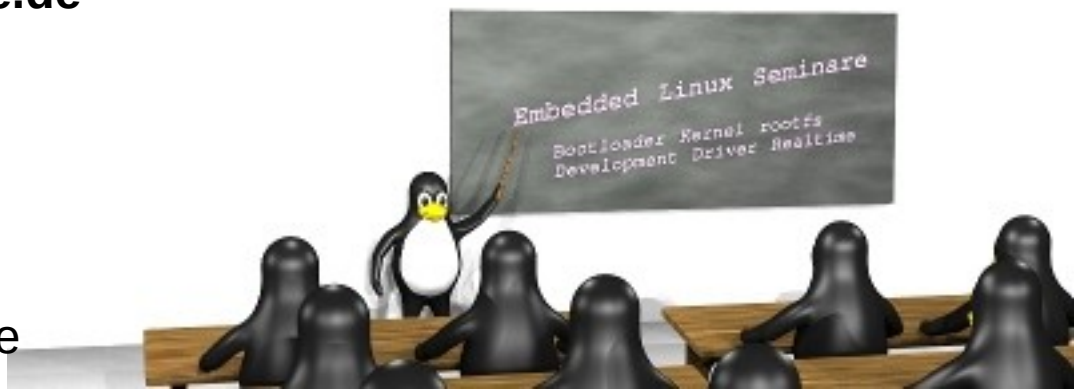
Embedded-Linux-Seminare

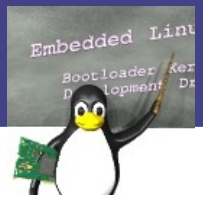
Linux als Betriebssystem

http://www.embedded-linux-seminare.de

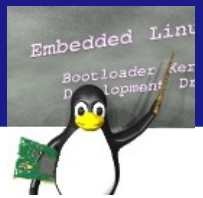
Diplom-Physiker Peter Börner
Spandauer Weg 4
37085 Göttingen

Tel.: 0551-7703465
Mail: info@embedded-linux-seminare.de



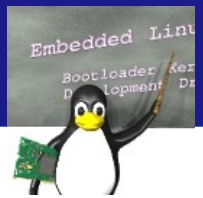


Kopier-Rechte



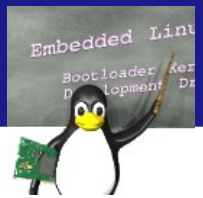
Anforderungen an Betriebssysteme

- Umgebung für Applikationen und User bereitstellen
- Zugriff auf die Hardware ermöglichen und verwalten
- Schutz des Systems vor falschen Zugriffen
  - Einige Betriebssysteme erlauben dem User direkten Zugriff auf die Hardware
    - DOS
    - Real-Time Systeme
  - Andere Betriebssysteme verbergen den Low-Level Zugriff auf die Hardware
    - Unix
    - Linux



Linux und Hardwarezugriff (1)

- Linux Kernel regelt den Low-Level Zugriff auf die Hardware
- Normale Applikationen haben nur Zugriff auf das virtuelle Filesystem und die Netzwerktreiber
- Zwei Level
  - Kernel-Space
    - Für Kernel und Treiber
    - Zugriffsrechte auf das komplette System
  - User-Space
    - User Applikationen und Bibliotheken
    - Kein direkter Zugriff auf die Hardware
- Zur Umsetzung muss die CPU mindestens zwei Zugriffs-Level unterstützen



Linux und Hardwarezugriff (2)

U s e r - S p a c e

Applikation

Applikation

Applikation

Applikation

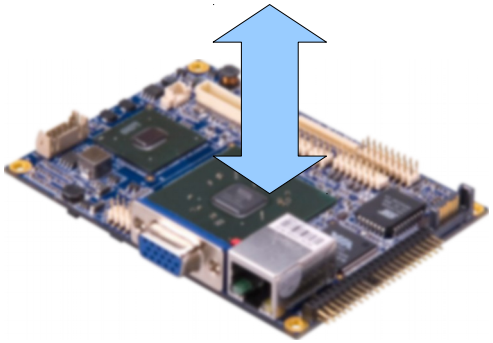
S y s t e m C a l l I n t e r f a c e S C I

Kernel

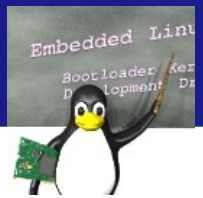
Treiber

Treiber

Treiber

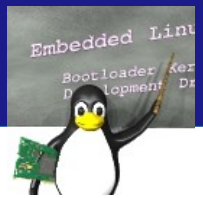


K e r n e l - S p a c e



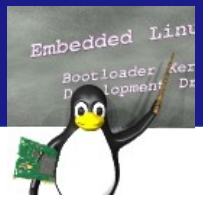
System Call Interface

- Ermöglicht den Übergang vom User-Space zum Kernel-Space
- Die Implementierung ist CPU abhängig
  - x86: int 0x80
  - arm: swi (Software Interrupt)
svc (Supervisor Call)
  - ppc: sc (System Call)
- Kernel und Treiber stellen Funktionen zur Verfügung, die über System-Calls zu erreichen sind.
- Es gibt ca. 300 System-Calls für
  - Datei und Device Zugriffe, Netzwerk, Inter-Prozess Kommunikation, Prozess Management, Speicher Verwaltung, Timer, Threads, etc.
- Das Interface ist stabil und es können nur neue System Calls durch Entwickler hinzugefügt werden.



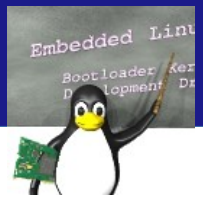
System-Call Beispiel x86

```
section .data
hello:

.ascii ''hello world!\n''

.section .text

.global   _start

_start:

     movl    $4, %eax          # 4 = write syscall
     movl    $1, %ebx          # 1 = stdout
     leal    hello, %ecx       # pointer to string
     movl    $13, %edx         # string length
     int     $0x80             # do syscall
     movl    $1, %eax          # 1 = exit syscall
     movl    $0, %ebx          # 0 = return code
     int     $0x80             # do syscall
```



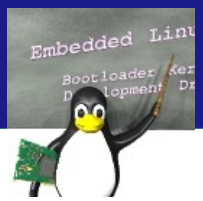
System-Call Beispiel arm

```
.data
msg:   .ascii "Hello World!\n"

.text
.global _start
_start:
        mov         r0, #1              # 1 = stdout
        ldr         r1, .L0             # string address
        mov         r2, #13             # string length
        mov         r7, #4              # 4 = write syscall
        svc         0x00000000          # syscall
        mov         r0, #0              # 0 = exit code
        mov         r7, #1              # 1 = exit syscall
        svc         0x00000000          # syscall
.L0:
    .word msg
```

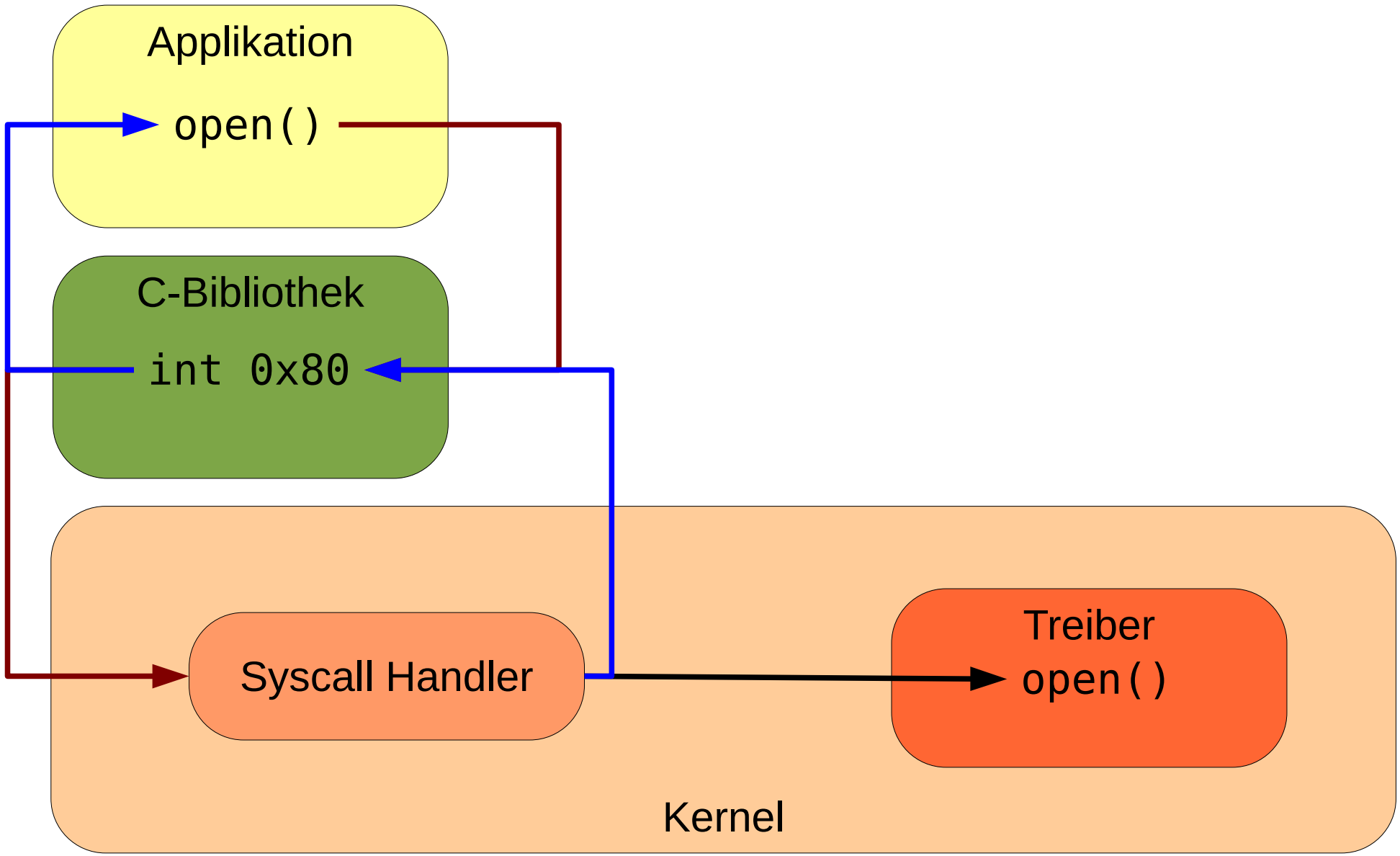


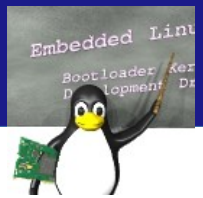
System Call

- Die System Calls können in Assembler umgesetzt werden
  - Vorteil: Das Programm ist einfach und klein
  - Nachteil: Das Programm ist nur mit viel Aufwand auf andere Hardware portierbar.
- Sinnvoll wäre ein Interface, das von der Applikation genutzt werden kann
  - System Calls in Bibliothek packen.
  - Bei Architekturwechsel muss nur diese neu generiert werden
  - Standard C Bibliothek
    - libc6 / GNU glibc2
    - eglibc
    - rawlib
    - dietlib
    - uclibc



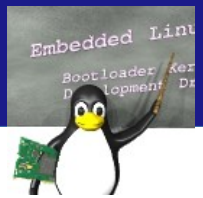
System Call


Applikation
open()
C-Bibliothek
int 0x80
Syscall Handler
Treiber
open()
Kernel



Standard C-Bibliothek

- Applikationen können gegen die Standard C-Bibliothek gelinkt werden.
  - statisch
    - Alle referenzierten Funktionen werden in die Applikation eingefügt
    - Die C-Bibliothek braucht auf dem Target nicht vorhanden zu sein
    - Die Applikation wird groß
    - Das gesamte System kann kleiner werden, da die C-Bibliothek wegfallen kann
  - dynamisch
    - Alle referenzierten Funktionen existieren nur in der Bibliothek
    - Die Applikation wird kleiner
    - Loader wird benötigt
    - C-Bibliothek muss auf dem Target vorhanden sein

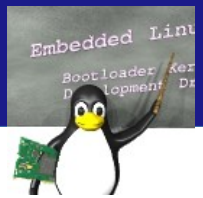


C-Bibliotheken

- Die Größe der fertigen Applikationen variiert mit den verschiedenen C-Bibliotheken

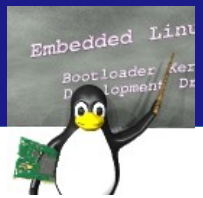
```
int main() {}
```

	Nicht gestripped	Gestripped
libc6 (dynamisch)	13 kB	3 kB
libc6 (statisch)	1519 kB	379 kB
µClibc (statisch)	6292 Bytes	1992 Bytes
dietlibc (statisch)	1384 Bytes	752 Bytes



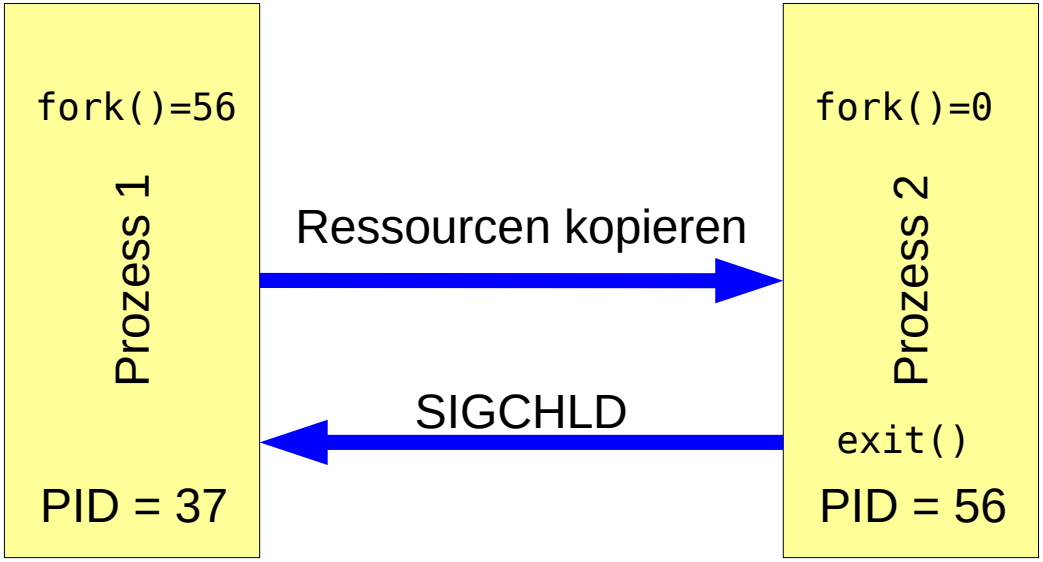
Prozesse

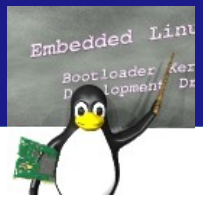
- Prozesse übernehmen die Arbeit im Linux-System
- Ein Prozess wird durch `fork()` erzeugt und enthält
  - Einen Adress-Raum mit dem Programm Code, Daten, Stack, Shared Libraries, etc.
  - Einen Thread der die `main()` Funktion ausführt
- Weitere Threads können im existierenden Prozess erzeugt werden und nutzen den selben Adress-Raum
- Jeder laufende Thread wird durch den Kernel mit einer Structure vom Typ `task_struct` verwaltet.
- Der Scheduler verwaltet den ersten Thread eines Prozesses genauso wie die weiteren erzeugten Threads.
- Jeder Prozess
  - glaubt der gesamte Speicher gehört ihm
  - glaubt die ganze CPU gehört ihm



Prozesse

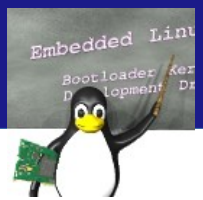
- Prozesse werden durch PID gekennzeichnet
  - eindeutiger Wert innerhalb des laufenden Systems
  - Default Maximum Wert ist 32768 (short int)
  - Maximum kann geändert werden
    - echo 16384 > /proc/sys/kernel/pid_max



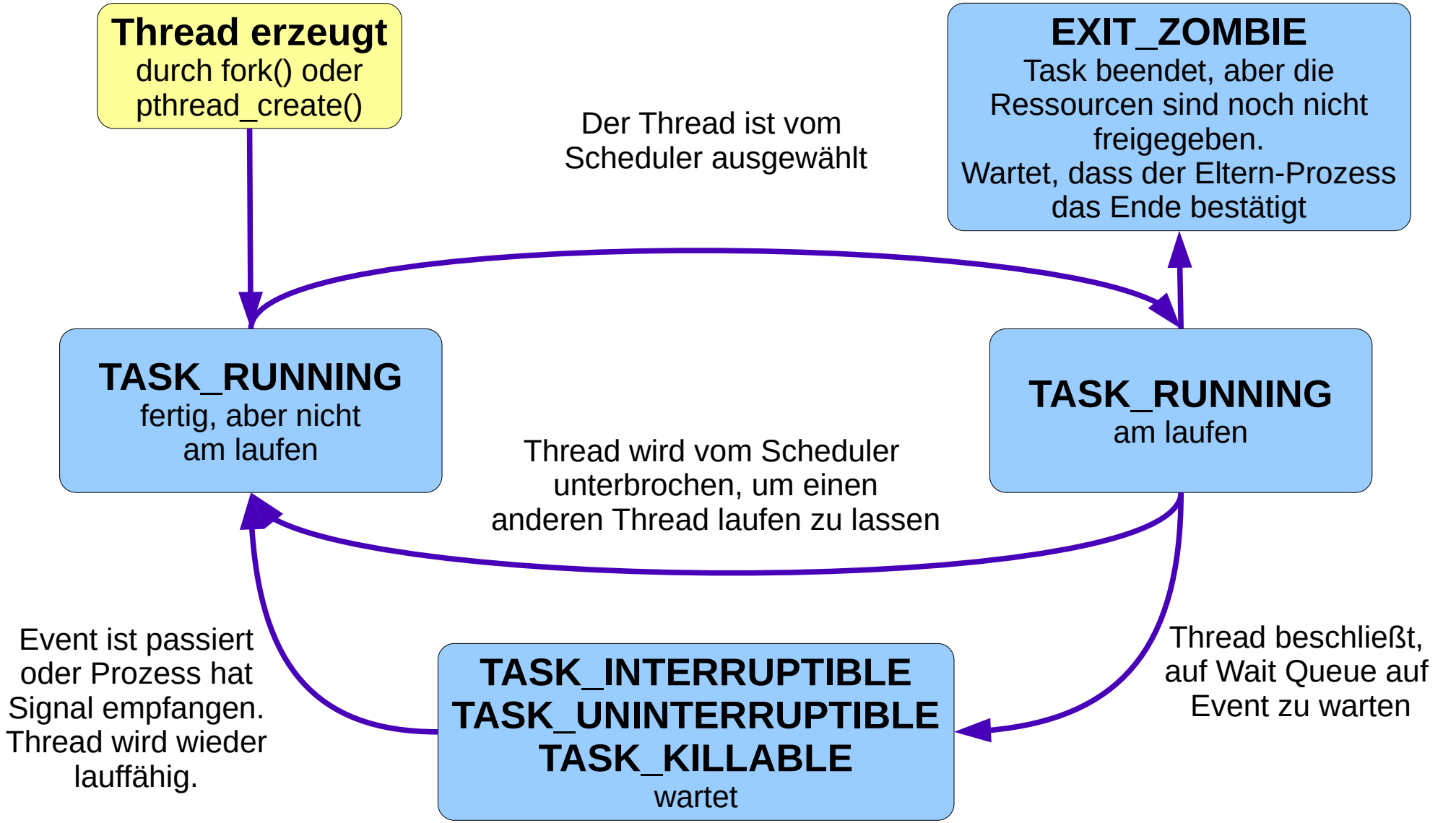


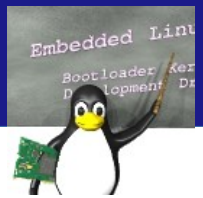
Prozesse

- Alle Prozesse des Linux Systems formen einen Baum
- Ein Prozess kann mehrere Childs haben
- Jeder Prozess hat nur einen Parent
- Der oberste Prozess ist `init`
- Prozesse können zu Gruppen zusammengefasst werden



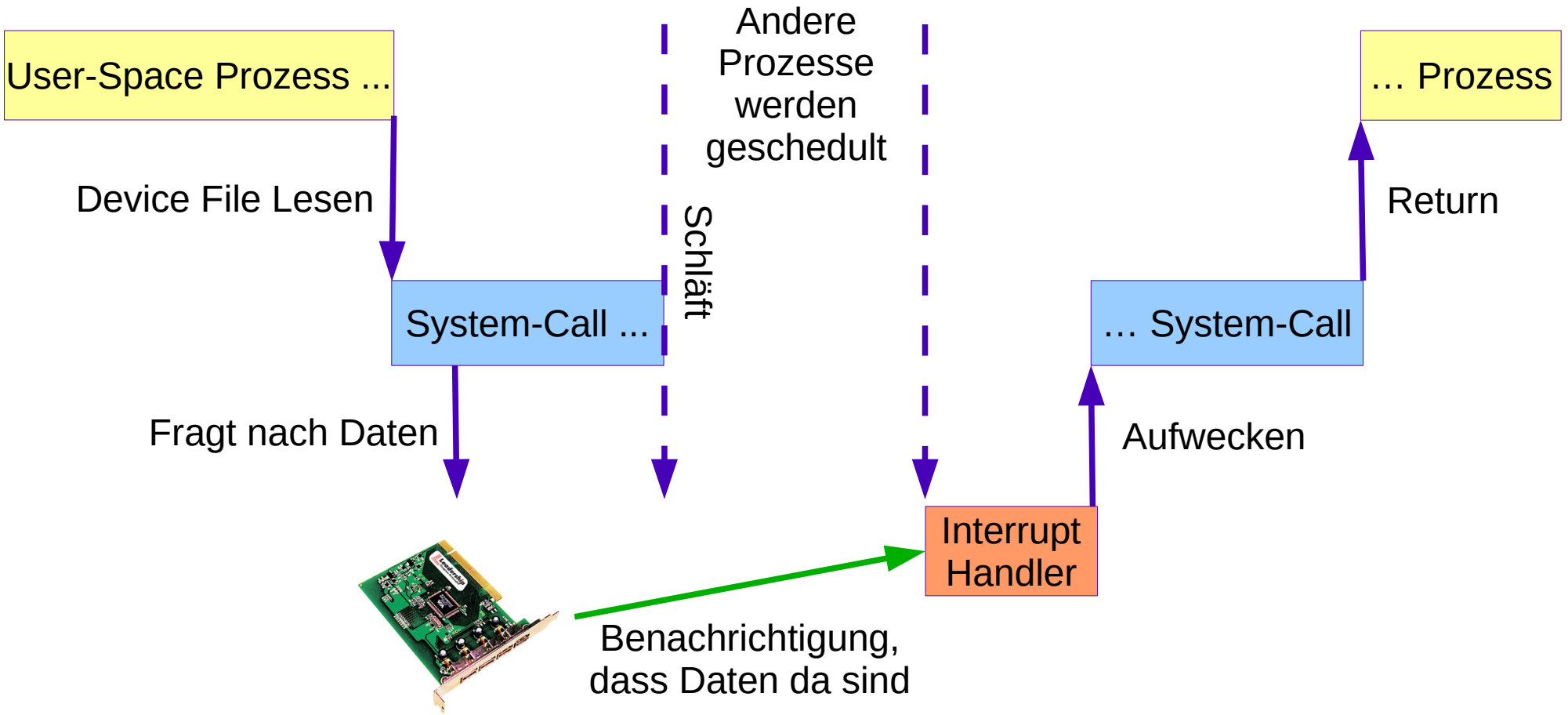
Lebenszyklus eines Threads

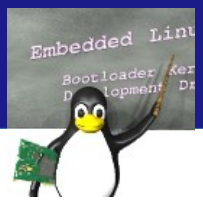




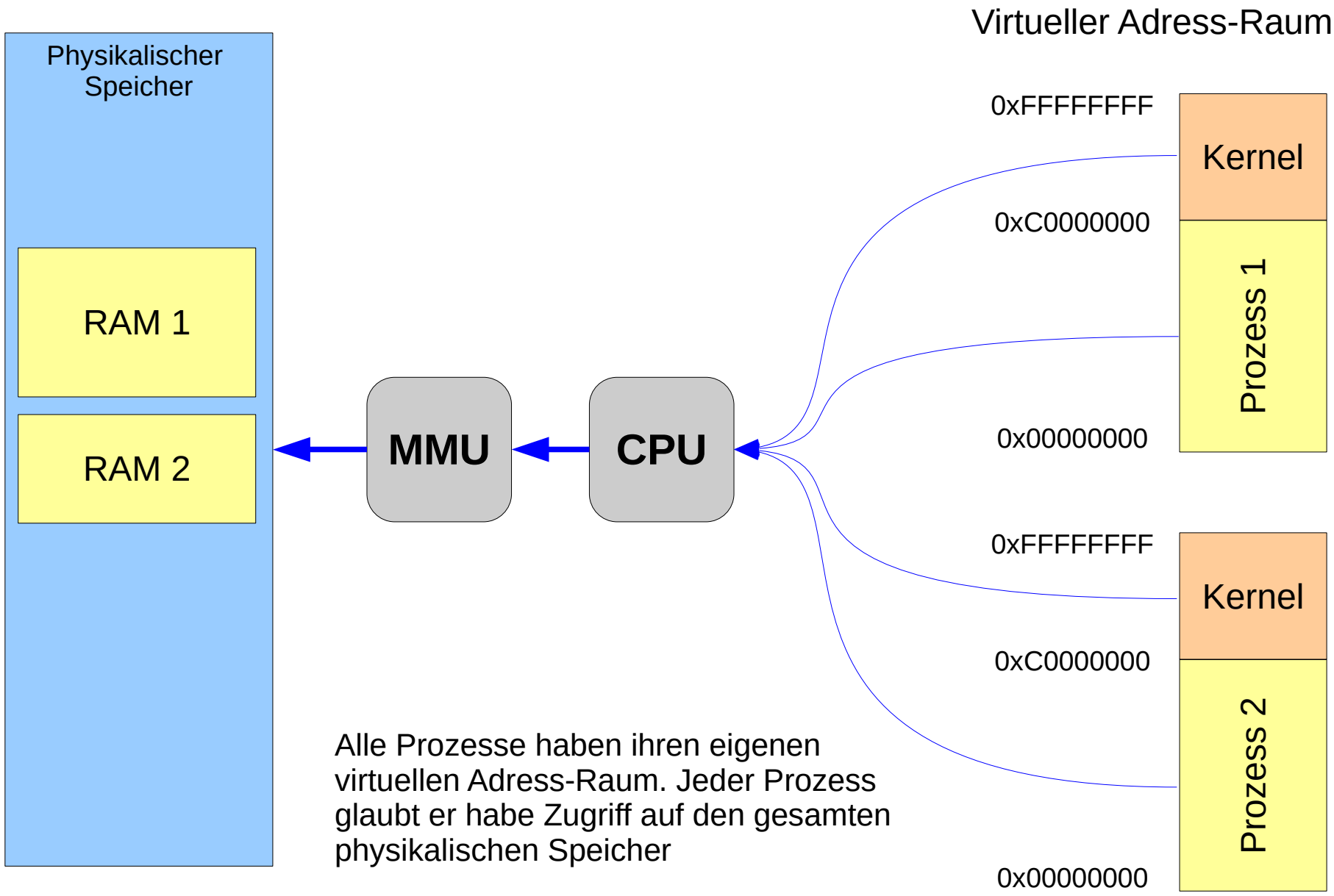
Schlafen

Ein Prozess kann sich schlafen legen, wenn er auf Daten wartet

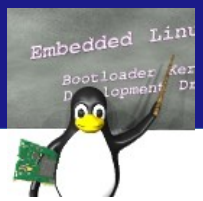




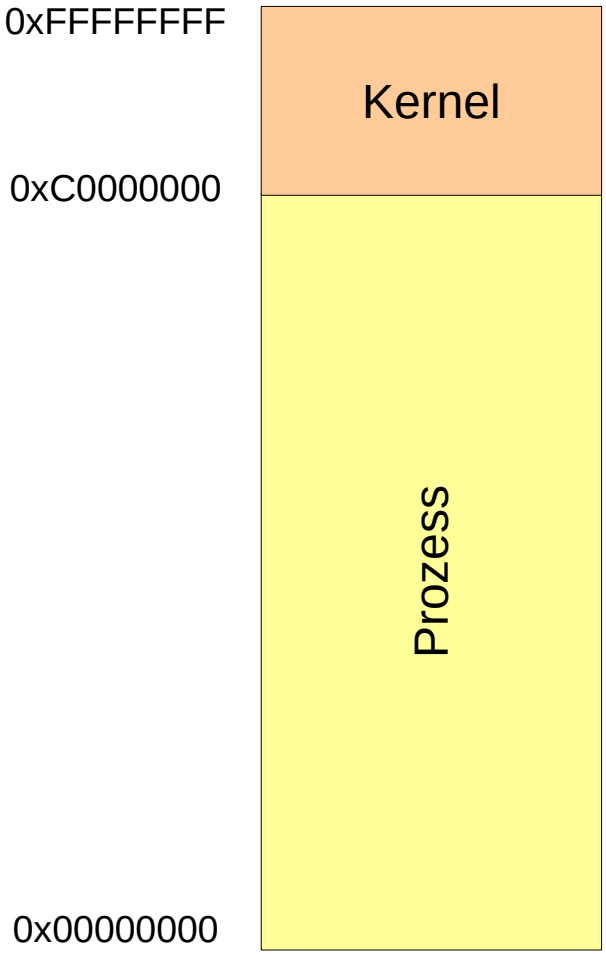
Speicher Verwaltung



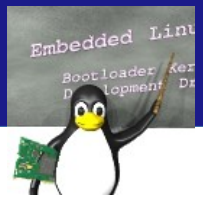
Alle Prozesse haben ihren eigenen virtuellen Adress-Raum. Jeder Prozess glaubt er habe Zugriff auf den gesamten physikalischen Speicher



Speicher Organisation

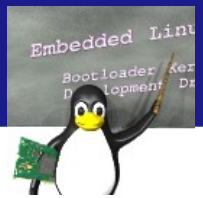


- 1GB für Kernel Space reserviert
- Enthält Kernel Code und Daten Strukturen
- Der meiste Speicher kann direkt auf den physikalischen Speicher an einem festen Offset gemappt werden.
- Jeder User Prozess sieht 3GB Speicher
  - Prozess Code, Daten, Stack, …
  - Memory-mapped Files
- Dieser Speicher ist nicht unbedingt auf den physikalischen Speicher gemappt.
- Speicher wird dynamisch durch Page Fault Exception zugewiesen
  - Kann zu Out of Memory führen



Scheduler

- Der Scheduler verwaltet die Prozesse und weist ihnen CPU Zeit zu
- Linux setzt einen Preemptive Scheduler ein
- Prozesse auf gleicher Prioritätsebene werden Round Robin verwaltet
- Prozesse können von Prozessen höherer Priorität unterbrochen werden
- Linux besitzt zwei Prioritäts Bereiche
  - User Prioritäten oder Nice Level: Werte von -20 bis +19
  - Soft-Realtime Prioritäten



Completly Fair Scheduler CFS

- Verwendet Load Balancing mit einem Rot-Schwarz Baum
- Verwendet Scheduling Klassen
- Die Berechnung ist nur mathematisch nicht heuristisch
- Nano-Sekunden basiertes Accounting unabhängig von HZ oder Jiffies
- Task mit der längsten Wartezeit im Rot-Schwarz Baum wird als nächstes selektiert
- Wenn ein Task in die Runqueue kommt wird ein wait_runtime Wert inkrementiert abhängig von der Anzahl der Prozesse die aktuell in der Runqueue sind und ihren Prioritäten
- Wenn der Task geschedult wird, wird der wait_runtime Wert dekrementiert.