Betriebssysteme (BS)

03. Prozesse

https://sys.cs.tu-dortmund.de/DE/Teaching/SS2022/BS/

20.04.2022

Peter Ulbrich

peter.ulbrich@tu-dortmund.de
bs-problems@ls12.cs.tu-dortmund.de

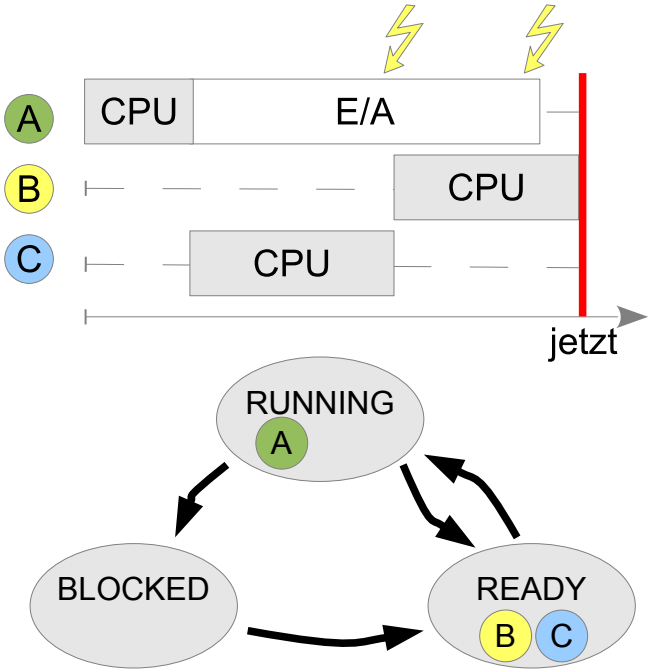
Basierend auf *Betriebssysteme* von Olaf Spinczyk, Universität Osnabrück

technische universität
dortmund

arbeitsgruppe
systemsoftware

Wiederholung

- **Prozesse** sind Programme in Ausführung
  - Dynamisch, nicht statisch
  - Abwechselnde Folge von *CPU-Stößen* und *E/A-Stößen*
- benötigen **Betriebsmittel** des Rechners
  - CPU, Speicher, E/A-Geräte
- haben einen **Zustand**
  - READY, RUNNING, BLOCKED
- werden konzeptionell als unabhängige, nebenläufige Kontrollflüsse betrachtet
- unterliegen der Kontrolle des Betriebssystems
  - Betriebsmittel-Zuteilung
  - Betriebsmittel-Entzug



Inhalt

- Das UNIX-Prozessmodell
  - Shells und E/A
  - UNIX-Philosophie
  - Prozesserzeugung
  - Prozesszustände
- Leichtgewichtige Prozessmodelle
  - „Gewicht“ von Prozessen
  - Leichtgewichtige Prozesse
  - Federgewichtige Prozesse
- Systeme mit leichtgewichtigen Prozessen
  - Windows
  - Linux

Tanenbaum
2.1: Prozesse
10.1-10.3: UNIX u. Linux

Silberschatz
3.1-3.3: Process Concept
21.4: Linux

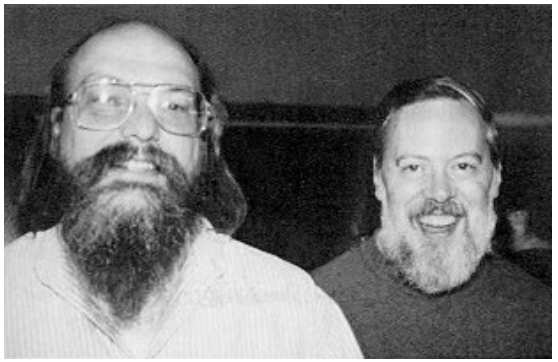
Tanenbaum
2.2: Threads

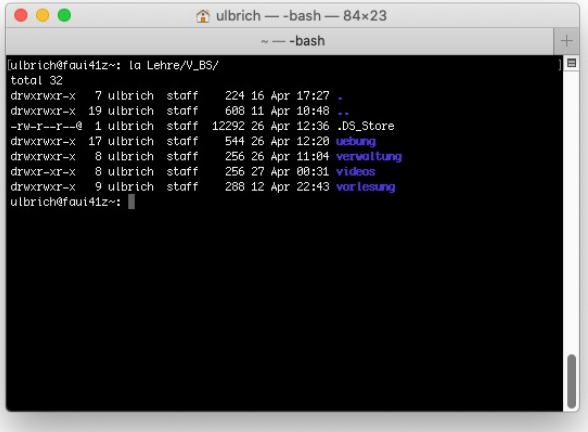
Silberschatz
4: Multithreaded Prog.

Inhalt

- **Das UNIX-Prozessmodell**
  - Shells und E/A
  - UNIX-Philosophie
  - Prozesserzeugung
  - Prozesszustände
- Leichtgewichtige Prozessmodelle
  - „Gewicht“ von Prozessen
  - Leichtgewichtige Prozesse
  - Federgewichtige Prozesse

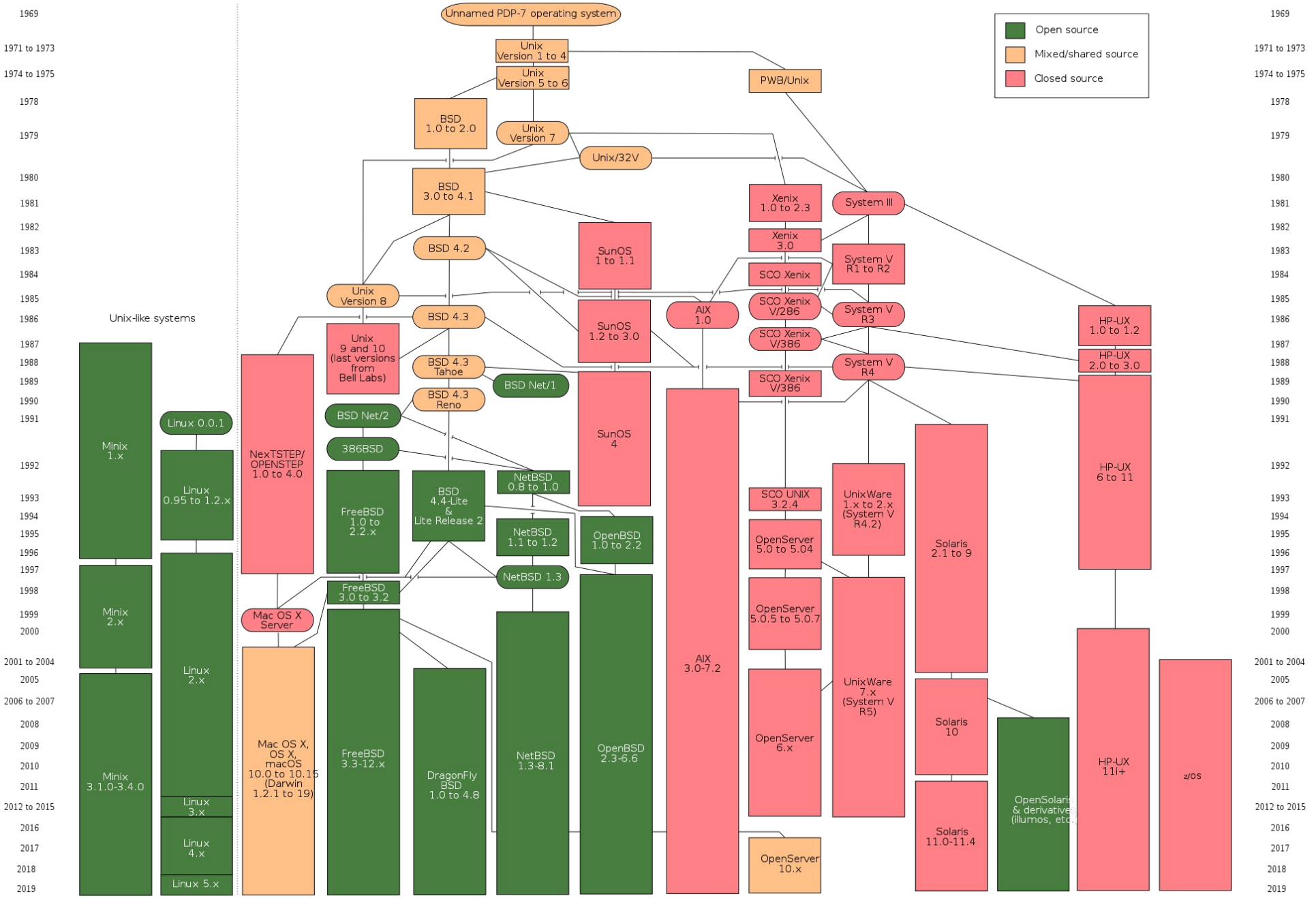
UNIX (K. Thompson, D. Ritchie, 1968)





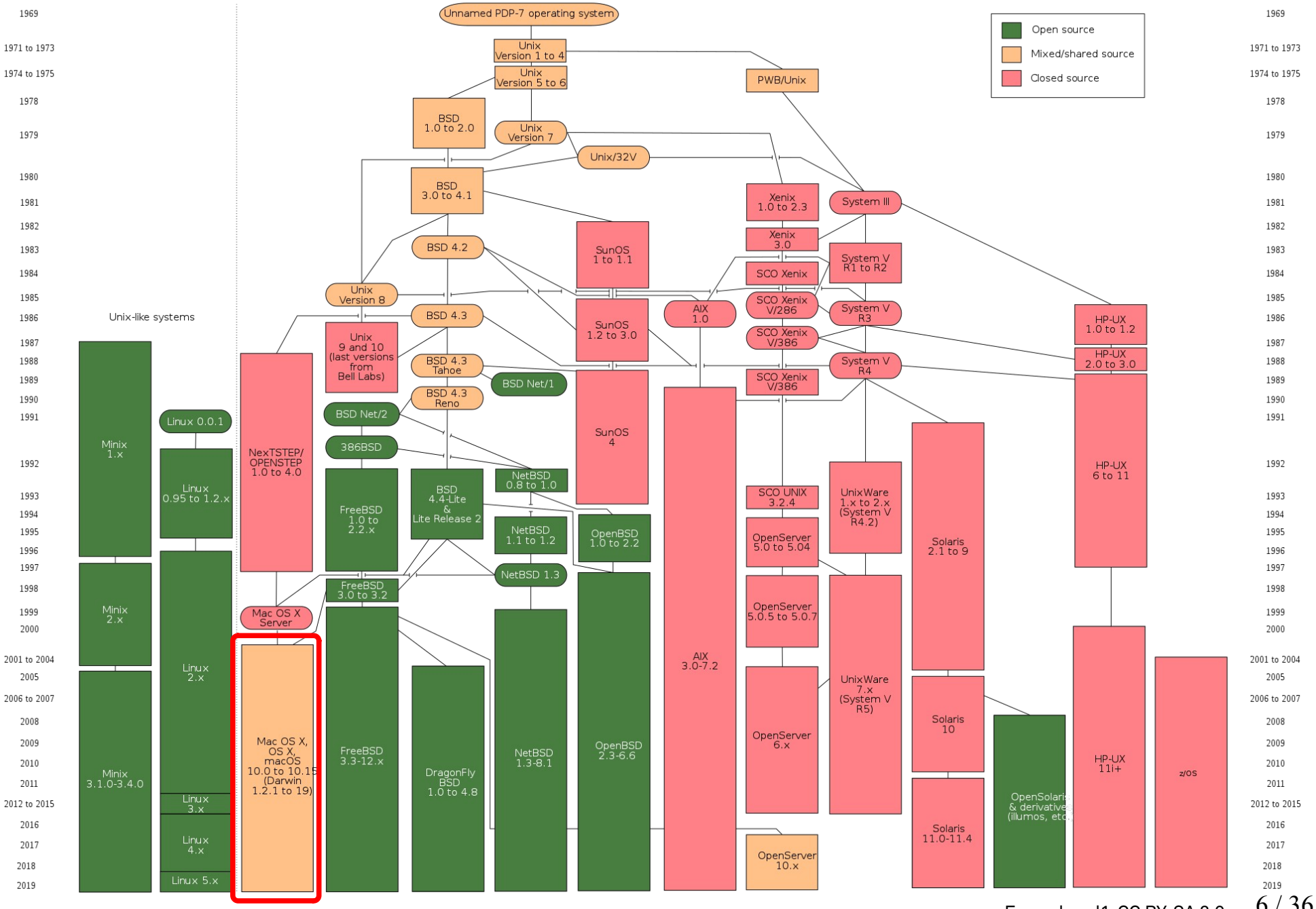
- Eine lange Geschichte ...
- Ursprung: Bell Labs
  - Alternative zu „Multics“
- Version 1 entstand auf einer DEC PDP 7
  - Assembler, 8K 18-Bit-Worte
- Version 3 in der Programmiersprache C realisiert

Die Geschichte von UNIX



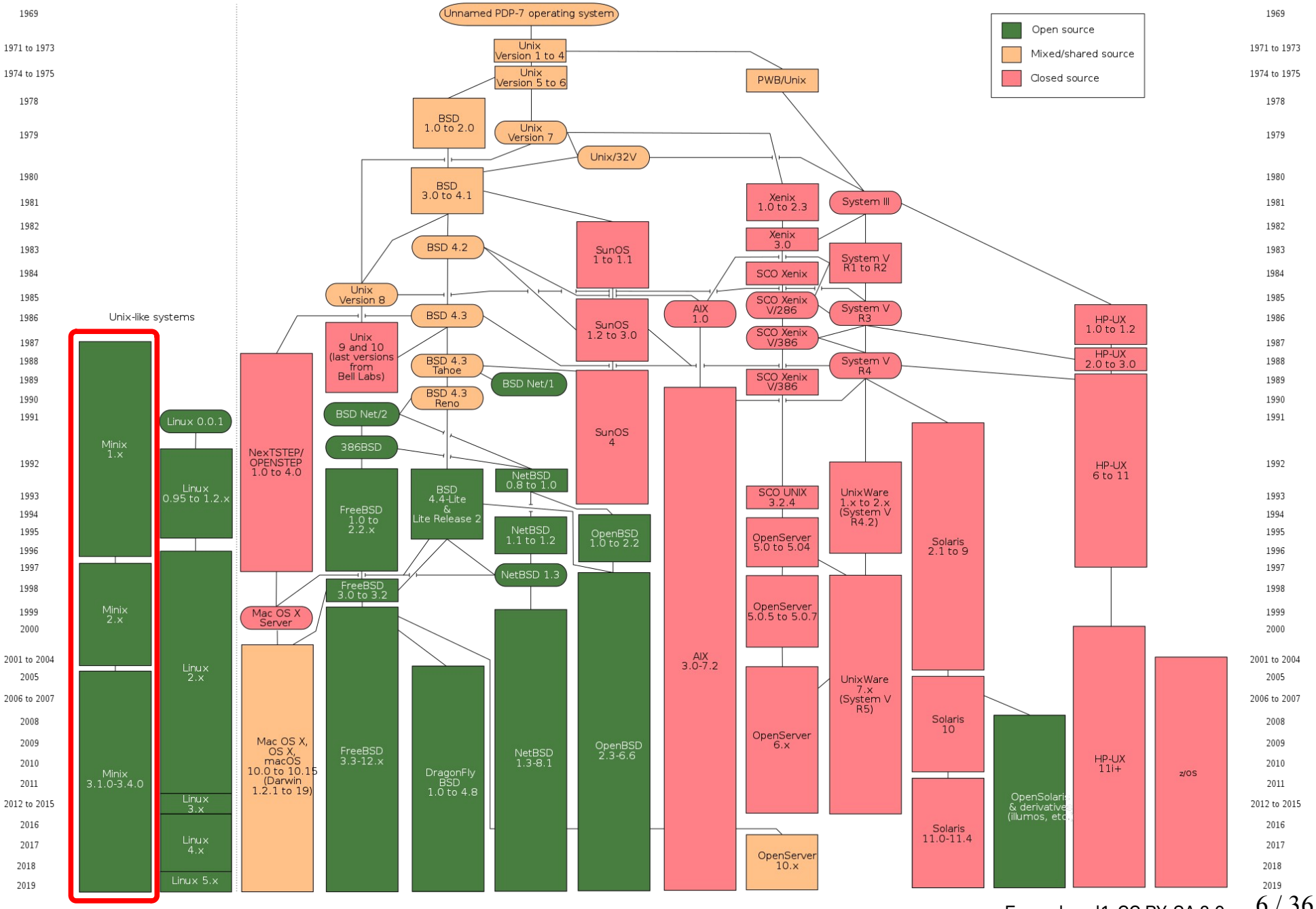
Eraserhead1, CC BY-SA 3.0

Die Geschichte von UNIX



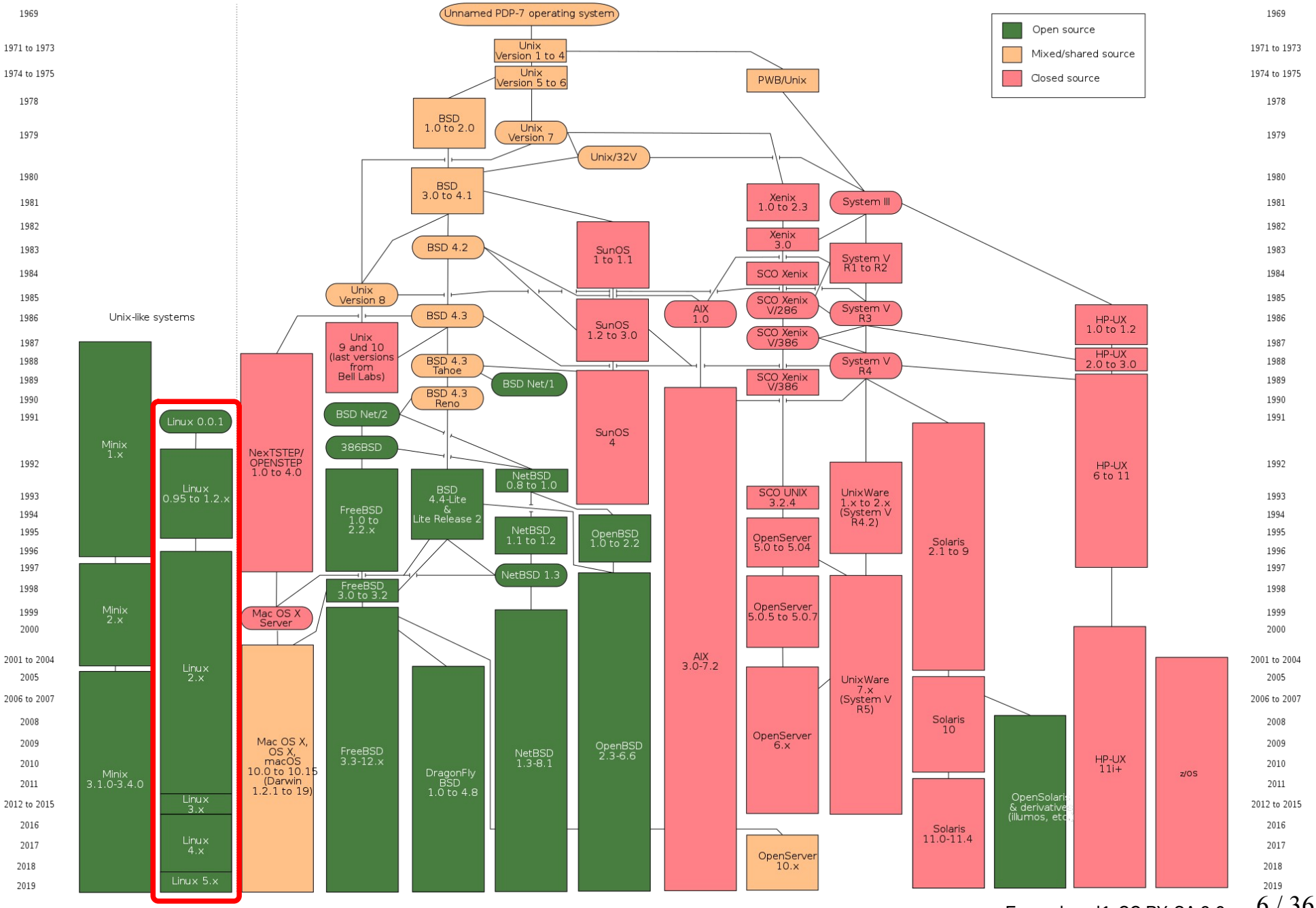
Eraserhead1, CC BY-SA 3.0

Die Geschichte von UNIX



Eraserhead1, CC BY-SA 3.0

Die Geschichte von UNIX



Eraserhead1, CC BY-SA 3.0

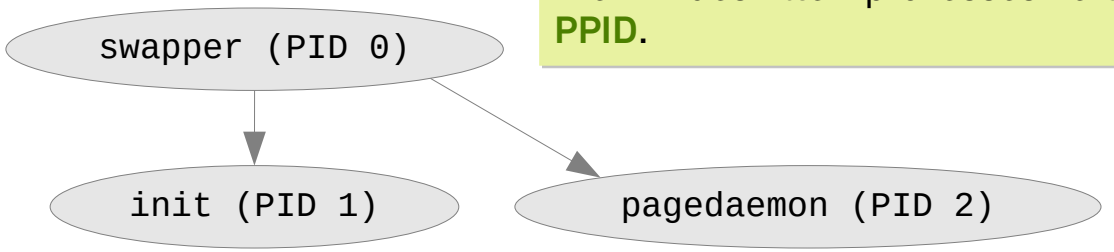
UNIX-Prozesse …

- sind primäres **Strukturierungskonzept** für Aktivitäten
  - Anwendungsprozesse und Systemprozesse
- können leicht und schnell weitere Prozesse erzeugen
  - Elternprozess → Kindprozess

UNIX-Prozesse …

- sind primäres **Strukturierungskonzept** für Aktivitäten
  - Anwendungsprozesse und Systemprozesse
- können leicht und schnell weitere Prozesse erzeugen
  - Elternprozess → Kindprozess
- bilden eine **Prozess-Hierarchie:**

Jeder UNIX-Prozess hat eine eindeutige Nummer (Prozess-ID). Die **PID** des Elternprozesses heißt **PPID**.

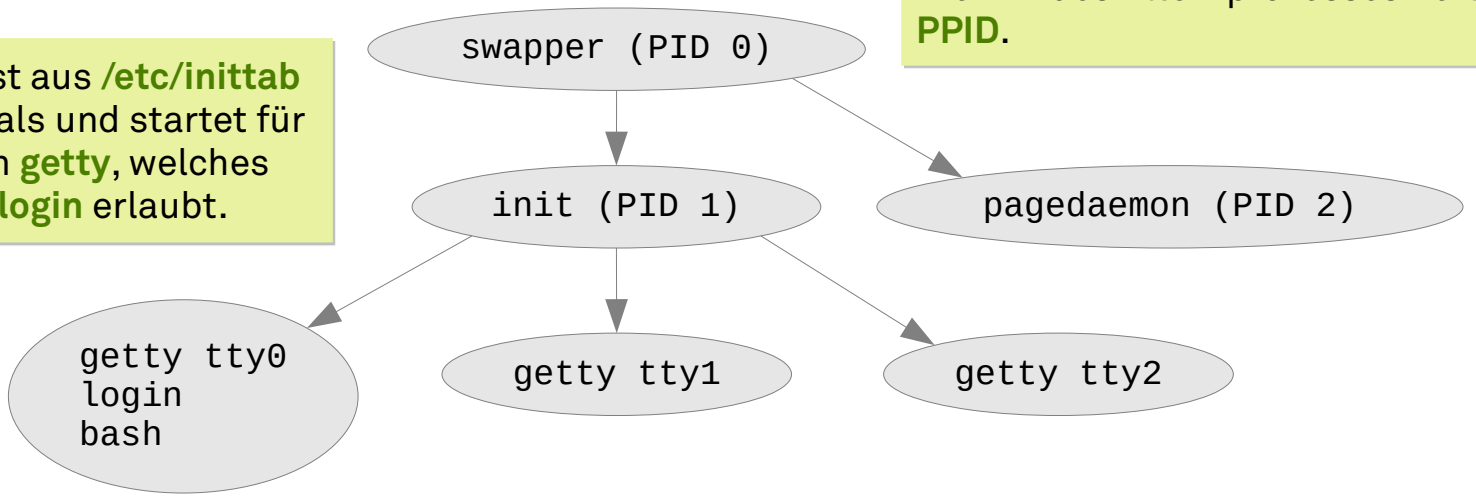


UNIX-Prozesse …

- sind primäres **Strukturierungskonzept** für Aktivitäten
  - Anwendungsprozesse und Systemprozesse
- können leicht und schnell weitere Prozesse erzeugen
  - Elternprozess → Kindprozess
- bilden eine **Prozess-Hierarchie:**

Jeder UNIX-Prozess hat eine eindeutige Nummer (Prozess-ID). Die **PID** des Elternprozesses heißt **PPID**.

Der **init**-Prozess liest aus **/etc/inittab** die Liste der Terminals und startet für jedes das Programm **getty**, welches die Einwahl mittels **login** erlaubt.

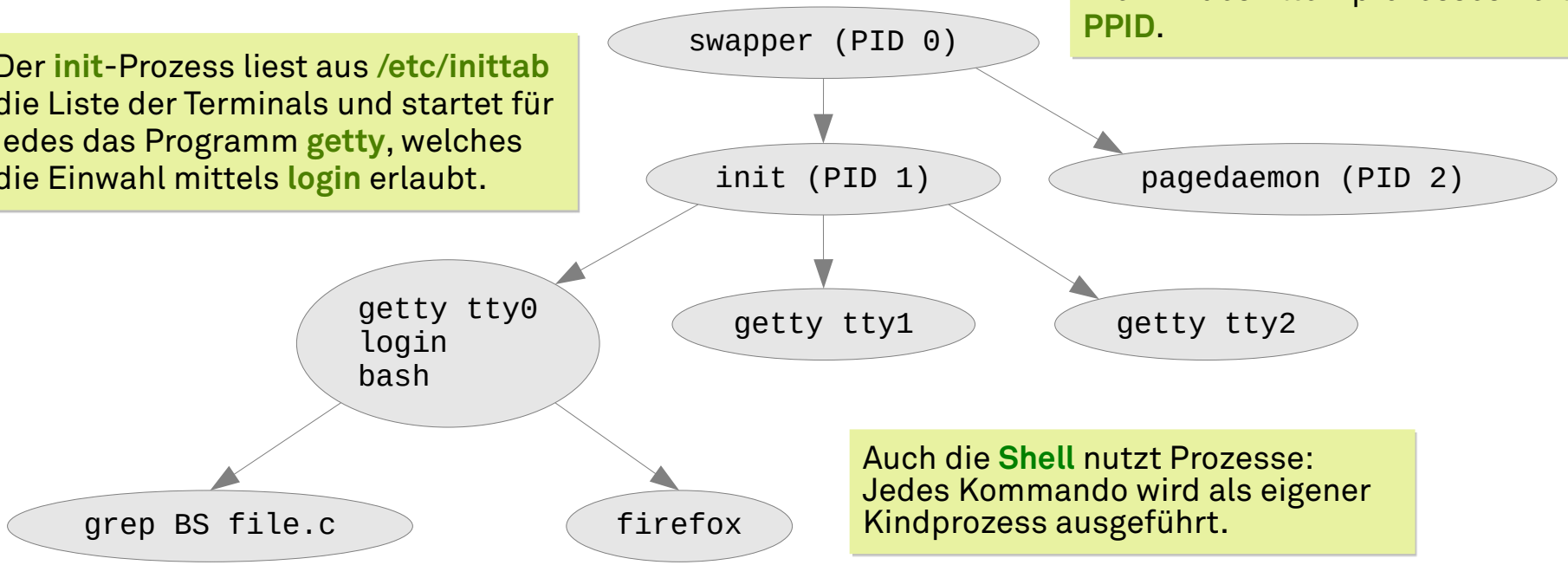


UNIX-Prozesse …

- sind primäres **Strukturierungskonzept** für Aktivitäten
  - Anwendungsprozesse und Systemprozesse
- können leicht und schnell weitere Prozesse erzeugen
  - Elternprozess → Kindprozess
- bilden eine **Prozess-Hierarchie:**

Jeder UNIX-Prozess hat eine eindeutige Nummer (Prozess-ID). Die **PID** des Elternprozesses heißt **PPID**.

Der **init**-Prozess liest aus **/etc/inittab** die Liste der Terminals und startet für jedes das Programm **getty**, welches die Einwahl mittels **login** erlaubt.



Auch die **Shell** nutzt Prozesse: Jedes Kommando wird als eigener Kindprozess ausgeführt.

UNIX-Shells

- **Schale** (*shell*), die den **Kern** (*kernel*) umgibt
  - Eingeführt mit Multics (um 1964) als Dialogstation-Kommandointerpreter
- Textbasierte Nutzerschnittstelle zum Starten von Kommandos:
  - Suche im Dateisystem entsprechend `$PATH` (z.B. `/usr/bin:/bin:...`)

```
ulbrich@ios:~$ which emacs
/usr/bin/emacs
```

Das Kommando **which** zeigt an, *wo* ein bestimmtes Kommando gefunden wird.

  - Jedes ausgeführte Kommando ist ein eigener Kindprozess
  - Typischerweise blockiert die Shell bis das Kommando terminiert
  - Man kann aber auch Kommandos stoppen und fortsetzen (*job control*) oder sie im Hintergrund ausführen ...

UNIX-*Shells*: *Job Control*

```
ulbrich@ios:~$ emacs foo.c
```

- Kommando wird gestartet
- die *Shell* blockiert

Ctrl-Z

```
[1]+  Stopped    emacs foo.c
ulbrich@ios:~$ kate bar.c &
[2] 19504
ulbrich@ios:~$ jobs
[1]+  Stopped    emacs foo.c
[2]-  Running    kate bar.c &
ulbrich@ios:~$ bg %1
[1]+ emacs foo.c &
ulbrich@ios:~$ jobs
[1]-  Running    emacs foo.c &
[2]+  Running    kate bar.c &
```

UNIX-*Shells*: *Job Control*

```
ulbrich@ios:~$ emacs foo.c
```

- Kommando wird gestartet
- die *Shell* blockiert

Ctrl-Z

- Kommando wird gestoppt
- die *Shell* läuft weiter

```
[1]+  Stopped    emacs foo.c
ulbrich@ios:~$ kate bar.c &
[2] 19504
ulbrich@ios:~$ jobs
[1]+  Stopped    emacs foo.c
[2]-  Running    kate bar.c &
ulbrich@ios:~$ bg %1
[1]+ emacs foo.c &
ulbrich@ios:~$ jobs
[1]-  Running    emacs foo.c &
[2]+  Running    kate bar.c &
```

UNIX-*Shells*: *Job Control*

```
ulbrich@ios:~$ emacs foo.c
```

- Kommando wird gestartet
- die *Shell* blockiert

Ctrl-Z

- Kommando wird gestoppt
- die *Shell* läuft weiter

```
[1]+  Stopped    emacs foo.c
ulbrich@ios:~$ kate bar.c &
[2] 19504
ulbrich@ios:~$ jobs
[1]+  Stopped    emacs foo.c
[2]-  Running    kate bar.c &
ulbrich@ios:~$ bg %1
[1]+ emacs foo.c &
ulbrich@ios:~$ jobs
[1]-  Running    emacs foo.c &
[2]+  Running    kate bar.c &
```

- Durch das **&** am Ende wird **kate** im Hintergrund gestartet

UNIX-*Shells*: *Job Control*

```
ulbrich@ios:~$ emacs foo.c
```

- Kommando wird gestartet
- die *Shell* blockiert

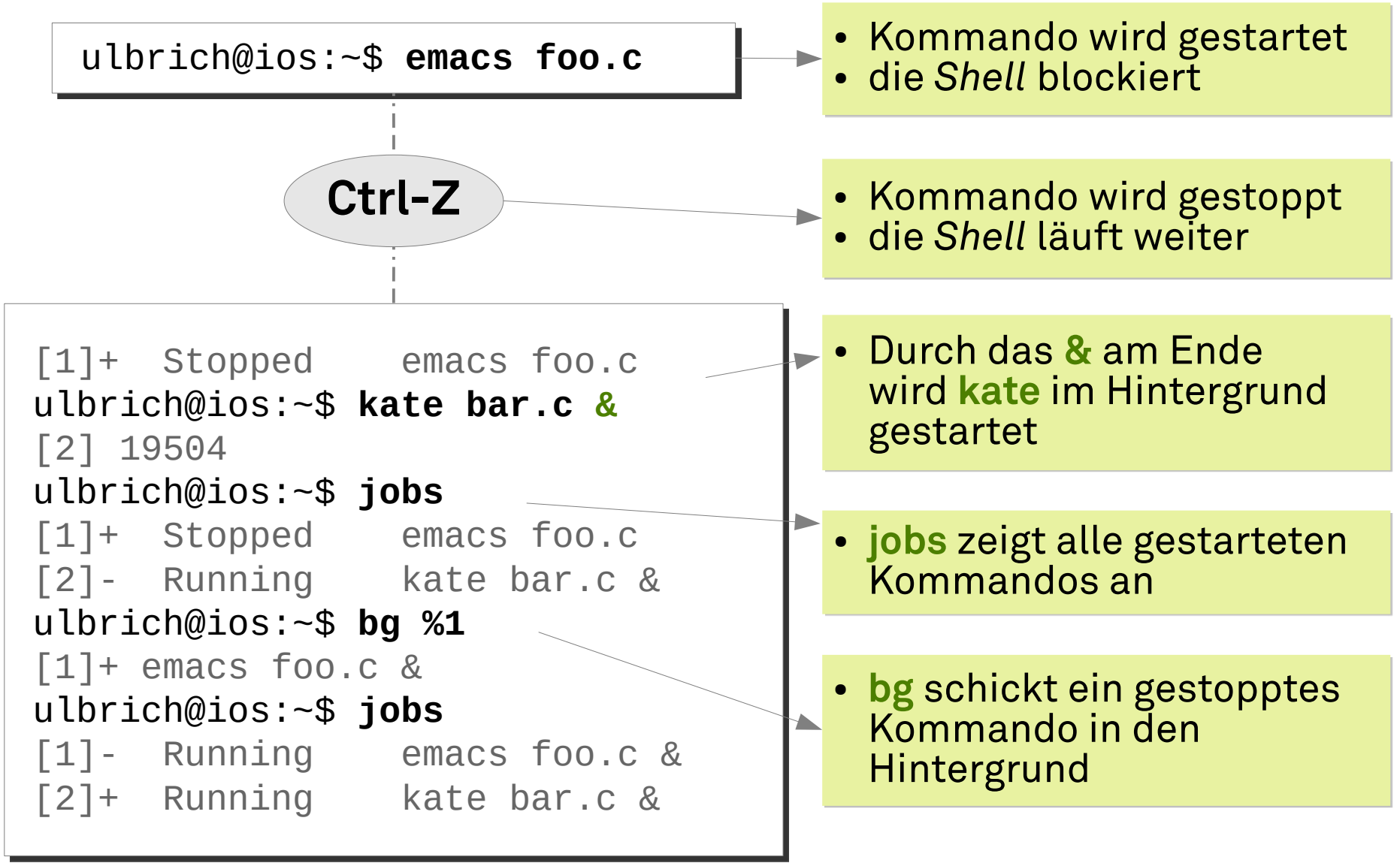
Ctrl-Z

- Kommando wird gestoppt
- die *Shell* läuft weiter

```
[1]+  Stopped    emacs foo.c
ulbrich@ios:~$ kate bar.c &
[2] 19504
ulbrich@ios:~$ jobs
[1]+  Stopped    emacs foo.c
[2]-  Running    kate bar.c &
ulbrich@ios:~$ bg %1
[1]+ emacs foo.c &
ulbrich@ios:~$ jobs
[1]-  Running    emacs foo.c &
[2]+  Running    kate bar.c &
```

- Durch das **&** am Ende wird **kate** im Hintergrund gestartet
- **jobs** zeigt alle gestarteten Kommandos an

UNIX-*Shells*: *Job Control*



Standard-E/A-Kanäle von Prozessen

- Normalerweise verbunden mit dem Terminal, in dem die Shell läuft, die den Prozess gestartet hat:
  - **Standard-Eingabe** — Zum Lesen von Benutzereingaben (*Tastatur*)
  - **Standard-Ausgabe** — Textausgaben des Prozesses (*Terminal-Fenster*)
  - **Standard-Fehlerausgabe** — Separater Kanal für Fehlermeldungen (*normalerweise auch das Terminal*)
- Praktisch alle Kommandos akzeptieren auch Dateien als Ein- oder Ausgabekanäle (statt des Terminals) → Operatoren
- Shells bieten eine einfache Syntax, um die Standard-E/A-Kanäle umzuleiten …

Standard-E/A-Kanäle umleiten

```
ulbrich@ios:~$ ls -l > d1
ulbrich@ios:~$ grep "May  4" < d1 > d2
ulbrich@ios:~$ wc < d2
  2  18 118
```

Das gleiche noch etwas kompakter ...

```
ulbrich@ios:~$ ls -l | grep "May  4" | wc
  2  18 118
```

Standard-E/A-Kanäle umleiten

Umleitung der Standard-Ausgabe in die Datei **d1** mit **>**

```
ulbrich@ios:~$ ls -l > d1
ulbrich@ios:~$ grep "May  4" < d1 > d2
ulbrich@ios:~$ wc < d2
  2  18 118
```

Das gleiche noch etwas kompakter ...

```
ulbrich@ios:~$ ls -l | grep "May  4" | wc
  2  18 118
```

Standard-E/A-Kanäle umleiten

Umleitung der Standard-Ausgabe in die Datei **d1** mit **>**

```
ulbrich@ios:~$ ls -l > d1
ulbrich@ios:~$ grep "May  4" < d1 > d2
ulbrich@ios:~$ wc < d2
   2   18 118
```

Umleitung der Standard-Eingabe auf die Datei **d2** mit **<**

Das gleiche noch etwas kompakter ...

```
ulbrich@ios:~$ ls -l | grep "May  4" | wc
   2   18 118
```

Standard-E/A-Kanäle umleiten

Umleitung der Standard-Ausgabe in die Datei **d1** mit **>**

```
ulbrich@ios:~$ ls -l > d1
ulbrich@ios:~$ grep "May  4" < d1 > d2
ulbrich@ios:~$ wc < d2
   2   18 118
```

Umleitung der Standard-Eingabe auf die Datei **d2** mit **<**

Das gleiche noch etwas kompakter ...

```
ulbrich@ios:~$ ls -l | grep "May  4" | wc
   2   18 118
```

Mit **|** (*pipe*) verbindet die *Shell* die Standard-Ausgabe des linken mit der Standard-Eingabe des rechten Prozesses.

Die UNIX-Philosophie

Doug McIlroy, der Erfinder der UNIX-*Pipes*, fasste die Philosophie hinter UNIX einmal wie folgt zusammen:

“ ***This is the Unix philosophy:***

- *Write programs that do one thing and do it well.*
- *Write programs to work together.*
- *Write programs to handle text streams, because that is a universal interface.”*

Die UNIX-Philosophie

Doug McIlroy, der Erfinder der UNIX-*Pipes*, fasste die Philosophie hinter UNIX einmal wie folgt zusammen:

“ ***This is the Unix philosophy:***

- *Write programs that do one thing and do it well.*
- *Write programs to work together.*
- *Write programs to handle text streams, because that is a universal interface.”*

Für gewöhnlich wird das abgekürzt:

„Do one thing, do it well.“

Einige Systemaufrufe der UNIX-Prozesssteuerung

Ein erster Überblick ...

- **`getpid`**`(2)` liefert PID des laufenden Prozesses
- **`getppid`**`(2)` liefert PID des Elternprozesses (PPID)
- **`getuid`**`(2)` liefert die Benutzerkennung des laufenden Prozesses (UID)

- **`fork`**`(2)` erzeugt neuen Kindprozess
- **`exit`**`(3)`, **`_exit`**`(2)` beendet den laufenden Prozess
- **`wait`**`(2)` wartet auf die Beendigung eines Kindprozesses
- **`execve`**`(2)` lädt und startet ein Programm im Kontext des laufenden Prozesses

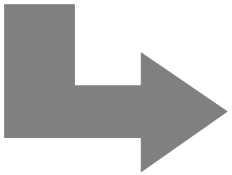
Der fork() Systemaufruf

System Call: `pid_t fork (void)`

- Dupliziert den laufenden Prozess (Prozesserzeugung!)
- Der Kindprozess erbt ...
  - Adressraum (code, data, bss, stack)
  - Benutzerkennung
  - Standard-E/A-Kanäle
  - Prozessgruppe, Signaltabelle (dazu später mehr)
  - Offene Dateien, aktuelles Arbeitsverzeichnis (dazu viel später mehr)
- Nicht kopiert wird ...
  - *Process* ID (PID), *Parent Process* ID (PPID)
  - anhängige Signale, *Accounting*-Daten, ...
- Ein Prozess ruft **fork** auf, aber zwei kehren zurück!

Verwendung von fork()

```c
/* includes */
int main () {
  int pid;
  printf("Elternpr.: PID %d PPID %d\n", getpid(), getppid());
  pid = fork(); /* Prozess wird dupliziert!
                   Beide laufen an dieser Stelle weiter. */
  if (pid > 0)
    printf("Im Elternprozess, Kind-PID %d\n", pid);
  else if (pid == 0)
    printf("Im Kindprozess, PID %d PPID %d\n",
           getpid(), getppid());
  else
    printf("Oh, ein Fehler!\n"); /* mehr dazu in der TÜ */
}
```



```
ulbrich@ios:~$ ./fork
Elternpr.: PID 7553 PPID 4014
Im Kindprozess, PID 7554 PPID 7553
Im Elternprozess, Kind-PID 7554
```

Kosten der Prozesserzeugung

- Das Kopieren des Adressraums erzeugt hohe Kosten
  - Speicher und Ausführungszeit
  - Insbesondere bei direkt folgendem `exec..()` pure Verschwendung!
- Historische Lösung: `vfork()`
  - Elternprozess wird suspendiert, bis Kindprozess **exec..**() aufruft oder mit **_exit**() terminiert
  - Kindprozess benutzt Code und Daten des Elternprozesses (kein Kopieren!).
  - Der Kindprozess darf keine Daten verändern
    - teilweise nicht so einfach: z.B. kein `exit()` aufrufen, sondern `_exit()`!
- Heutige Lösung: **copy-on-write**
  - Mit Hilfe der MMU teilen sich Eltern- und Kindprozess dasselbe Code- und Datensegment
  - Erst wenn der Kindprozess Daten ändert, wird das Segment kopiert
  - Wenn nach dem `fork()` direkt ein `exec..()` folgt, kommt das nicht vor
  - `fork()` mit **copy-on-write** ist kaum langsamer als `vfork()`

Der _exit() Systemaufruf

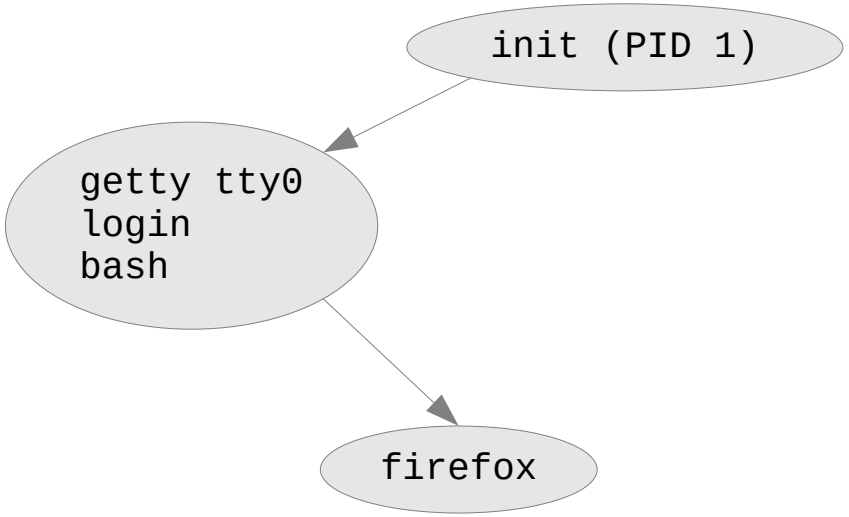
System Call: `void _exit (int)`

- **Terminiert den laufenden Prozess** und übergibt das Argument als *exit status* an den Elternprozess
  - Aufruf kehrt nicht zurück!
- Gibt die belegten Ressourcen des Prozesses frei
  - offene Dateien, belegter Speicher, ...
- Sendet dem eigenen Elternprozess das Signal `SIGCHLD`
- Die Bibliotheksfunktion **`exit(3)`** räumt zusätzlich noch die von der `libc` belegten Ressourcen auf
  - Gepufferte Ausgaben werden beispielsweise herausgeschrieben!
  - Normale Prozesse sollten **`exit(3)`** benutzen, nicht **`_exit`**

Verwaiste Prozesse

(engl. *orphan processes*)

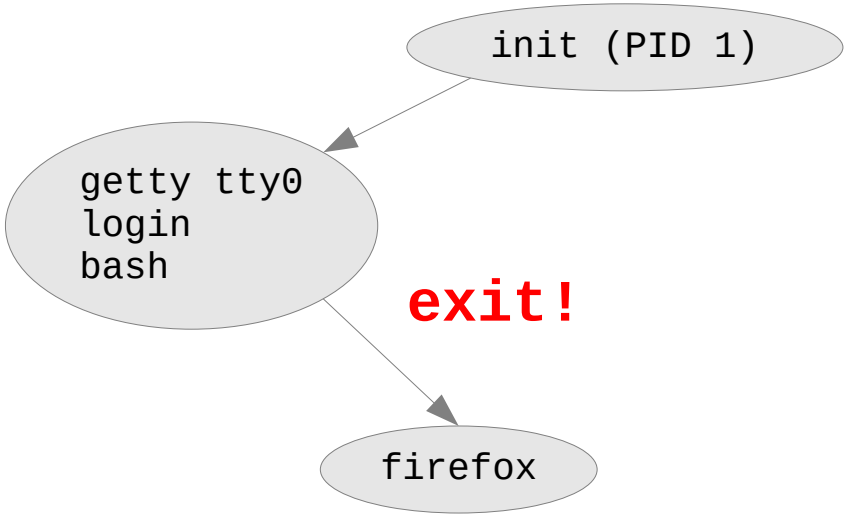
- Ein UNIX-Prozess wird zum **Waisenkind**, wenn sein Elternprozess terminiert.
- Was passiert mit der Prozesshierarchie?



Verwaiste Prozesse

(engl. *orphan processes*)

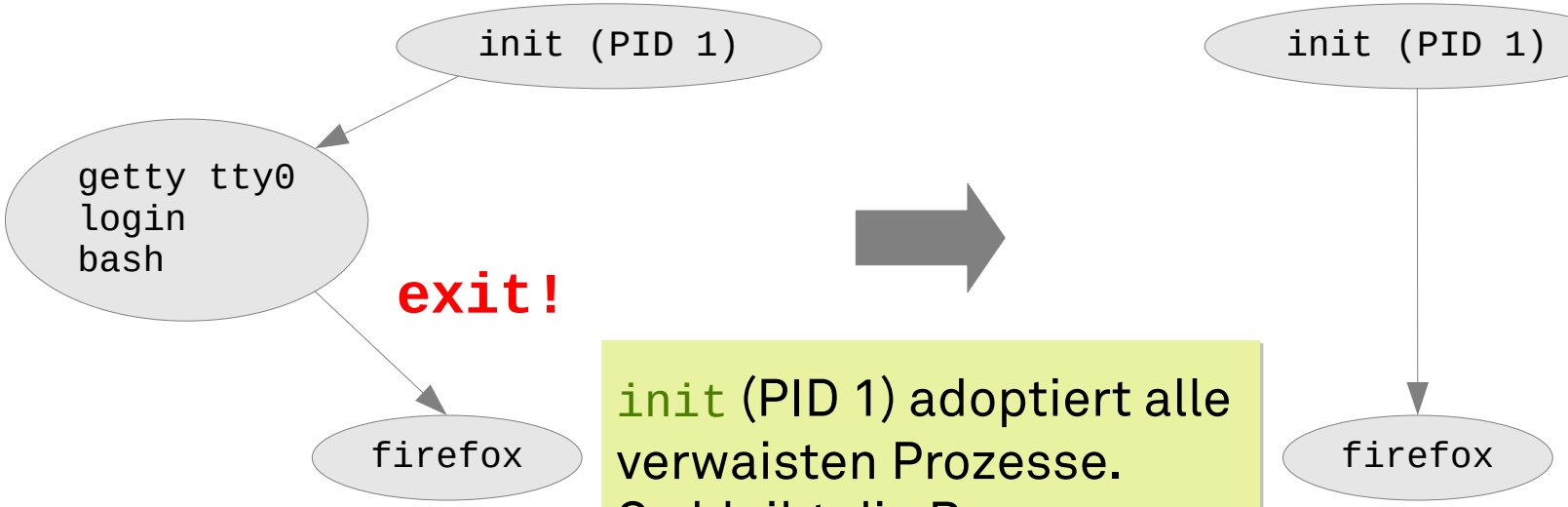
- Ein UNIX-Prozess wird zum **Waisenkind**, wenn sein Elternprozess terminiert.
- Was passiert mit der Prozesshierarchie?



Verwaiste Prozesse

(engl. *orphan processes*)

- Ein UNIX-Prozess wird zum **Waisenkind**, wenn sein Elternprozess terminiert.
- Was passiert mit der Prozesshierarchie?



`init` (PID 1) adoptiert alle verwaisten Prozesse.
So bleibt die Prozess-hierarchie intakt.

Der wait() Systemaufruf

System Call: `pid_t wait (int*)`

- Der aufrufenden Prozess **blockiert bis ein Kindprozess terminiert**
- Der Rückgabewert ist dessen PID
- Zeigerargument (`wstatus`) liefert u.a. den Exit-Status
  - Wert kann per Makro geprüft werden (z.B. `WIFEXITED(wstatus)`)
- Kehrt sofort zurück, falls bereits ein Kindprozess terminiert ist

Zombies

- Bevor der Exit-Status eines terminierten Prozesses mit Hilfe von **`wait`** abgefragt wird, ist er ein **Zombie**
- Die Ressourcen solcher Prozesse können freigegeben werden, aber die Prozessverwaltung muss sie noch kennen
  - Insbesondere der *exit status* muss gespeichert werden

```
ulbrich@ios:~$ ./wait &
ulbrich@ios:~$ ps
  PID TTY          TIME CMD
 4014 pts/4    00:00:00 bash
17892 pts/4    00:00:00 wait
17895 pts/4    00:00:00 wait <defunct>
17897 pts/4    00:00:00 ps
ulbrich@ios:~$ Exit Status: 42
```

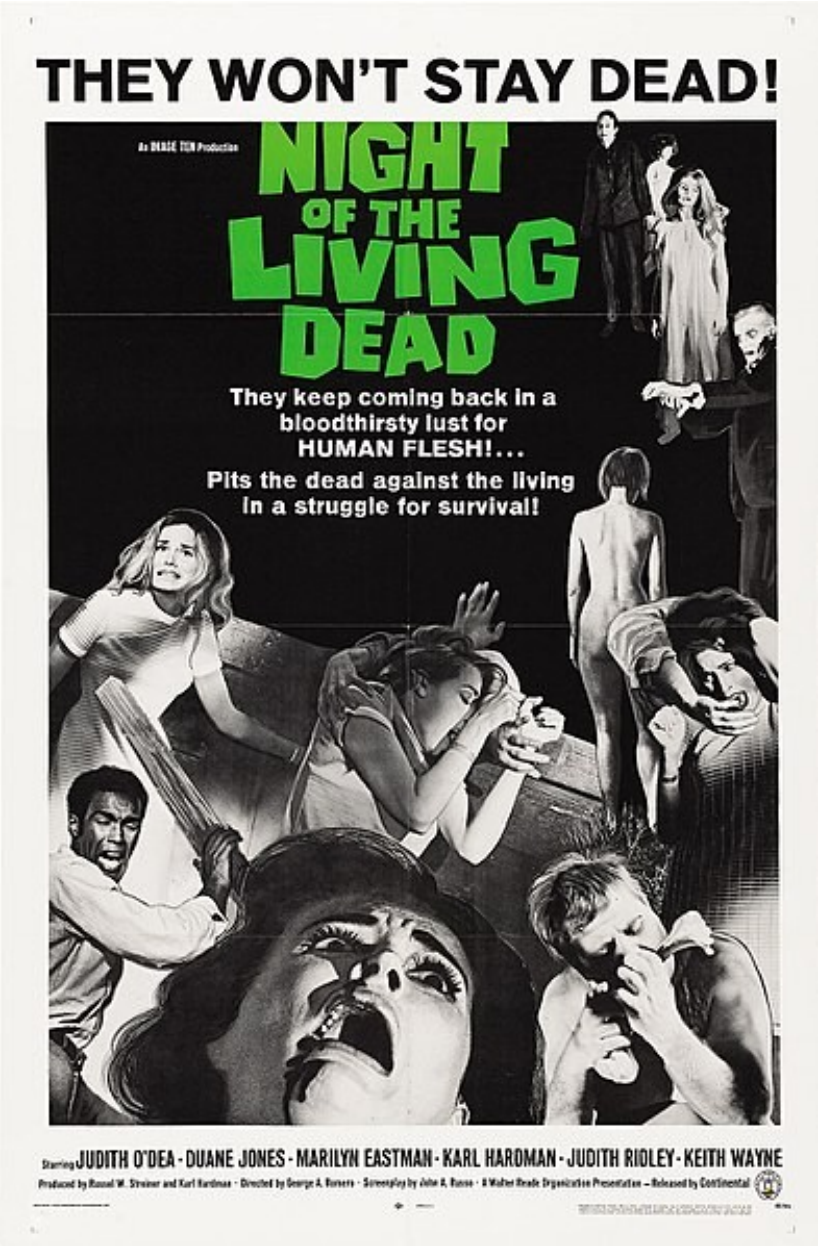
Beispielprogramm von eben während der 5 Sekunden Wartezeit.

Zombies werden von `ps` als `<defunct>` dargestellt.

Zombies ...

- Film vom 1968
- Regie: G. A. Romero

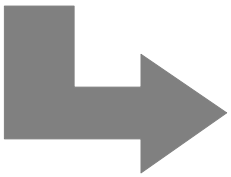
Wikipedia:
In 1999 the Library of Congress entered it into the United States National Film Registry with other films deemed „historically, culturally or aesthetically important.“



Quelle: Wikipedia (Public Domain)

Verwendung von wait()

```
... /* includes, main() { ... */
pid = fork(); /* Kindprozess erzeugen */
if (pid > 0) {
  int status;
  sleep(5); /* Bibliotheksfunktion: 5 Sek. schlafen */
  if (wait(&status) == pid && WIFEXITED(status))
    printf ("Exit Status: %d\n", WEXITSTATUS(status));
}
else if (pid == 0) {
  exit(42);
}
...
```

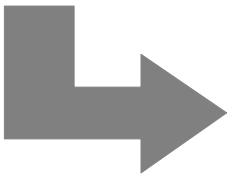


```
ulbrich@ios:~$ ./wait
Exit Status: 42
```

Verwendung von wait()

```
... /* includes, main() { ... */
pid = fork(); /* Kindprozess erzeugen */
if (pid > 0) {
  int status;
  sleep(5); /* Bibliotheksfunktion: 5 Sek. schlafen */
  if (wait(&status) == pid && WIFEXITED(status))
    printf ("Exit Status: %d\n", WEXITSTATUS(status));
}
else if (pid == 0) {
  exit(42);
}
...
```

Ein Prozess kann auch von außen *getötet* werden, d.h. er ruft nicht `exit` auf. In diesem Fall würde `WIFEXITED` 0 liefern.



```
ulbrich@ios:~$ ./wait
Exit Status: 42
```

UNIX-Prozesse im Detail: execve()

System Call:
```
int execve (const char *kommando,
            const char *args[],
            const char *envp[])
```

- Lädt und **startet das angegebene Kommando**
- Der Aufruf kehrt nur im Fehlerfall zurück
- Der komplette Adressraum wird ersetzt
- Es handelt sich aber weiterhin um denselben Prozess!
  - Selbe PID, PPID, offenen Dateien, ...
- Die **libc** biete Hilfsfunktionen, die intern execve aufrufen:
  - **`execl`**, **`execv`**, **`execlp`**, **`execvp`**, ...
  - Unterscheiden sich in Argumenten, Umgebungsvariablen, Suchpfad

Verwendung von exec..()

```
... /* includes, main() { ... */
char cmd[100], arg[100];
while (1) {
  printf ("Kommando?\n");
  scanf ("%99s %99s", cmd, arg);
  pid = fork(); /* Prozess wird dupliziert!
                   Beide laufen an dieser Stelle weiter. */
  if (pid > 0) {
    int status;
    if (wait(&status) == pid && WIFEXITED(status))
      printf ("Exit Status: %d\n", WEXITSTATUS(status));
  }
  else if (pid == 0) {
    execlp(cmd, cmd, arg, NULL);
    printf ("exec fehlgeschlagen\n");
  }
...
}
```

Diskussion: Warum kein forkexec()?

- Durch die Trennung von `fork` und `execve` hat der Elternprozess mehr Kontrolle:
  - Operationen im Kontext des Kindprozesses ausführen
  - Voller Zugriff auf die Daten des Elternprozesses
- Shells nutzen diese Möglichkeit zum Beispiel zur …
  - Umleitung der Standard-E/A-Kanäle
  - Aufsetzen von Pipes

UNIX-Prozesszustände

- ein paar mehr als wir bisher kannten ...

Bild in Anlehnung an M. Bach „UNIX – Wie funktioniert das Betriebssystem?“

Inhalt

- Das UNIX-Prozessmodell
  - Shells und E/A
  - UNIX-Philosophie
  - Prozesserzeugung
  - Prozesszustände
- **Leichtgewichtige Prozessmodelle**
  - „Gewicht“ von Prozessen
  - Leichtgewichtige Prozesse
  - Federgewichtige Prozesse

Das „Gewicht“ von Prozessen

- Das **Gewicht** eines Prozesses ist ein **bildlicher Ausdruck** für die Größe seines Kontexts und damit die Zeit, die für einen Prozesswechsel benötigt wird.
  - CPU-Zuteilungsentscheidung
  - alten Kontext sichern
  - neuen Kontext laden
- Klassische UNIX-Prozesse sind **schwergewichtig**.

Leichtgewichtige Prozesse (*Threads*)

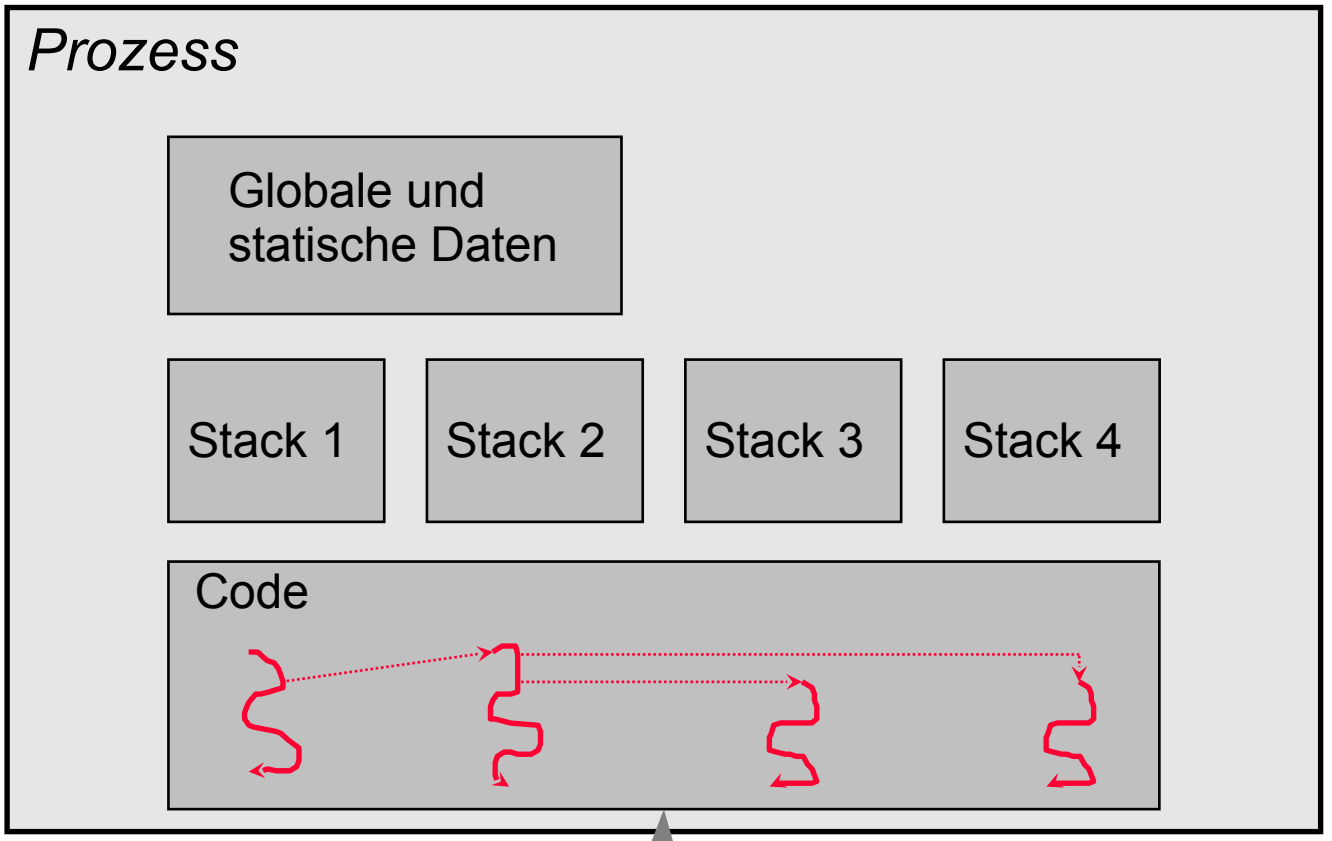
- Die 1:1-Beziehung zwischen Kontrollfluss und Adressraum wird aufgebrochen
  - Eng kooperierende **Threads** (deutsch „Fäden“)
  - **Adressraum teilen** (*code* + *data* + *bss* + *heap*, aber nicht *stack*!)
- **Vorteile**:
  - **Auslagerung aufwändiger Operationen** in einen leichtgewichtigen Hilfsprozess, während der Elternprozess erneut auf Eingabe reagieren kann
    - Typisches Beispiel: Webserver
  - Programme, die aus mehreren unabhängigen Kontrollflüssen bestehen, profitieren unmittelbar von Multiprozessor-Hardware
  - Schneller Kontextwechsel, wenn man im selben Adressraum bleibt
  - Je nach *Scheduler* eventuell mehr Rechenzeit
- **Nachteil**:
  - **Programmierung ist schwierig**:
    Zugriff auf gemeinsame Daten muss koordiniert werden.

Federgewichtige Prozesse

(engl. ***User-Level Threads***)

- Werden komplett auf der Anwendungsebene implementiert. Das Betriebssystem weiß nichts davon.
  - Realisiert durch Bibliothek: *User-Level Thread Package*
  - Im Gegensatz zu *Kernel-Level Threads*
- **Vorteile:**
  - Extrem schneller Kontextwechsel: Nur wenige Prozessorregister sind auszutauschen. Ein ***Trap* in den Kern entfällt**.
  - Jede Anwendung kann sich das passende *Thread-Package* wählen.
- **Nachteile:**
  - Blockierung eines federgewichtigen Prozesses führt zur Blockierung des ganzen Programms.
  - Kein Geschwindigkeitsvorteil durch Multi-Prozessoren.
  - Kein zusätzlicher Rechenzeitanteil.

Threads in Windows (1)



Ein Prozess enthält 1 bis N Threads, die auf denselben globalen Daten operieren.

Threads in Windows (2)

- **Prozess**: Umgebung und Adressraum für *Threads*
  - Ein Win32-Prozess enthält immer mindestens 1 *Thread*
- ***Thread***: Code-ausführende Einheit
  - Jeder *Thread* verfügt über einen eigenen *Stack* und Registersatz (insbes. Instruktionszeiger / PC = *program counter*)
  - *Threads* bekommen vom *Scheduler* Rechenzeit zugeteilt
- Alle *Threads* sind *Kernel-Level Threads*
  - *User-Level Threads* möglich („*Fibers*“), aber unüblich
- Strategie: Anzahl der Threads gering halten
  - Überlappte (asynchrone) E/A

Threads in Linux

- Linux implementiert POSIX Threads in Form der **pthread**-Bibliothek
- Möglich macht das ein Linux-spezifischer System Call

Linux System Call:

```
int _clone (int (*fn)(void*), void *stack
            int flags, void *arg, ...)
```

- Universelle Funktion, parametrisiert durch **flags**
  - `CLONE_VM` Adressraum gemeinsam nutzen
  - `CLONE_FS` Information über Dateisystem teilen
  - `CLONE_FILES` Dateideskriptoren (offene Dateien) teilen
  - `CLONE_SIGHAND` Gemeinsame Signalbehandlungstabelle

- Für Linux sind alle Threads und Prozesse **intern Tasks**:
  - Der Scheduler macht also keinen Unterschied.

Zusammenfassung

- Prozesse sind die zentrale Abstraktion für Aktivitäten in heutigen Betriebssystemen
- UNIX-Systeme stellen diverse System Calls zur Verfügung, um Prozesse zu erzeugen, zu verwalten und miteinander zu verknüpfen
  - alles im Sinne der Philosophie: *„Do one thing, do it well.“*
- Leichtgewichtige Fadenmodelle haben viele Vorteile
  - in UNIX-Systemen bis in die 90er Jahre nicht verfügbar
  - in Windows von Beginn an (ab NT) integraler Bestandteil