte Aufgabe zuständig
  - Partitionen mounten
  - Dateisysteme überprüfen
  - zusätzliche Module laden
  - Dienstprogramme (Daemons) starten
  - Netzwerk starten

# Runlevel

---

- ❑ verschiedene Systemkonfigurationen
  - ❑ Wechsel mit `init` *runlevel*
- 1 Single User Mode
  - 2 kein Netzwerk
  - 3 normaler Runlevel (mit Netzwerk)
  - 5 graphischer Runlevel
- (6 Reboot)  
(0 Halt)

# Runscripts

---

- befinden sich in `/etc/init.d` oder `/etc/rc.d/init.d`
- welche gestartet werden, wird durch symbolische Links festgelegt
- `/etc/rc.d/rc<runlevel>.d`
- bei Gentoo `/etc/runlevels`

# Ende des Bootvorgangs

---

- init startet getty auf den angegebenen virtuellen Terminals
- getty startet login
- Voilà, wir können uns einloggen.

# Remotetools

---

# Remotetools

---

- bauen eine Verbindung zum PC über das Netzwerk auf
- heute meist verschlüsselte Protokolle
- ermöglichen eine Administration des Rechners ohne lokalen Zugang
  
- früher hauptsächlich telnet
  - dummerweise unverschlüsselt

# SSH – Secure Shell

---

- verbreitetste Remote Shell
- baut Verbindung verschlüsselt auf
  - in Version 2 sicher gegen Man-in-the-Middle Attacken
  
- `ssh benutzername@zielrechner [kommando]`

# SCP – Secure Copy

---

- Teil des SSH-Pakets
- macht sichere Dateitransfers möglich
- funktioniert in beide Richtungen
  
- `scp [user@quelle:]/pfad  
[user@ziel:]/pfad`

# SSH – Schlüsselbasierte Authentisierung

---

- ❑ basiert auf Public-Key-Verfahren
- ❑ es wird keine geheime Information übertragen
- ❑ `ssh-keygen` erzeugt Schlüsselpaar
- ❑ öffentlicher Schlüssel muss auf entfernten Rechnern bekannt sein

# SSH – X Forwarding

---

- X ist netzwerkfähig
  - Programme auf entferntem Rechner leiten Ausgabe auf lokalen Schirm
  - benötigt X-Server auf Empfängerseite
  - Verbindung aber ohne Authentisierung
- SSH ermöglicht sichere Weiterleitung der GUI-Ausgabe
  - Authentisierung
  - Verschlüsselung

# SSH – Port Forwarding

---

- kann beliebige TCP-Ports weiterleiten
- Anwendungsgebiete
  - Verschlüsselung für Klartextprotokolle
  - Umgehung von Portrestriktionen
  
- `ssh -L port:host:hostport`
- `ssh -R port:host:hostport`

# Dateitypen

---

# Devices

---

- dienen – wie gesagt – der Hardwareabstraktion
- Character Devices
  - I/O auf einzelne Bytes (Characters)
  - bei `ls -l` angezeigt durch `c`
- Block Devices
  - I/O auf Datenblöcke (z.B. Festplatten)
  - bei `ls -l` angezeigt durch `b`
- werden angelegt durch `mknod`

# Interprozesskommunikation

---

- Named Pipes
  - funktionieren wie Pipes bei Ausgabeumleitung, nur beständig
  - werden mit `mkfifo` angelegt
  - Ein-/Ausgabe auf Named Pipe umleiten
  - bei `ls -l` angezeigt durch `p`

# Interprozesskommunikation

---

- Unix Sockets
  - werden in Programmen wie Netzwerk angesprochen
  - werden von Programmen bei Bedarf angelegt (es gibt kein Kommando dafür)
  - bei `ls -l` angezeigt durch `s`

# Normale Dateitypen

---

- Datei
  - bei `ls -l` angezeigt durch `-`
- Verzeichnis
  - bei `ls -l` angezeigt durch `d`
- Symbolischer Link
  - Verweis auf anderen Dateipfad
  - wird mit `ln -s quelle ziel` erstellt
  - bei `ls -l` angezeigt durch `l`

# Die Shell

---

# Aufgaben der Shell

---

- Steuerung der Ein- und Ausgabe
- ermöglicht Interaktion mit System durch Kommandointerpretation
  - Ausführen von Programmen
  - Navigieren im Dateisystem
- austauschbar, da nicht fix in OS integriert

# Allgemeine Features der Shell

---

- besitzt eingebaute Kommandos
  - werden "built-ins" genannt
  - `cd`, `echo`, ...
- unterstützt konditionale Kommandoausführung
  - Schleifen (`for`, `while`, ...)
  - `if`-Konstrukt
- kennt und arbeitet mit Variablen

# Ablauf

---

- meldet sich mit Eingabe-Prompt
- interpretiert Eingabezeile nach Enter
  - teilt einzelne Kommandos auseinander
  - trennt Argumente eines Programms
  - ersetzt Variablen durch Werte
  - führt Kommandosubstitution durch
  - leitet Ein- und Ausgabe um
  - löst Sonderzeichen auf
  - führt Kommando(s) aus

# Bash – Bourne Again Shell

---

- `/bin/bash`
- Standardshell für GNU/Linux
- komfortabel im Umgang
  - bietet Tab-Komplettierung für Befehle
  - führt Kommando-Historie

# Andere Shells

---

- Große Auswahl verschiedener Shells
- unterscheiden sich in
  - Syntax der built-ins
  - Variablendeklaration
  - Komfortfeatures
- csh – C Shell
- tcsh – Technical C Shell
- ksh – Korn Shell

# Kommandohistorie

---

- `history` listet letzte Befehle nummeriert auf
- `!x` führt Befehl `#x` erneut aus
- Cursor-oben blättert in Historie
- Strg+R sucht rückwärts in Historie

# Variablen

---

- ❑ Zuweisung durch name=wert
- ❑ Auf-/Abruf durch \$name
- ❑ sind typenlos
  - können beliebige Werte annehmen
- ❑ gelten ab Zuweisung
- ❑ gelten jedoch nur in aktueller Shell
- ❑ set zeigt alle aktuell gesetzten an

# Umgebungsvariablen

---

- sind für alle Prozesse sichtbar
- werden von Programmen ausgewertet
  - oftmals wie selbstverständlich erwartet
  - auch von Shell selbst (z.B. \$HOME)
- NAME üblicherweise großgeschrieben
- "normale" können Umgebungsvariablen werden
  - export NAME
- env zeigt alle gesetzten an

# Umgebungsvariablen der Shell

---

- ❑ PATH – Verzeichnisse, in denen nach Programmen gesucht wird
- ❑ HOME – Heimverzeichnis
- ❑ PS1 – Promptzeichenfolge
- ❑ HOSTNAME – Rechnername
- ❑ TERM – um welche Art Terminal handelt es sich
- ❑ USER – Name des eingeloggten Users

# Metazeichen

---

- werden von Shell interpretiert

<b>?</b>	genau ein beliebiges Zeichen
<b>*</b>	beliebig viele (auch null) beliebige Zeichen
<b>[abc]</b>	genau eines der angegebenen Zeichen
<b>[a-f,x-z]</b>	ein Zeichen aus dem angegebenen Bereich
<b>[!abc]</b>	keines der angegebenen Zeichen
<b>[^abc]</b>	wie [!abc]
<b>~</b>	Home-Verzeichnis

# Quoting

---

- zur Aufhebung der Interpretation
- `\` hebt Interpretation des nächsten Zeichens auf
- `"..."` -- hebt Metazeichenersetzung auf, nicht aber Variablenuflösung
- `'...'` -- hebt jegliche Interpretation auf
- ``...`` -- Inhalt wird als Kommando ausgeführt und die Ausgabe substituiert

# Jobcontrol

---

# Jobcontrol

---

- eine Shell kann mehrere Jobs gleichzeitig verwalten
- Jobs sind einzelne Prozesse
- nur ein Job kann im "Vordergrund" laufen
  - Vordergrund = Zugriff auf das Terminal
- beliebig viele können im Hintergrund laufen

# Prozesse in den Grund schicken

---

- Strg+Z
  - stoppt Ausführung (und gibt Job-ID aus)
- `fg id`
  - holt Job *id* in den Vordergrund
- `bg id`
  - lässt Job *id* im Hintergrund laufen
- `kommando &`
  - startet *kommando* im Hintergrund (und gibt Job-ID aus)

# Shellprogrammierung

---

# Shellprogrammierung

---

- ähnlich mächtig wie "gewöhnliche" Programmiersprachen
- statt interaktiver Eingabe kann Kommandoabfolge auch in einer Datei vorliegen

```
#!/bin/bash  
echo "Hello, world!"
```

# Vor- und Nachteile von Skripten

---

- müssen nicht kompiliert werden
- werden direkt von Shell interpretiert
- sind langsamer als binäre Programme
- im Umgang mit Dateien etc. sehr praktisch, da Shell dafür konzipiert

# Erstellen von Shellskripten

---

- mit normalen Texteditor
- wichtig: erste Zeile "#!/bin/bash"
- Execute-Recht auf Datei setzen
  
- # -- Kommentar

# Arten der Ausführung

---

- mittels Interpreterangabe
  - bash skript
  - Datei muss nicht ausführbar sein
- wie normales Programm
  - ./skript
  - Datei muss ausführbar sein
  - Interpreter wird von Kernel gesucht (anhand der ersten Zeile)
- immer in eigener Shell ausgeführt

# Spezielle Variablen

---

- stehen in jedem Shellskript zur Verfügung
- \$0 – Name des Shellskripts
- \$1-9 – 1-9tes Argument
- \$# – Anzahl der Argumente
- \$@ – alle Argumente  
(zur Weitergabe)
- \$? – Rückgabewert des letzten Kommandos

# Beispiele für Argumentauswertung

---

```
#!/bin/bash
echo "Mein Name ist $0"
echo "Es wurden $# Argumente übergeben"
echo "1. Argument: $1"
echo "2. Argument: $2"
echo "3. Argument: $3"
echo "Alle zusammen: @$"
```

# Beliebig viele Argumente

---

- \$1 - \$9 reichen nicht immer aus
- `shift` verschiebt Argumentenliste um eine Stelle nach links
- linkstes Argument fällt weg
  - 10. Argument wird zu \$9
  - 9. Argument wird zu \$8
  - ...
  - 1. Argument fällt weg

# Bedingungen testen

---

- `test bedingung oder [ bedingung ]`
  - prüft *bedingung* und liefert 0 wenn wahr, 1 wenn falsch und 2 falls *bedingung* einen Fehler enthält
- Beispiel
  - `test -w /etc/passwd`
  - `[ -w /etc/passwd ]`
  - prüft ob `/etc/passwd` beschreibbar ist

# Dateien testen

---

- `test -f datei`
  - prüft ob existent und reguläre Datei
- `test -d datei`
  - prüft ob existent und Verzeichnis
- `test -r/w/x datei`
  - prüft die Berechtigungen von `datei`
- `test -s datei`
  - prüft ob `datei` nicht leer ist

# Vergleiche und Verknüpfungen

---

- `test ausdruck1 -a ausdruck2`
  - Logisches UND
- `test aus1 -o aus2`
  - Logisches ODER
- `test aus1 -eq/ge/gt/le/lt aus2`
  - gleich, größer gleich, größer, kleiner gleich, kleiner
- `test aus1 -ne aus2`
  - ungleich

# Kontrollstrukturen

---

- `if ausdruck; then`  
    `kommandos`  
`else`  
    `kommandos`  
`fi`
- `for varname in liste`  
`do`  
    `kommandos`  
`done`

# Kontrollstrukturen

---

- `while bedingung`  
  `do`  
    `kommandos`  
  `done`
- `until bedingung`  
  `do`  
    `kommandos`  
  `done`

# if Beispiel

---

```
❑ USERS=`who | wc -l`  
  if test $USERS -lt 5  
  then  
      echo "Weniger als 5 Benutzer"  
  else  
      echo "Mehr als 5 Benutzer"  
  fi
```

# for Beispiel

---

- ❑ 

```
for x in hans heinz karl luise
do
    echo " Name: $x"
done
```
- ❑ 

```
for datei in *
do
    cat $datei
done
```

# while Beispiel

---

```
□ while [ 1 -eq 1 ]  
do  
    echo "DoS for lame people"  
done
```

# Berechnungen durchführen

---

- mittels `expr`
- Punktrechnung vor Strichrechnung
- Beispiel
  - `x=`expr $A + $B + \( 2 \* 3 \)``

# sonstige Fragen und/oder Anmerkungen

---