Betriebssysteme I

Vorlesungs- und Praktikumsnotizen

Mitschrift von Falk-Jonatan Strube

Vorlesung und Praktikum von
Prof. Dr.-Ing. Baumgartl

11. Februar 2016

Inhaltsverzeichnis

Mitschrift von
Falk-Jonatan Strube

Hinweise

Bei der Klausur sind keine Hilfsmittel zugelassen.
Die Praktikumsaufgaben, die mit * gekennzeichnet sind, sind nicht Voraussetzung für die Prüfung.

1 Einführung

Unix ist eine Betriebssystem-Familie.

Lizensierung: open source VS closed source
GNU: „vererbendes“ open source

1.1 Klassifikation BS

- Nutzer: single/multi
- Tasking: single/multi
- Kommunikation: autonom (batch)/interaktiv
- Verteilung: lokal/verteilt
- Architektur und Zweck: Server/eingebettet/Echtzeit/PC/...

1.2 Modellierung BS

- monolithisch: jede Routine/Funktion/... darf auf alles zugreifen (kein Information Hiding)
- geschichtet: Kommunikation nur zwischen benachbarten Schichten.

  Bsp.:

App
...
Treiber
...
Hardware

- client-server: Server arbeitet Wünsche von Clienten ab ($\Rightarrow$ Speicherverwaltung innerhalb BS)

Zweck eines BS:

- Bereitstellen von Diensten + Abstraktionen (Prozess, Datei, Treiber, ...)
- Ressourcenverwaltung
- Ablaufkoordination
- Schutz + Sicherheit

2 Shell

2.1 Wichtige Shell-Befehle

Mitschrift von
Falk-Jonatan Strube

Befehl	Wirkung
ls	Dateien in Ordner auflisten (-l: detailliert)
cd	in Ordner wechseln
cp	Datei kopieren
scp	Datei im Netzwerk kopieren
mv	Datei bewegen
rm dir	Datei löschen (-r: rekursiv durch Ordner)
mkdir	Ordner anlegen
rmdir	leeren Ordner löschen
chmod	Nutzerrechte ändern (u/g/o +/- r/w/x)
chown	Eigentümer ändern
less	Dateiausgabe (seitenweise)
cat file	Dateiausgabe im Terminal
w	Anzeige eingeloggter Nutzer und deren Prozesse
grep	Durchsuche Datei nach Zeichenkette, gebe passende Zeilen aus
	(-o: gebe passende Worte aus (auch bei doppeltem Vorkommen in Zeile)
wc -l-w-c	Word Count (-l Zeilen, -w Worte, -c Bytes)
cut file	Zeilen beschneiden
	-d ' ': Worttrenner (hier: Leerzeichen)
	-f 1: Welche Worte anzeigen: -f 1 $\rightarrow$ 1. Wort; -f -3 $\rightarrow$ ab 3. Wort
uniq	nur einzigartige Zeilen ausgeben
sort	Zeilen alph. Sortieren (-n: numerisch; -r: in umgekehrter Reihenfolge
find -name *test*	Suche Datei (-name: nach Name)
man	Handbuch über alle Befehle
ps	aktuelle Information über Prozess
	(-A: alle Prozesse; r: Prozesse die bereit sind; X: Inhalt stackpointer, . . . ;
	f: Verwandschaftsverhältnisse [besser: pstree]; -l: langes Format)
kill	Signal senden
bg	Schickt Programm in den Hintergrund
top	Anzeige der rechenintensivsten Programme
mount	Datenträger einbinden
du	Anzeige Platzbedarf von Datei(en) (-s: Summe aller)
df	Anzeige Belegung Dateisystem
ln	Verweis erstellen
shred	sicheres Löschen
stat	Anzeige von Dateiattributen
fdisk	Partitionierung
mkfs	Dateisystem anlegen
fsck	Prüfung+Reperatur Dateisystem

Mitschrift von
Falk-Jonatan Strube

hdparm	Detailinformationen Massenspeicher
pgrep	Suche nach Prozess anhand von regexp.
nice	Setze Prozess-Priorität

2.2 Weiterleitung

ls > ls.txt	Ausgabe von ls in Datei ls.txt (neu erstellt)
ls » ls.txt	Ausgabe von ls in Datei ls.txt (ergänzend, hängt an)
foo < ls.txt	Datei ls in Eingabe von Prozess foo
	(wenn ls.txt mehrere Zeilen enthält, sind das mehrere Eingaben für bspw. read())
foo | bar	Ausgabe von foo in Eingabe von bar (siehe Pipe)

Ausgabemöglichkeiten:

stdin	1	
stdout	1	
stderr	2	find xyz 2>/dev/null (Fehler ins nichts umleiten)

2.3 Schleifen/Anweisungen

```
if [ ... ]
then
  ...
fi

while [ ... ]
do
  ...
done

for x in ...  (bspw. $1/*)
do
 ...
done

for (( x=0; $x < $y; x++ ))
do
 ...
done
```

Bedingungen:

```
-lt, -le, -eq, -ne, -ge, -gt  (für numerische Vergleiche)
$#          Anzahl Parameter
-f ...      Prüft, ob ... Datei ist
"$1" != ""  Prüft, ob Variable/Parameter leer ist
$?          Rückgabewert letzter Funktion/Anweisung
$*          gibt alle Parameter aus
$RANDOM     Zufallszahl

return      max zwischen 0...255

x = '...' ⇔ x = $(...)    $(...) weißt stdout einer Fkt einer Variablen zu
let x = $a + $b ⇔ (( x = $a + $b ))
```


Mitschrift von
Falk-Jonatan Strube

2.4 Regex

.	beliebiges Zeichen
*	beliebig viele des vorhergehenden Zeichens
\?	0 oder 1 des vorhergehenden Zeichens
\+	1 oder mehr des vorhergehenden Zeichens
[xyz]	eines der in eckigen Klammern stehendes Zeichen
[^xyz]	alle außer eines der Klammer-zeichen
^	Zeilenanfang
$	Zeilenende
[0-9]	Bereich
[[:alpha:]]	$\Leftrightarrow$ [a-z A-Z]
[[:digit:]]	[[0-9]]
[[:alnum:]]	$\Leftrightarrow$ alpha+digit
[[:upper:]]	Großbuchstaben
[[:lower:]]	Kleinbuchstaben
[[:space:]]	Leerzeichen
\.	Escape-Char \$\Rightarrow$. wird normal ausgegeben
\< \>	Wortanfang / -ende
\{ m,n \}	mind m, höchstens des vorhergehenden Zeichen
\{ m \}	m mal
\(xyz \)	xyz wird als Zeichen behandelt (zusammenfassen von Ausdrücken)
x | y	x oder y

2.5 Brace Expansion

{a,b,c}xyz $\rightarrow$ Permutation von a,b,c mit xyz: axyz, bxyz, cxyz
{a..c}{x..z} = {a,b,c}{x,y,z} $\rightarrow$ alle a,b,c mit x,y,z: ax, ay, az, bx, by, bz, cx, cy, cz
Auch geschachtelt: { {a..z} $\underset{oder}{,}$ {A..Z} } $\rightarrow$ alle Buchstaben: Kleinbuchstaben oder Großbuchstaben

Sorten von Dateien

1.) gewöhnliche Datei

2.) Verzeichnis

3.) Spezialdateien
   - $\rightarrow$ Links
   - $\rightarrow$ Geräte
   - $\rightarrow$ ...

Mitschrift von
Falk-Jonatan Strube

3 Dateisystem

Abstraktion zur Strukturierung von Informationen
→ Abhängig von: Geschwindigkeit des Mediums, Informationsmenge, Fehlertoleranz
Aufbau einer Festplatte:

- hat mehrere Platten in einem Zylinder
- Platte hat Spuren (verschiedene Plattenradien) und Sektoren („Kuchenstücke“ der Platte)

Datei (Grundeigenschaften)

- Eigentümer
- Zugriffsrechte
- Sichtbarkeit
- Dateiname
- Zeitstempel
- Größe
- Dateipositions-Zeiger
- Typ (⇒ Magic Word an Beginn der Datei)

Dateifktn C		Systemruf Unix	... über Verzeichnis
fopen()		open()	opendir()
fclose()		close()	closedir()
fread()	beim Verzeichnis: sequentiell über alle	read()	readdir()
fwrite()	schreiben	write()	
fprintf()	schreiben (formatiert)		
feof()	Test auf Dateiende		
ferror()	Test auf Fehler		
fseek()	Versetzen Pos.-Zeiger	lseek()	
ftell()	Abfragen Pos.-Zeiger		
flock()	Sperren der Datei		
	Verweis anlegen	link()	symlink() (Softlink)
	umbenennen	rename()	
	Datei in Hauptspeicher einbleden	mmap()	
	Verz. anlegen		mkdir()
	Löschen		rmdir()
	Suche nach Einträgen		scandir()
	Zurücksetzen Eintragszeiger		rewinddir()

Mitschrift von
Falk-Jonatan Strube

Rechteverwaltung verschieden Möglichkeiten:
Zugriffsmatrix

	DateiA	DateiB
NutzerA	Rechte	Rechte
NutzerB	Rechte	

Daraus die Projektionen:

- Access Control List (ACL):
  - DateiA: NutzerA(Rechte), NutzerB(Rechte)
  - DateiB: NutzerA(Rechte)
- Capability List:
  - NutzerA: DateiA(Rechte), DateiB(Rechte)
  - NutzerB: DateiA(Rechte)

Darstellung in Unix

Owner	Group	Others	All Users
u	g	o	u+g+o⇒a
rwx	rwx	rwx	

Beispiel in Bash:

```
chmod u+rwx g+r-wx o-rwx FILE
```

Darstellung ls -l:

$-$ $rwxrwxrwx$ 1 $nutzerA$ $gruppeC$ 1234 $2016-...07:42$ $FILE$

Typ | Rechte (u g o) | Anz. Verz. | owner | group | size | last changed | file name

4 Ressource

Schnittstelle	Protokoll
statischer Aspekt d. Kommunikation	dynamischer Aspekt d. Kommunikation
synchron/asynchron	
Hardware/Software	

Def. Ressource Wird von Aktivitäten genutzt.

- existiert in allen Schichten des Systems (Datei, Code, Festplatte (Hardware), . . .)
- hat Zustand (Dateiinhalt, Registerinhalt, . . .)

4.1 Ressource entziehen

Voraussetzung Entziehbarkeit:

- Ressourcen-Zustand ist vollständig auslesbar
- Ressourcen-Zustand kann beliebig manipuliert werden

Mitschrift von
Falk-Jonatan Strube

Bsp.:

entziehbar	nicht entziehbar
CPU	CPU-Cache
Hauptspeicherblock	Drucker
	Netzwerkkarte

Vorgang ist für Aktivität transparent

1.) Aktivität anhalten

2.) Ressourcenzustand sichern

3.) (entzogene Ressource anderweitig verwenden)

4.) Zustand restaurieren

5.) Aktivität fortsetzen

Exlusivität

- maximal von einer Aktivität geilchzeitig verwendbar (wird von BS durchgesetzt durch Zuteilung)

4.2 Klassifikation

entziehbar (Prozessor, Speicher)	nicht entziehbar (verbrauchbare Betriebsmittel, ext. Hardware)
gleichzeitig nutzbar (Code, Datei, Speicher)	exlusiv nutzbar (Prozessor, Drucker, Signal)
wiederverwendbar (CPU, Datei, Speicher)	verbrauchbar (Signal, Nachricht, Interrupt)
physisch (Hardware: CPU, RAM, . . .)	logisch/virtuell (Datei, Signal, CPU)

4.3 Ressourcen-Transformation

(in Ebenen)

Applikation	↑ Byte der Datei
Dateisystem	↑ Logischer Block
Treiber	↑ physischer Sektor
Hardware	

4.4 Kernel

Kernel-Modus (Gegensatz: User-Modus)
Beschreibt einen previligierten Zugriff auf Hardware (bei entsprechender Unterstützung der CPU)
Nutzer hat nur eingeschränkte Rechte, Kernel exklusive:

- erstellen eines neuen Prozesses

Mitschrift von
Falk-Jonatan Strube

- Treiber laden/entfernen
- $\Rightarrow$ Diensterbringung des BS

Damit der Nutzer trotzdem auf diese Systemfunktionen zugreifen kann gibt es die:

4.4.1 Systemrufe

Anweisungen, mit denen ein User Kernel-Dienste nutze kann (siehe Tabelle im Kapitel 3 (Dateisysteme))

5 Aktivität

5.1 Prozesseigenschaften

- hat Lebenszyklus: Erzeugung $\rightarrow$ Abarbeitung $\rightarrow$ Beendigung
- benötigt Ressourcen beim Start (Speicher, PID, Code)
- benötigt dynamisch Ressourcen beim Abarbeiten
- hat eigenen virtuellen Prozessor und Speicher (Adressraum)
- besitzt immer einen Vaterprozess (und ggf. einen Kindprozess)
  - nur Prozess kann Prozess erzeugen (bspw. durch Doppelklick auf Icon im GUI, über die Shell usw.)

Mögliche Prozesszustände

- aktiv (arbeitet)
- bereit (kann arbeiten)
- wartend (wartet auf Ressource zum Arbeiten)

Übergänge:

- aktiv $\rightarrow$ bereit (Prozess wird verdrängt (bspw. durch Scheduling))
- aktiv $\rightarrow$ wartend (Prozess benötigt Ressource)
- wartend $\rightarrow$ bereit (Prozess hat alle benötigten Ressourcen)
- bereit $\rightarrow$ aktiv (Prozess beginnt (wieder) zu arbeiten)

Jeder Prozess starte zuerst im bereit-Zustand.

Adressraum Prozess

Umgebung, Argumente
Stack ↓ ... ↑ Heap
uninitialisierte Daten
initialisierte Daten
Text (Code)

5.2 Prozess beenden

- durch sich selbst:
  - Verlassen der main (durch letzte main-Zeile oder return)
  - exit() an beliebiger Stelle
- Fremdbeedigung:
  - Signal (⇒ kill)
  - fataler Fehler (bspw. 0-Division)

5.3 Fork

```c
pid_t fork(void);
```

- klont aufrufenden Prozess
  - → nur unterschiedliche PID, PPID
  - → geforkter Prozess setzt an gleicher Stelle fort, wie aufrufender Prozess (also nach dem fork()-Aufruf)
  - → Prozess-Abarbeitungsreihenfolge nicht determiniert (Vater kann vor Sohn weiter abgearbeitet werden oder anders herum)
  - → return-value von fork():
    Vater:
    - -1 (Fehler)
    - PID vom Sohn (Erfolg)

    Sohn:
    - 0
  - → Variablen sind jeweils Privat (übernehmen jeweils die Werte wie vor dem fork, Änderungen im Vater/Sohn finden aber nur dort statt)

```c
ret = fork();
if ( ret == -1 ) {
  perror("fork"); exit(EXIT_FAILURE);
}
```

```
5  if ( ret == 0 ) {
6    // Sohn
7  } else {
8    // Vater
9    puts(ret);
10 }
```

5.4 Exec

```
1 int execl("Programmpfad", "Argumente...");
```

- ersetzt aktuellen Programmcode durch entsprechende Binärdatei aus Pfad
  → springt diesen Code sofort an (kehrt nur im Fehlerfall zurück, bspw. bei falscher Pfadangabe oder mangelden Zugriffsrechten)
  → Rückkehr in Ausgangsprozess danach unmöglich!
- erzeugt keinen neuen Prozess (muss bspw. im Sohn nach fork() ausgeführt werden, wenn der Prozess (Vater) erhalten werden soll)!

5.5 System

```
1 int system("Kommando");
```

- kombiniert fork+exec und führt Kommando aus
  → kehrt erst zurück, wenn Kommando abgearbeitet ist

5.6 Wait

```
1 pid_t wait (int *status);
```

- bringt aufrufenden Prozess in Wartezustand, falls ein Kind vorhanden ist. Wartezustand wird beendet, sobald ein (irgendein) Kind-Prozess beendet wird.
- Status kann u.a. erwarteter Rückkehrcode des Sohns sein.
- return: -1 bei Fehler, Sohn-PID sonst

6 Kommunikation

IPC (inter process communication)
bspw. über: Datei, Pipe, Signal, shared memory, ...
Unterscheidung bzgl. Anforderung:

- Teilnehmerzahl: 1:1, 1:m, n:1, n:m
- Richtung: uni-/bidirektional
- lokale/entfernte Kommunikation
- direkte/indirekte Kommunikation

6.1 Übertragungsarten

Synchron

- wartet, bis Sende-/Empfangvorgang abgeschlossen ist

Asynchron

- Senden/Empfangen ohne „Bestätigung“
  - → unabhängig davon, ob etwas (oder wie viel bereits) gesendet oder empfangen wurde wird weiter gearbeitet
  - → braucht Zwischenspeicher
  - → gut, da kein Deadlock wegen Warten entstehen kann

Verbindungsorientiert

- Verbindungsabbau
- Übertragung
- Verbindungsabbau

bspw. Telefon, Pipe

Verbindungslos

- nur Übertragung

bpsw. Telegramm, Signal

Verbindungsarten	$1-1$	unicast
	$1-x_i$	multicast
	$1-all$	broadcast (bpsw. in Subnetz)

6.2 Übertragung über Datei

Schlecht, da:

- doppelter Zugriff auf langsame HDD (nur gut, wenn Dateisystem im RAM)
- überlappender Zugriff muss mit lockf() geregelt wender $\Rightarrow$ fehleranfällig

6.3 Übertragung über Pipe

```
1 int pipe ( int filedes[2] );
```

2 : 2 Diskriptoren: 0 $\widehat{=}$ read, 1 $\widehat{=}$ write

Mitschrift von
Falk-Jonatan Strube

Vorgehen

1.) pipe(x[2])

2.) fork()

3.) Prozesse schließen jeweils einen Diskriptor:
   - Sohn: close(x[1]); (write geschlossen, also lesend)
   - Vater: close(x[0]); (read geschlossen, also schreibend)

4.) Datenübertragung:
   - Sohn: read(x[0], intoVar, length); (Sohn wartet hier, bis Vater etwas schickt)
   - Vater: write(x[1], text, lenght);

5.) beide Prozesse rufen close() (des noch nicht geschlossenen Diskriptors) auf ⇒ Pipe geschlossen

```
int pipe[2]
ret = fork();
if ( ret == 0 ) {
  close(pipe[1]);
  x = read(pipe[0], var, 80);
  close(pipe[0]);
} else {
  close(pipe[0]);
  write(pipe[1], text, 80);
  close(pipe[1]);
}
```

Eine Pipe ist:

- unidirektional (sonst 2 Pipes nötig)
- keine persistente Ressource (verschwindet nach close() aller Teilnehmer)
- nur zwischen verwandten Prozessen möglich!

Pipe in Shell:

```
du  sort -n -r  less
```

⇒stdout von du in stdin von sort

6.3.1 popen

```
FILE *popen (Kommando, Pipe-Typ [w/r]);
```

- legt Pipe an, forkt Prozess, der dann Kommando ausführt
  - je nach Typ: w: stdin vom Kommando zeigt auf Pipe ⇒ Kommando liest r: stdout vom Kommando zeigt auf Pipe ⇒ Kommando schreibt
- pclose() ⇒ wieder schließen

6.4 Signale

- Informationsübertragung (in der Form von Anweisungen) ohne nötige Vererbung

Ablauf

1.) Sendeprozess generiert Signal

2.) Signal zustellen (durch System)

3.) Aufruf Signalhandler (falls vorhanden)

4.) Aufruf default-Aktion (falls kein handler implementiert)

Signal bpsw. SIGINT mit default-Aktion: Abbruch (auch durch Strg+C zu erreichen)

6.4.1 Signalhandler

Bash

```
trap handleIt SIGINT
```

C

```
sig_t ret;

void handler (int c){
  // bspw.: default signal reaktivieren:
  ret = signal(SIGINT, SIG_DFL);
}

main (){
  ret = signal (SIGINT, (sig_t) &handler);
  if (ret == SIG_ERR){
    perror("signal"); exit(EXIT_FAILURE);
  }
}
```

6.4.2 kill

Signal senden

```
int kill (pid_t pid, int sig);
```

- sendet Signal (falls Nutzer Berechtigung dazu hat)
- falls pid=-1 $\Rightarrow$ senden an alle

Zustellung

1.) nicht abfangbare Signale ausführen

2.) abfangbare Signale an handler geben, sonst default-Aktion ausführen

Anweisung bzgl. Signalen:

- pause() - wartet auf Signal
- alarm(x) - wartet x, dann wird Signal SIGALRM zugestellt

Einordnung

- unzuverlässig
- keine Nutzdaten-Übertragung
- keine Priorisierung
- keine Speicherung (Warteschlange)

6.5 Shared Memory

- gemeinsam genutzer Speicher
- Segmente bleiben persistent

6.6 Message Passing

Prinzip

1.) Sender trägt Nachricht in Puffer ein
2.) Sender sendet: send()
3.) System transportiert Nachricht
4.) Empfänger empfängt: recieve()
und schreibt Daten in Puffer

- bei nicht gemeinsamen Speicher nützlich
- synchron und asynchron möglich

7 Prozessorzuteilung

Zielgrößen Prozess(or)-Verteilung:

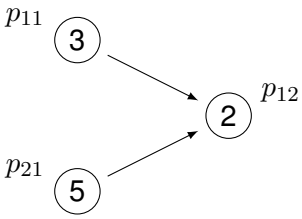
- Durschnittliche Reaktionszeit Prozesse
- Durschnittliche Verweilzeit Prozesse
- maximale CPU-Auslastung
- maximale Anzahl an Datenströmen
- garantierte maximale Reaktionszeit vorhanden?
- Fairness? n Prozesse $\rightarrow \frac{1}{n}$ Prozessorzeit
- Ausschluss des Verhungerns

Mitschrift von
Falk-Jonatan Strube

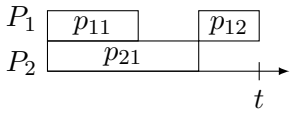
7.1 Off-Line Scheduling

- Ermittlung Abarbeitungsreihenfolge und Prozessorzuteilung vor Laufzeit
  - → – inflexibel
    + hohe Auslastung möglich
  - → alle Randbedingungen müssen bekannt sein

Präzendenzgraph zum ermitteln (topologisch sortierter gerichteter Graph)



⇒ **Gantt-Diagramm**:



7.2 On-Line Scheduling

- Auswahl der Abarbeitungs-Reihenfolge zu Laufzeit
  - → + flexibel
    – keine Zeit für lange Auswahlverfahren ⇒ Kompromiss bei Optimierung

7.2.1 Zeitgesteuertes Scheduling

- Abläufe periodisch
- keine Interrupts

Time Division Multiple Access (TDMA)

- innerhalb jeder Periode wird Periodenlänge zwischen n Teilnehmern aufgeteilt. Jeder hat $\frac{1}{n}$ Zeit (auch wenn nicht direkt genutzt wird).
  ⇒ keine Kollision möglich

7.2.2 Ereignisgesteuertes Scheduling

- reagiert auf Einflüsse von außen
- keine Garantie von Ausführungszeiten möglich, da Interrupts unvorhersehbar
  - → interaktive System (GUI, . . .)

Interrupt passiert asynchron zum Programmablauf. Bspw. durch IO-Geräte, BS, Programm, . . .

Mitschrift von
Falk-Jonatan Strube

7.2.3 Schedulingzeitpunkt

- präemtptives Multiptasking :
  Unterbrechung jederzeit durch BS möglich (Prozess blockiert, bereit, fertig) [idR. an bestimmten „Preemption Points“)
- kooperatives Multitasking :
  Freiwilliges Unterbrechen durch Prozess, bpsw. durch Systemaufrufe
- oder wenn Aktivität komplett (run-to-completion)

7.2.4 Priorisierung

Prozesse besitzen unterschiedliche Wichtigkeiten (unfair)

- statische Priorität
  - – Priorität konstant
  - \+ einfacher Scheduler
  - \+ einfache Analyse
  - – nicht flexibel

  Bsp.: Fixed External Priorities (FEP)
  jeder Prozess erhält vor LZ einen Prioritäts-Paramter zugeordnet
  $\rightarrow$ zur LZ wird immer der höchste gewählt
- dynamische Priorität
  - – periodische Neuberechnung der Abarbeitungsreinfolge $\Rightarrow$ Aufwand
  - \+ flexibel
  - – schwer zu analysieren

  Bsp.: Implizite Prioritäten
  Priorität basiert auf bspw.:
  - – Joblänge
  - – verbleibende Abarbeitungszeit
  - – Zeit der letzten Aktualisierung
  - – Deadline

Uni-/Multiprozessor zusätzliche Probleme

- wo wird Prozess abgearbeitet (am besten unbeschäftigter Prozessor oder Prozessor auf dem Prozess zuvor lief)

7.2.5 Round-Robin

Kenngrößen:
t_q Prozesszeit (Quantum)
t_{cs} Umschaltzeit
Je größer die Prozesszeit ist, desto träger kann das System werden (lange Reaktionszeit). Bei kurzer Prozesszeit ist die Umschaltzeit im vergleich relativ groß $\Rightarrow$ ineffizient (aber schnelle Reaktionszeit)

Mitschrift von
Falk-Jonatan Strube

7.2.6 FIFO/FCFS

fair, leicht zu analysieren (→ Warteschlange)

- Prozesse werden in Reihenfolge des Eintreffens vollständig abgearbeitet

7.2.7 Shortest Job Next (SJN)

- kleine Prozesse werden immer vor großen abgearbeitet
  ⇒ kann zum verhungern führen, unfair!
- Ausführzeit muss bekannt sein!

7.3 Unix

- Unterscheidet fürs Scheduling 2 Arten von Prozessen:
  1.) interaktive (→ nutzen Zeit nicht aus (wartend))
  2.) rechnende
- bevorzugt interaktive

Linux

- Prioritäten + Zeitscheiben (Prioritäten von 19 (niedrigste) über 0 (normal) bis -20 (höchste))

7.4 Priority Boost

Priorität wird durch langes Warten erhöht und durch Abarbeiten schrittweise verringert.

Mitschrift von
Falk-Jonatan Strube