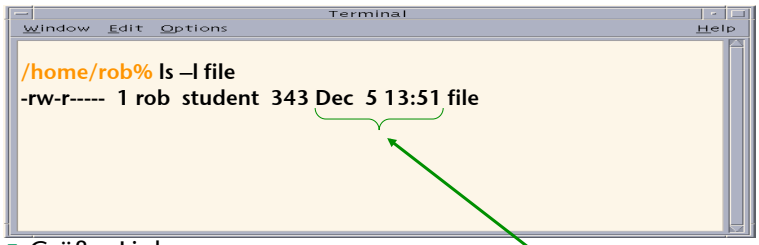
Exkurs: ACLs

- Oft feinere / flexiblere Regelung der Zugriffsrechte gewünscht
- ACLs = access control lists
- Features:
  - Individuelle Permissions pro User möglich
  - Selbst-definierte Gruppen
  - Permissions pro selbst-definierter Gruppe
  - ...
- Für rel. kleine Arbeitsgruppen ist das "normale" Unix-Permissions-Modell völlig ausreichend

Weitere File-Attribute

- Zeiten:
  - Modification (write): **ls -l**
  - Creation: **ls -lc**
  - Access (read): **ls -lu**



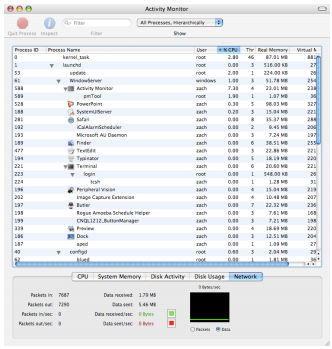
- Größe, Links, ...

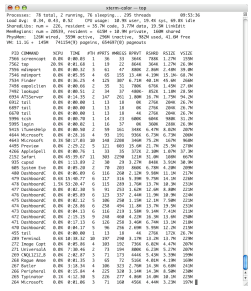
Prozesse

- Programm, das gerade läuft, schläft, oder hängt
- Jeder Prozeß führt ein *Environment* mit sich:
  - Prozeß-ID (PID), User-ID (UID), Group-ID (GID), u.a. IDs
  - current working directory (CWD, manchmal auch PWD)
  - Environment-Variablen (Paare von Strings)
  - ...
- Relative Pfade werden bzgl. des CWD's des Prozesses interpretiert

Spawning processes

- Ein Prozeß wurde *immer* von einem anderen erzeugt
- Heißt Vater-Prozeß (*parent process*)
- Vorgang heißt engl. *to spawn*
- *Child process* erbt das komplette Environment (außer seinen IDs u.ä.)





top in der Shell

Prozesse aus Sicht der Shell

- 3 Zustände eines Prozesses (aus Sicht der Shell)
  - Foreground: Default
    - Ausgabe (stdout) des Prozesses erscheint im Terminal-Fenster
    - Eingabe (stdin) des Prozesses kommt vom Keyboard
  - Background:
    - Ausgabe erscheint im Fenster
    - Eingabe nicht erlaubt
  - Gestoppt:
    - Prozeß schläft

Kommandos zur Prozesskontrolle

Befehl	Funktion
ps	Prozesse anzeigen
ps -edfjw	Alle Prozesse anzeigen
ps -auxw	dito für einige andere Unix-Varianten
kill pid	Prozeß mit PID **pid** abbrechen (wie Ctrl-C)
kill -9 pid	... wenn der Prozeß trotzdem nicht aufhören will ☺
command ... &	Prozeß im Hintergrund starten
jobs	Prozesse im Hintergrund anzeigen
Ctrl-C	Foreground-Prozeß abbrechen (interrupt)
Ctrl-Z	Foreground-Prozeß anhalten (stoppen)
fg	Zuletzt angehaltenen Prozeß im Foreground weiterlaufen lassen
bg	Angehaltenen Prozeß im Background weiterlaufen lassen
Ctrl-S	Ausgabe des Foreground-Prozesses anhalten (Pr. läuft weiter!)
Ctrl-Q	Ausgabe weiterlaufen lassen
top	tabellarische Ansicht aller Prozesse und deren CPU-Verbrauch

Bash vs. tcsh

- Annahme im folgenden: **tcsh** (csh) !
  - Sehr häufig ist auch bash, aber IMHO für interaktive CL-Nutzung nicht so komfortabel
- Wie stellt man fest, welche Shell man gerade hat?
  - Einfach '**set**' eingeben:

Resultat unter tcsh

```
% set
...
tcsh        6.14.00
...
version  tcsh 6.14.00 ...
```

Resultat unter bash

```
% set
...
BASH=/bin/bash
...
BASH_VERSINFO=([0]="3" ...
BASH_VERSION='3.2.17(1)-...
```

- Wer bash hat: das Kommando **tcsh** eingeben

Shells: Was passiert mit einem Kommando?

- Shell durchläuft folgenden Zyklus mit jedem eingegebenen Befehl:
  1. Kommando wird aus der Eingabezeile oder einem Script gelesen
  2. Aliases werden expandiert
  3. Variable Substitutions werden vorgenommen
  4. Wildcards (File-Patterns) werden expandiert
  5. Das Kommando wird ausgeführt:
     - Entweder von der Shell selbst (built-in commands)
     - Oder in einem neuen Prozeß

1. Aliases

- Ersatz für häufige / längliche Kommandos
- Beispiel:

```
% alias mo less
% alias uni "ssh –l zach –X hera.uni-bonn.de
% mo program.cpp
% uni
```

- **alias** zeigt alle Aliases an

Alias 'uni' wird ersetzt ...
Alias 'mo' wird ersetzt ...
Definiert Alias 'uni'
Definiert Alias 'mo'

2. Variablen

- Variable = Name + Wert, Wert = Zeichenkette
- 2 Arten:
  - Normale Shell-Variablen
  - Environment-Variablen,
    sind auch in Kind-Prozessen (child process) bekannt
- Setzen:

```
% setenv TMP "/tmp"
% set tmpdir = "/home/stud/zach/tmp"
```

- Verwenden:

```
% cp file ${tmpdir}
% echo ${TMP}
```

- Anhängen:

```
% setenv PATH ${PATH}:${HOME}/bin
% setenv PATH ${HOME}/bin:${PATH}
```


Wichtige Environment-Variablen

Variable	Bedeutung
DISPLAY	Display, auf dem neue Fenster geöffnet werden **(Bsp.: aurikel:0.0)**
HOME	Home-Verzeichnis
PRINTER	Default-Drucker
TMP	Verzeichnis für temporäre Files
PATH	Suchpfad für Kommandos
MANPATH	Suchpfad für Man-Pages
PWD	Aktuelles Verzeichnis, in dem man sich gerade befindet (CWD)

- **printenv** druckt alle Environment-Variablen aus
- **echo $VAR** druckt Wert dieser Environment-Variablen

4. File Patterns

- File-Name mit Wildcards: ***** **?** **[]**

Wildcard	Bedeutung
?	Genau ein beliebiges Zeichen
*	Beliebig viele beliebige Zeichen (auch 0)
{0,1,2}	Genau ein Zeichen aus der Menge {0,1,2}
[0-9]	Genau ein Zeichen aus der Menge {0,...,9}
[a-zA-Z0-9]	Genau ein Zeichen aus der Menge {a,...,z,A,...,Z,0,...,9}
[^0-9]	Genau ein Zeichen *nicht* aus der Menge {0,...,9}

- Beispiele:

```
% ls *.cpp *.h
```

```
% ls [0-9][0-9]*.ppt
```

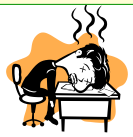
```
% ls *[^a-zA-Z0-9_.,-]*
```


Kleine Warnung zu **rm**

Task: Shoot Yourself in The Foot

The proliferation of modern programming languages (all of which seem to have stolen countless features from one another) sometimes makes it difficult to remember what language you're currently using. This handy reference is offered as a public service to help programmers who find themselves in such a dilemma.

```
% ls
foot.c foot.h foot.o toe.c toe.o
% rm * .o
rm: .o no such file or directory
% ls
%
```



Quotation

- Achtung: das Kommando sieht die Wildcards nie!
- Shell expandiert Wildcards
- Quotation verhindert, daß Wildcards (allg. Meta-Zeichen) von der Shell expandiert werden
- Arten:
  - \ verhindert Expansion des folgenden Zeichens

```
% echo \*.ppt \$PATH
```

  - "**...**" verhindert Expansion der Wildcards, erlaubt Variablen

```
% echo "*.ppt $PATH"
```

  - '**...**' verhindert jegliche Expansion (Wildcards, Variablen, ...)

```
% echo '*.ppt $PATH'
```


5. Wie die Shell ein Kommando ausführt

1. Built-in: Shell führt Kommando selbst aus
   - Beispiel: **echo**
2. Sonst: externes Programm
   - Beispiel: **ls dir**
3. **command** in **PATH** (Environment-Variable) suchen
4. Falls nicht gefunden, Fehlermeldung
5. Kind-Prozeß erzeugen
   - Erinnerung: erbt Environment des Vater-Prozesses (Shell)
6. Argumente (Zeichenketten) dem Prozeß bereitstellen
   - Argumente werden durch Space (i.a.) getrennt
7. Warten bis Kind-Prozeß beendet

Suchpfade

- Environment-Variable mit Liste von durch **:** getrennten Verzeichnissen
- Beispiel:

```
% echo $PATH
.:/usr/bin:/bin:/usr/local/bin:/home/II/zach
```

- Kommandos werden in **PATH** gesucht
  - File mit Namen des Kommandos im ersten Verzeichnis und executable? → ausführen
  - Sonst: nächstes Verzeichnis in PATH untersuchen ...
- Analog für Man-Pages und andere

File-"Viewer"

- **more, less**
  - Interaktiv, zeigen *jeden* File-Typ im Terminal-Fenster an
  - Unter Linux/Mac ist **more** = **less**
  - Suchen mit '/', weitersuchen mit 'n', rückwärts 'N', u.v.m.
  - Environment-Variable: **setenv PAGER 'less -I'**
- Nicht-interaktiv:
  - **cat** ("concatenate" = aneinanderhängen)
  - **head** , **tail** = ersten / letzten paar Zeilen anzeigen

Editoren

- Programmierer schreiben ASCII, insbesondere Software
  - Nur für reines ASCII (kein "Markup" irgendwelcher Art)
  - Für kleine Listen
  - Zum Editieren irgendwelcher Text-Files
    - Z.B.: VisualStudio-Project-Files, XML-Files, HTML-Files
    - Vor allem zum "remote" Editieren
- Heiliger Krieg, welches der beste ist
- Ein Programmier-Editor sollte ...
  - Effizientes UI haben (i.A. nicht intuitiv!)
    - Wenige Tasten / Mauskilometer für die häufigen Aktionen
  - Syntax Highlighting
  - Makros
  - Reguläre Ausdrücke zum Suchen und Ersetzen
  - Cross-platform sein

Einige Editoren zur Auswahl

- vim / gvim (Obermenge von vi, welcher immer installiert ist)
  - Effizientester Editor, steilste Lernkurve
  - Die Homepage von vim: www.vim.org
- emacs/xemacs (extrem umfangreich)
  - "Emacs wäre gar kein so schlechtes Betriebssystem, wenn es nur einen brauchbaren Editor hätte" ☺
- kate oder nedit (kein non-GUI-Mode → taugt nicht zum remote Editieren, sonst durchaus eine Alternative)
- ...

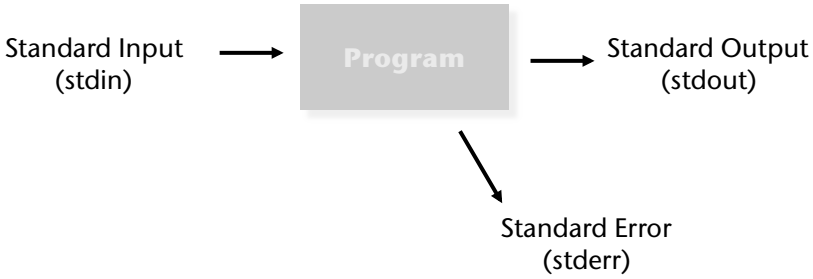
- Meine Empfehlung:
  - Für diejenigen, die sog. "power user" werden wollen: suchen Sie sich einen mächtigen, effizienten, und cross-plattform ASCII-Editor aus, und lernen Sie diesen beherschen (dauert eine Weile)
    - **`vim`**, **`emacs`** / **`xemacs`**, **`nedit`**, ...
  - Für alle anderen: nehmen Sie den im Menü angebotenen Editor
    - Unter Linux: **`kedit myfile.txt`** (oder vom "K"-Menü aus)
    - Falls Sie remote editieren möchten / müssen: **`mcedit`**
- Reference-Cards für VIM und Emacs auf der Homepage der Vorlesung

Die wichtigsten Befehle in vi / vim / gvim

- Besonderheit: Vim ist Mode-basiert !!
- Default- ("Home"-) Modus = Kommando-Modus
  - Aus jedem anderen Mode kommt man mit <Esc> dorthin zurück
- Tasten:
  - 'i' → Insert-Mode = Einfügen von Text an der aktuellen Cursor-Pos.
  - 'R' → Replace-Mode = Überschreiben
  - 'x' → aktuelles Zeichen löschen
  - :w → File speichern
  - :q → Vim verlassen
  - '/' → suchen
  - 'n' → nächstes Vorkommen suchen
  - '.' → letztes Kommando wiederholen

Standard-I/O

- Shell etabliert 3 Kanäle zu / vom Prozeß:



- Default-mäßig mit Terminal (Fenster & Keyboard) verbunden

Redirection

- Verbindet Standard-I/O mit Files
- Prozeß (Programm) bemerkt davon nichts!

Redirection	Bedeutung
> file	stdout wird nach file geschrieben
>> file	stdout wird an file angehängt
>& file	**stdout** und **stderr** umlenken
< file	Programm liest aus file, nicht von Keyboard
...	Shell bietet noch viele weitere Möglichkeiten

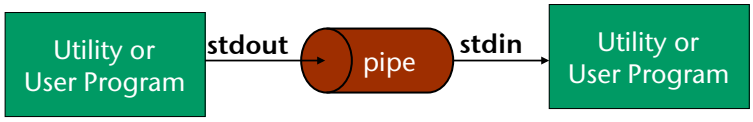
- Beispiel:

```
% ls –l > dir-listing
% sort dir-listing
```

```
% ps -edfyl > procs.log
```

Pipelines

- Effiziente Möglichkeit, komplexere Kommandos aus einfacheren zusammenzubauen!
- Selber Mechanismus wie bei Redirection, um Prozesse miteinander zu verbinden:



- Syntax:

```
% command | command | command | ...
```


Beispiele

- Listings weiterverarbeiten:

```
% ls  | sort | more
% ls -1 *.cpp | wc –l
% ls -1 *.cpp | sort > sorted-dir
```

```
% ls -1 | tr -s "_"
```

Tip:
Bauen Sie eine Pipeline immer eine Stufe nach der anderen auf!!

Unix-Philosophie

- "Small is beautiful"
- Make each tool do *one* thing only
- Make it do it well
- Read from stdin, write to stdout (if sensible)
- Use ASCII-Files

Filter

- Sind hauptsächlich dazu gedacht, um damit Pipelines aufzubauen
- Generelle Verwendung: Filter lesen von stdin, schreiben auf stdout
- Arbeiten oft zeilenweise

Utility	Funktion
cut	Felder oder Zeichenspalten ausschneiden
fmt	Auf 72 Zeichen umformatieren
sort	Zeilenweise sortieren (auch nach Teil-Key)
uniq	Duplikate entfernen
wc	Zeichen, Wörter, und Zeilen zählen
tr	Zeichen ersetzen
grep	Zeichenketten in der Eingabe suchen (s.u.)
rev	Reihenfolge der Zeichen umkehren (zeilenweise)

Beispiel

- Alle Filegrößen in einen File ausgeben:

```
% ls -l | tr -s " " | cut -d' ' -f5 > sizes.txt
```

- Nur die letzten paar Zeilen eines langen Directory-Listings:

```
% ls -l | tail -10
```