



# Tiny Shell

---

- Rudimentäre Shell
- Kompakt - 9 Seiten Code
- Eingabe per Datei oder Interaktiv
- Token Erkennung
- Interne/externe Befehle
- Prozesse, Signale



# Input Funktionen

---

Die *tsh* bekommt ihre Eingabe zeilenweise von:

1. Einer Datei
2. Einem Terminal (interactive shell, wobei „stdin“ ebenfalls als Datei angesehen wird)
3. Einem String (z.B. bei Umgebungsvariablen)

Die Zeilen werden von `handle_command()` geparkt und in Wörter (*tokens*) aufgeteilt.



# Token Erkennung

---

- Einteilung in *tokens* erfolgt gemäß Regeln die auf das jeweils nächste Zeichen (*char*) der Eingabe angewandt werden
- Ein *token* wird durch *delimiter* begrenzt



# Token Erkennung – Regeln (1)

---

Die Regeln beziehen sich auf das jeweils aktuelle Zeichen:

- Quoting, Kommentare

- Anführungszeichen (`` ` `), Escape (\) sowie Kommentarzeichen (#) beeinflussen wie die nächsten Zeichen eingelesen werden (spezielle Zeichen bzw. ganze Zeilen werden ggf ignoriert)

- Variablen

- Dollarzeichen (\$) starten eine Parameter Expandierung mittels `expand_var()`:
  - `${varname}` liefert `varname` gemäß `getenv`
  - `$$` liefert PID, `$?` liefert den *last exit status*



# Token Erkennung – Regeln (2)

---

- Globbing
  - Wildcards *\**, *?* können Datei- und Pfadnamen expandieren.
- Delimiter
  - Die Zeichen *blank*, *tab* trennen einzelne *tokens*
  - Die Zeichen *newline*, *semicolon*, *ampersand* beenden *tokens*, diese werden als Befehl ausgeführt
  - Die *tokens* werden vor dem Ausführen (`run_command( )`) expandiert bzw. geglobbt



# Befehlssuche und -erkennung: interne Ausführung

---

- `run_command( )` prüft zunächst ob der angegebene Befehl zu den eingebauten (internen) Befehlen gehört:
  - Falls ja, so wird er sofort ausgeführt
  - andernfalls wird `fork( )` benutzt um den Befehl extern auszuführen, hierbei bestimmt *ampersand (&)* ob der Befehl im *foreground* oder *background* läuft



# Befehlssuche und –erkennung: externe Ausführung

---

- *foreground* Ausführung:
  - Der aktuelle Prozess wird blockiert (*suspended*) bis sein Kindprozess (*child*) terminiert
- *background* Ausführung:
  - Der aktuelle Prozess wird blockiert und sofort wieder reaktiviert ohne auf die Terminierung seines Kindprozesses zu warten



# Interne Befehle

---

- `cd`
  - benutzt `chdir`; ohne Parameter wechselt `cd` zu `$HOME`, falls `$HOME` undefiniert - zum *root*-Directory, ansonsten wird `chdir` mit angegebenem Parameter ausgeführt
- `echo`
  - benutzt `printf` und `putchar` für *newline*
- `exec`
  - benutzt `execvp` (für externe Programme, und ihre Argumente)
- `exit`
  - benutzt `exit` und *last\_status* wenn möglich
- Variablenzuweisung
  - benutzt `putenv`, z.B. `HOME=/home/user`



# Prozesse in UNIX

---

- Erzeugung von Kindprozessen mittels `fork()`
- Warten auf Kindprozesse
- Abfrage des Status eines Kindprozesses



# Prozesse – userspace Adressraum

---

- Textsegment
  - Schreibgeschützter ausführbarer Programmcode, von mehreren Prozessen nutzbar
- Datensegment
  - Beinhaltet user-Daten, Heap, usw
  - Hat schreib(un)geschützte Bereiche
- Stacksegment
  - Lokale Variablen einer aufgerufenen Funktion
- Shared-Memory-segement(e)
  - von mehreren Prozessen nutzbar



# Prozesse und ihre Kinder (1)

---

- Prozesserzeugung mit *fork()*:
  - Es wird eine „Kopie“ des aktuellen Prozesses erstellt, insbesondere werden die Daten-, das Stacksegment und der Heap kopiert (der Kernel benutzt *copy-on-write*, dh. eine *page* wird nur dann kopiert falls wirklich darauf geschrieben wird)
  - Beide benutzen das gleiche Textsegment, allerdings mit unterschiedlichen Befehlszeigern
  - Oft werden mit *exec* die Datensegmente des Kindes ersetzt



# Prozesse und ihre Kinder (2)

---

- Prozesserzeugung mit *fork()*:
  - Der Elternprozess bekommt bei `pid=fork()` die PID seines Kindes, das Kind bekommt als return-Wert „0“, um seine eigene PID herauszufinden muss das Kind `getpid()` aufrufen:

```
if((pid = fork()) < 0) {
    /* Fehler beim Aufruf */
} else if(pid == 0) {
    /* Kindprozess */
} else {
    /* Elternprozess */
}
```



# Warten auf Kinderprozesse (1)

---

- Terminierung:
  - Terminiert ein Elternprozess vor seinem Kind, so wird das „verwaiste“ Kind dem Init-Prozess unterstellt
  - Terminiert das Kind, ohne dass sein Elternprozess darauf wartet, wird es zum „zombie“



# Warten auf Kinderprozesse (2)

---

- **Warte-Funktionen:**
  - `wait()` wartet bis eins der Kinder terminiert und gibt bei Erfolg dessen PID wieder
  - `waitpid()` wartet auf die Terminierung eines bestimmten Kindes mit angegebener PID (falls `PID=-1` so warte auf beliebiges Kind), die Option `WNOHANG` ermöglicht es den Elternprozess nicht zu blockieren obwohl das Kind noch nicht terminiert ist



# Warten auf Kinderprozesse (3)

---

- `wait()` sowie `waitpid()` können den Status der beendeten Kindprozesse anzeigen, z.B.
  - `WIFEXITED` ist `True` falls das Kind normal terminierte
  - `WIFSIGNALED` ist `True` falls es von einem nicht abgefangenen Signal unterbrochen wurde
  - `WIFSTOPPED` ist `True` falls das Kind angehalten wurde
  - `WSTOPSIG` liefert die Signalnummer, die das Kind stoppte



# Weitere Informationen

---

- Single Unix Specification

<http://www.opengroup.org/onlinepubs/009695399/>

- Linux man pages

- diet libc - a libc optimized for small size

<http://www.fefe.de/dietlibc/>