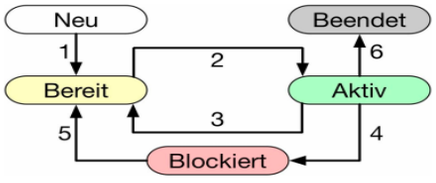
Analogie Kuchenbacken:
Programm = Rezept aus Rezeptbuch
Prozess = Backen eines Mandelkuchens Prozessor (CPU) = Köchi
Multitasking = zwei Kuchen „gleichzeitig“ backen.

Kapitel 2: CPU-Verwaltung und Prozesse

Konzept der Prozesse

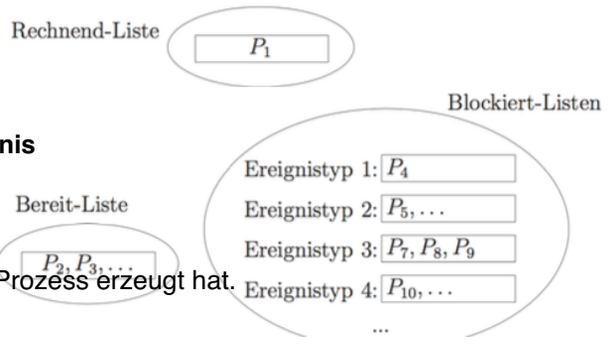
Programme und Prozesse

- Ein **Programm** ist eine statische Beschreibung für einen Algorithmus. (Assembler)
- Ein **Prozess** ist eine dynamische Ausführung eines Programms auf einem Computer.
- Ein leichtgewichtiger Prozess wird als **Thread** bezeichnet.
  Ein **Thread** bezeichnet die Programmausführung oder den Ausführungsstrang innerhalb eines Prozesses.
- Mit **Multithreading** wird die gleichzeitige Ausführung von mehreren Threads innerhalb eines Prozesses oder eines Anwendungsprogramms bezeichnet.
- Als **Multitasking** wird die Fähigkeit des Betriebssystems bezeichnet, mehrere Aufgaben nebenläufig auszuführen.
- Mit **Multiprocessing** wird in Mehrkernprozessoren die Möglichkeit bezeichnet, Aufgaben echtgleichzeitig auszuführen.
- Als **Multiprogramming** wird der Vorläufer des Multitasking bezeichnet.



Zustandsübergänge

1. Der **Prozess wurde gerade erzeugt**
   Er wird in die Liste der bereiten Prozesse eingetragen.
2. Der Prozess bekommt vom Betriebssystem die CPU zugeteilt und darf rechnen.
3. **Prozess bekommt die CPU wieder entzogen** (pre emptive), bzw. gibt sie freiwillig (non pre emptive) ab.
   „Normaler“ Entzugsgrund: Ablauf der Zeitscheibe.
4. **Prozess muss auf ein Ereignis warten**, z. B. Tastendruck oder andere Ein-/Ausgaben.
   Er kann nicht rechnen, selbst wenn er die CPU hätte.
5. **Die Ein-/Ausgabe ist abgeschlossen**. Der Prozess könnte weiter rechnen.
6. Das Programmende ist erreicht.



Prozesslisten

- Betriebssystem führt Listen für alle existierenden Prozesse:
  Aktiv-Liste, Bereit-Liste, Blockiert-Liste + eventuell welches Ereignis

Prozesskontrollblock (PCB)

- **Prozessnummer (PID):** Systemweit eindeutige Nummer
- **Prozessnummer des Elternprozesses (PPID):** Prozess, der diesen Prozess erzeugt hat.
- **Benutzer- und Gruppenidentität (UID, GID):**
  In welchem Namen bzw. für welchen Benutzer läuft der Prozess.
- **Schedulingzustand und -priorität**
- **CPU-Register**: Wo steht der Programmzähler (PC), usw.
- **Offene Dateien**: Lister der von diesem Prozess geöffneten Dateien inklusive aktueller Position in der Datei.
- **Aktuelles Arbeitsverzeichnis**: Das Verzeichnis in dem relativen Dateinamen dieses Prozesses beginnen. Wird mit dem Kommando pwd angezeigt.
- **Ressourcenbeschränkungen** bezüglich Hauptspeicher, CPU-Zeit, Datei-Locks, usw.
- **Terminal**, von dem aus dem Prozesse gestartet wurde.

Der **Scheduler**:

wählt aus der Liste der bereiten Prozesse einen aus, der als nächster die CPU bekommt.
Mögliche Auswahlkriterien: **Fairness Priorität Wartezeit**

- (**Round Robin Algorithmus**: Warteschlange von Prozesse in Bereit-Liste)

Der **Dispatcher**:

schaltet die CPU zwischen den einzelnen Prozessen hin und her.

1. Anhalten des laufenden Prozesses.
2. Sichern des Prozesszustands im Prozesskontrollblock (engl. process control block (PCB)).
3. Für den nächsten Prozess den Prozesszustand anhand des PCB rekonstruieren.
4. Den nächsten Prozess weiterlaufen lassen.

Round Robin Scheduling

Wie entscheidet das System, welcher Prozess als nächstes die CPU bekommen soll?
Idee dieses Verfahrens:

1. Neben dem laufenden Prozess gibt es eine Warteschlange mit bereiten Prozessen.
2. Spätestens nach Ablauf der Zeitscheibe wird der laufende Prozess unterbrochen.
3. Der bisher laufende Prozess kommt an das Ende der Warteschlange.
4. Der erste Prozess in der Warteschlange bekommt die CPU.

Vorteile:
Fair, jeder kommt dran, keiner verhungert.
Warteschlange einfach zu implementieren.

Nachteile:
Keine Bevorzugung „wichtiger“ Prozesse möglich.

Prozesse in UNIX

Existierende Prozesse mit ps ansehen

- Das Kommando **ps** zeigt Informationen über aktive Prozesse zum Aufrufzeitpunkt.
- Zu jedem Prozess wird eine Zeile ausgegeben, in der u. a. diese Informationen stehen:
  - Die **Prozessnummer (PID)** Process ID
  - Die **Prozessnummer des Elternprozesses (PPID)** Parent PID
  - Das **Terminal**, aus dem dieser Prozess gestartet wurde (TTY) TeleTYpe
  - Das **Kommando**, mit dem dieser Process gestartet wurde (COMMAND)
- Wichtige Optionen (Rest siehe man `ps`) sind:
- `-e` (every): Zeige alle Prozesse, nicht nur die eigenen und die aus demselben Terminal gestarteten.
- `-f` (full): ausführliche Informationen ausgeben
- `-l` (long): vollständige Informationen ausgeben

Beispiel: `ps -ef`

```
u5g-4b2-u1@enodia:~$ ps -f
UID          PID  PPID  C STIME TTY          TIME CMD
u5g-4b2+ 14343 14342  0 11:08 pts/29   00:00:00 -bash
u5g-4b2+ 14823 14343  0 11:10 pts/29   00:00:00 ps -f
```

Bedeutung der Spalten:

- **UID**: Benutzer, der den Prozess gestartet hat.
- **PID**: Prozess-ID
- **PPID**: PID des Elternprozesses
- **C:** Prozessorauslastung in Prozent
- **STIME**: Startzeit zu der der Prozess gestartet wurde.
- **TTY**: Das Terminal von dem der Prozess gestartet wurde.
- **TIME**: Akkumulierte CPU-Zeit bisher
- **CMD**: Das Kommando das zu diesem Prozess geführt hat.

Prozesse erzeugen in UNIX

- In UNIX gibt es nur einen Mechanismus, um einen neuen Prozess zu erzeugen: (**fork**)
  - Der Systemaufruf fork erzeugt eine identische Kopie des Prozesses, aus dem fork aufgerufen wurde.
- **Original-Prozess** wird Eltern-Prozess genannt.
- Kopie-Prozess wird **Kind**-Prozess genannt.
- Das Kommando **pstree** zeigt die Eltern-Kind-Beziehung aller Prozesse.
- Der „**erste Prozess**“ ist die einzige Ausnahme, da er nicht durch `fork` erzeugt wurde.
  Er heißt **init**-Prozess oder **systemd** (**launchd** in macOS) und hat die **PID Nr. 1**. Von ihm stammen dann alle anderen Prozesse ab.

Prozesse mit der Shell erzeugen und Hintergrundprozesse

- Werden externe Kommandos in einer Shell eingegeben, dann wird von der Shell ein neuer Prozess mit dem Kommando gestartet.
- Erst wenn das externe Kommando beendet ist, kann in der Shell das nächste Kommando eingeben werden.
- Externe Kommandos können im Hintergrund ausgeführt werden. Dazu wird das **&** am Ende der Zeile angehangen.
- Beispiel: `xosview&`
  - Ein neuer Prozess wird von der bash gestartet, in dem das Programm `xosview` ausgeführt wird.
  - Die PID dieses Prozesses wird angezeigt.
- Langlaufende Prozesse werden gerne in den Hintergrund gelegt, damit im Terminal weitergearbeitet werden kann.
- Bei Hintergrundprozessen sollte die Standardausgabe möglicherweise in eine Datei umgelenkt werden.

Prozesskommunikation durch Signale

- Prozesse kommunizieren untereinander, indem sie sich gegenseitig Signale senden.
- Auch Benutzer können in der Shell Signale an den laufenden Prozess senden.
  Wichtige Signale sind:

Signal	Tastenkürzel	Bedeutung
kill	CTRL + c	Abbruch des Prozesses
suspend	CTRL + z	Hält Prozess an

- Einen angehalten Prozess kann wieder zum Laufen gebracht werden.
  - Im Vordergrund weiterlaufen lassen: **fg**
  - Im Hintergrund weiterlaufen lassen: **bg**
- Mit dem Kommando **kill** können Prozessen Signale gesendet werden.
- Beispiel: `kill -9 1234`

Prozessverwaltung

- Mit dem Kommando **at** kann ein Prozess auch zu einem späteren Zeitpunkt gestartet werden.
  Beispiel: `at 2130 -f meinSkript`
- Mit `at -l` lässt sich sehen, welche Prozesse in der „Warteschlange“ stehen.
  Jeder Prozess hat eine Nummer in dieser Schlange.
- Mit `at -d 12` wird der Prozess mit der Nr. 12 aus der Schlange gelöscht und doch nicht ausgeführt.
- Das Kommando `jobs` zeigt an, welche Prozesse gerade unter dem aktuellen angemeldeten Benutzer laufen.
- Mit dem Kommando `killall` lassen sich mehrere Prozesse mit einem bestimmten Namen beenden;
  kill brauchte die Prozessnummer (PID).

Der Prozessbaum

- Mit dem Kommando `pstree` lässt sich eine Übersicht über alle laufenden Prozesse und die Eltern-Kind-Beziehungen erhalten.

```
init-+-acpid
     |-apache2---5*[apache2]
     |-atd
     |-softflowd
     |-sshd-+-bash---decompresspcaps---processDirector---ipfix2rdbms.sh-+-ipfix2rdbms.sh
     |      |                                                           |-logger
     |      |                                                           |-mysql
     |      |                                                           |-super_mediator---+
     |      |                                                           `-uniq
     |      +-bash-+-pstree
     |             `-xosview
     `-udevd---2*[udevd]
```

Systemauslastung mit top

```
top - 14:58:07 up 11 days, 48 min,  2 users,  load average: 0.52, 0.14, 0.14
Tasks: 205 total,   2 running, 203 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.6 us,  0.6 sy,  0.0 ni, 98.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:   66087544 total, 63223804 used,   2863740 free,   234136 buffers
KiB Swap: 134156284 total,        0 used, 134156284 free, 61751300 cached
PID   USER      PR  NI  VIRT  RES  SHR S  %CPU %MEM    TIME+  COMMAND
13812 worker    20   0  8740  632  468 S    64  0.0   0:32.98 gzip
13813 root      20   0     0    0    0 D    38  0.0   0:00.98 flush-254:0
13811 worker    20   0 18536 1124  932 D    19  0.0   0:05.25 tar
    1 root      20   0 10656  788  648 S     0  0.0   0:10.92 init
    2 root      20   0     0    0    0 S     0  0.0   0:00.02 kthreadd
    3 root      20   0     0    0    0 S     0  0.0   9:52.65 ksoftirqd/0
    6 root      rt   0     0    0    0 S     0  0.0   0:00.08 migration/0
    7 root      rt   0     0    0    0 S     0  0.0   0:02.66 watchdog/0
```

Einfluss der Benutzer*innen auf die Prozessverwaltung

- Das **Scheduling** des Betriebssystems können Benutzer*innen nicht beeinflussen.
- Benutzer*innen können mit dem Kommando `nice` ihren eigenen Prozessen eine niedrigere Priorität geben, so dass sie beim Scheduling nicht so schnell an der Reihe sind.
- Auf einem überlasteten System sollten keine weiteren Prozesse gestartet werden.
  Ggf. sollte geprüft werden, ob einzelne Prozesse nicht besser abgebrochen (mit dem kill-Kommando) werden sollten, damit andere Prozesse schneller fertig werden.

Benutzung der Shell

Datenströme in der Shell

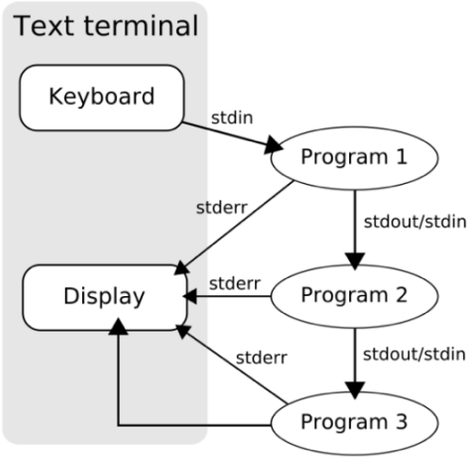
Es gibt **drei Datenströme** in einer Shell oder für ein Kommando:

1. **Standardeingabe** oder kurz **stdin** oder **Kanal 0:**
   Aus diesem Datenstrom kommen die Eingaben, häufig ist die Tastatur die Quelle dieses Datenstroms.
2. **Standardausgabe** oder kurz **stdout** oder **Kanal 1:**
   In diesem Datenstrom kommen die „normalen“ Ausgaben des Skriptes, bzw. Kommandos, häufig ist das das Terminal (der Bildschirm).
3. **Standardfehlerausgabe** oder **stderr** oder **Kanal 2:**
   In diesen Datenstrom kommen die Fehlermeldungen des Skripts oder des Kommandos, häufig ist das auch das Terminal.

Jeder dieser Datenströme lässt sich umleiten, bspw. in eine Datei: (Die Datei wird dabei überschrieben.)

Strom	Zeichen	Beispiel
stdout	`>` oder `1>`	`ls -l > liste`
stdin	`<`	`wc < liste`
sterr	`2>`	`find / -name xy 2> /dev/null`

- Mit dem Zeichen >> (oder 2>>) wird die Datei nicht überschrieben, sondern die Ausgaben werden hinten an die **Datei angehängt**.
  Beispiel: `ls -l >> liste`
- Stdout und stderr lassen sich zusammen in eine Datei umlenken. Dazu gibt es die Zeichenfolge 2>&1
  Beispiel: `find / -name xy > liste 2>&1`
- In einer Shell lässt sich die **Standardausgabe eines Kommandos direkt mit der Standardeingabe** des folgenden Kommandos verbinden.
  - Dazu benutzt wird das Zeichen | verwendet
  - Beispiel: `ls -l | wc`
  - Dieser Mechanismus wird **pipe** genannt.
  - Hierbei können auch mehr als zwei Kommandos verbunden werden.



Zusammenfassung

- Ein auf einem Computer aktuell ausgeführtes Programm nennt wird Prozess genannt.
- Prozesse können unterschiedlich weit vorangekommen sein und in verschiedenen Zuständen sein. Die drei wichtigsten Zustände sind:
  - 1 Rechnend
  - 2 Bereit
  - 3 Blockiert
- Ein **Scheduler** wählt aus den bereiten Prozessen denjenigen aus, der als nächster Rechnend wird.
- Der **Dispatcher** schaltet die CPU dann zwischen dem aktuell rechnenden Prozess und dem von Scheduler ausgewählten Prozess um.
- Unter UNIX wird mit dem **fork-Mechanismus** ein neuer Prozess erzeugt.