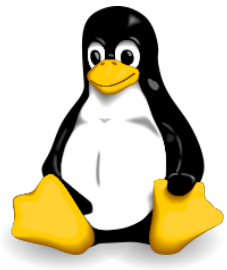
Keine Angst vor Unix

Dipl.-Ing. Martin Weitzel
Technische Beratung für EDV
64380 Roßdorf, Germany
www.tbfe.de

Linux vs. Unix



- Linux ist ein
  - portables
  - skalierbares
  - quell-offenes
- Betriebssystem
- Es basiert auf UNIX,
  - das vor über 30 Jahren entworfen wurde,
  - als Multi-User / Multi-Tasking System,
  - implementiert in der Programmiersprache „C“,
  - für damalige „Mini-Computer“.

Was ist ein „Mini-Computer“?

PDP 7

PDP 11

Vollständige Historische Landkarte

NO SPACE ON DEVICE

Persönliche Historische Landkarte

1978	1. BSTJ-Heft mit Schwerpunkt UNIX der AT&T-Forscher
1982	Erste Erfahrungen mit UnixFLEX (Unix V6 Clone auf 6809 µP)
1984	2. BSTJ-Heft mit Schwerpunkt UNIX der AT&T-Forscher
1986	Schulungen für SINIX (= Unix V7 Derivat auf 8086 µP)
1989	*1. eigener Unix-Rechner (= Unix System III auf 80186 µP)*
1991	*2. eigener Unix-Rechner (= „Interactive Unix“ auf 80386 µP)*
...	*3. eigener Unix-Rechner (= „Unix System V“ auf 80386 µP)*
1995	*1. eigener Linux-Rechner (PC-basiert, 80486 µP)*
1996	Proklamation einer „Microsoft-freien Zone“ ☺
...	*weitere eigene Linux-basierte Desktops / Notebooks*
2009	Embedded Linux Projekt mit Tcl/Tk GUI (ARM)
...	*weitere Linux-basierte Devices (DSL-Router, IP-Cam, RAID File-Server, DVB-S-Receiver ...)*
2010	
...	
2012	Beteiligung an Linux/Qt Projekt im Embedded Umfeld
	Back to the roots: Linux im FPGA-Umfeld

MMU ohne Speicher-Schutz

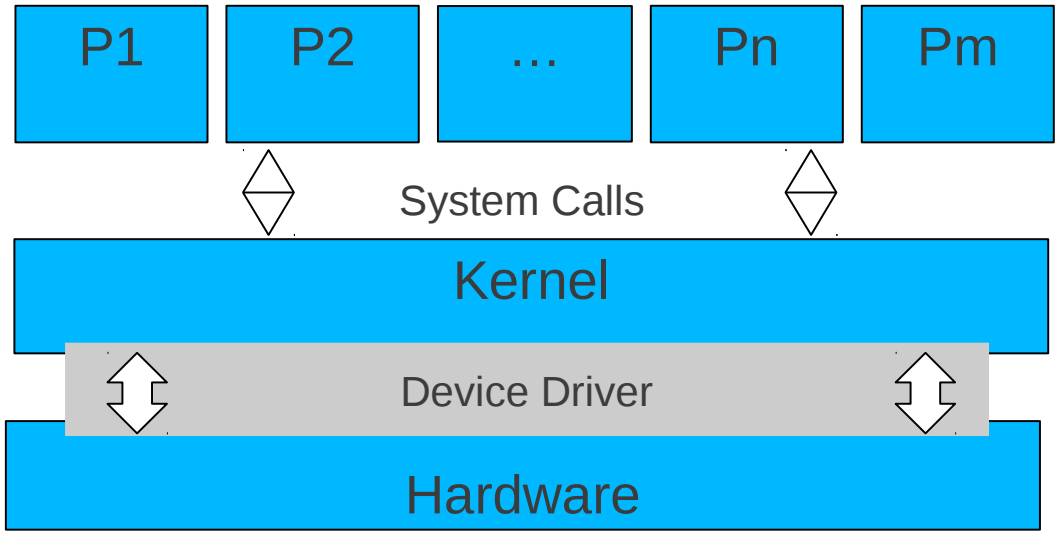
MMU mit Speicher-Schutz

Und los geht‘s …

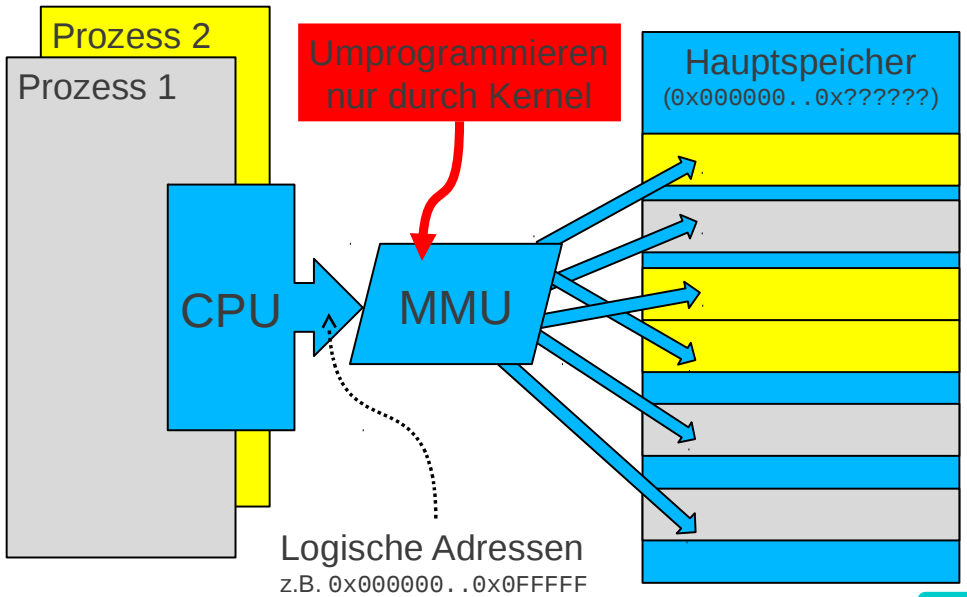
Bitte beachten:

- Die nachfolgende Folien demonstrieren teilweise
  - abstrakte Sachverhalte
  - an konkreten Beispielen
- aber …:

System-Struktur



Memory-Management (MMU)



System-Calls (1)

- Anfänglich weniger als 50
  - Prozess-Steuerung (`fork`, `exec`, `kill`, ...)
  - Dateiverwaltung (`open`, `read`, `write`, `stat`, ...)
  - Sonstiges
    - File-System eingliedern/entfernen (`mount`, `umount`)
    - Treiber konfigurieren (`ioctl`)
    - Echtzeituhr abfragen/stellen (`time`, `stime`)
    - Prozess-Umgebung (`cd`, `getpid`, `getuid`, `setuid`, ...)
    - Profiling / Debugging (`clock`, `ptrace`, `monitor`, ...)

Fortsetzung

System-Calls (2)

Fortsetzung

- Heute mehr (100..150)
  - z.B. Interprozess-Kommunikation
  - Vernetzung (TCP/IP-Sockets)
- Generell der **einzige** Weg, auf dem Prozesse
  - die Dienstleistungen des Kernels in Anspruch nehmen
  - auf Gerätetreiber (und damit die Hardware) zugreifen können.
- Aus Applikationssicht (in C/C++) Bibliotheksfuntionen

System-Calls (MMU)

- Eine **MMU mit Speicherschutz(!)** kann i.d.R. zuverlässig verhindern, dass ein Prozess
  - einen anderen Prozess schädlich beeinflusst …
  - … oder gar das System zum Absturz bringt.
- **Lediglich Fehler in Gerätetreibern können die Stabilität des Systems beeinträchtigen!**

System-Calls (Beispiel-Code)

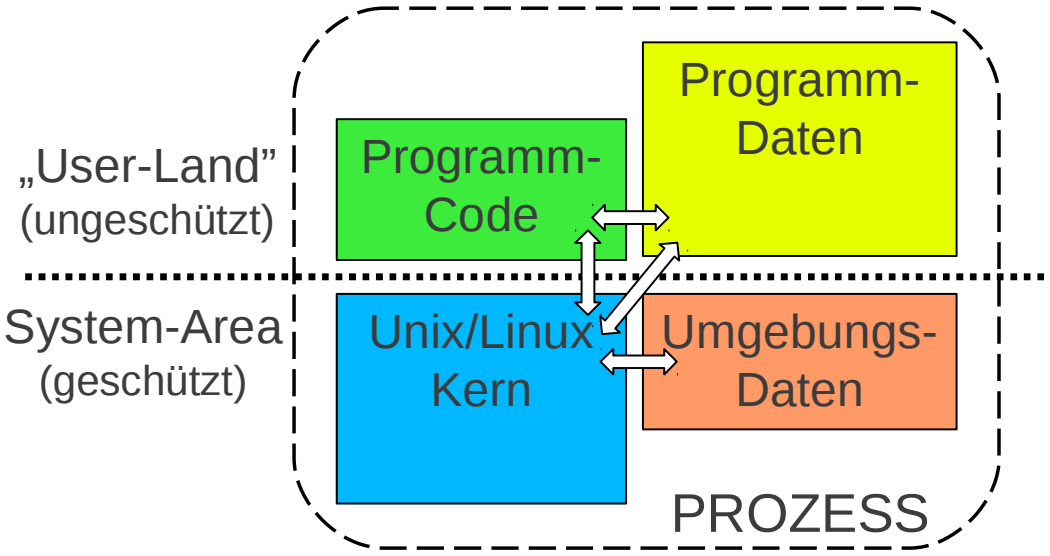
```
/*
 * Datei kopieren (ohne Fehlerbehandlung)
*/
void copyFile(const char *from, const char *to) {
    int fd1= open(from, O_RDONLY);
    int fd2= open(to, O_WRONLY | O_TRUNC);
    char buff[512];
    int count;
    while ((count= read(fd1, buff, 512)) > 0)
        write(fd2, buff, count);
    close(fd1);
    close(fd2);
}
```

System-Calls (Fehlerbehandlung)

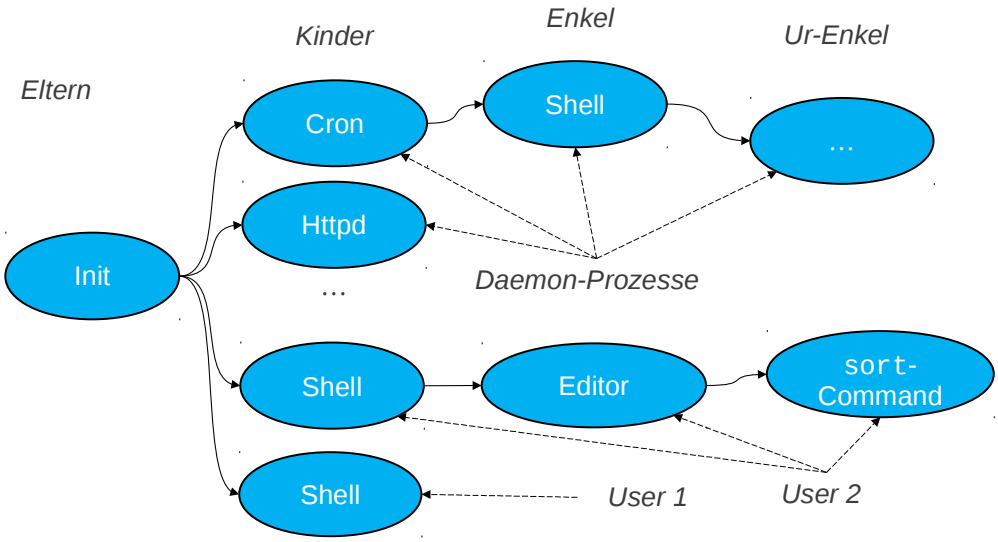
- Fehlerbehandlung i.d.R. „freiwillig“
- Anzeige typisch durch Rückgabewert
  - **OK: `0`** (oder „Nutzdaten“, z.B. File-Descriptor)
  - **Fehler: `-1`**
- Schwerwiegende Fehler generieren „Signale“
  - Abbruch des laufenden Prozesses (Default) oder
  - Einsprung in vom Prozess festgelegte Funktion:

```
void killHandler(int sig) { … }
signal(SIGINT, killHandler);
```

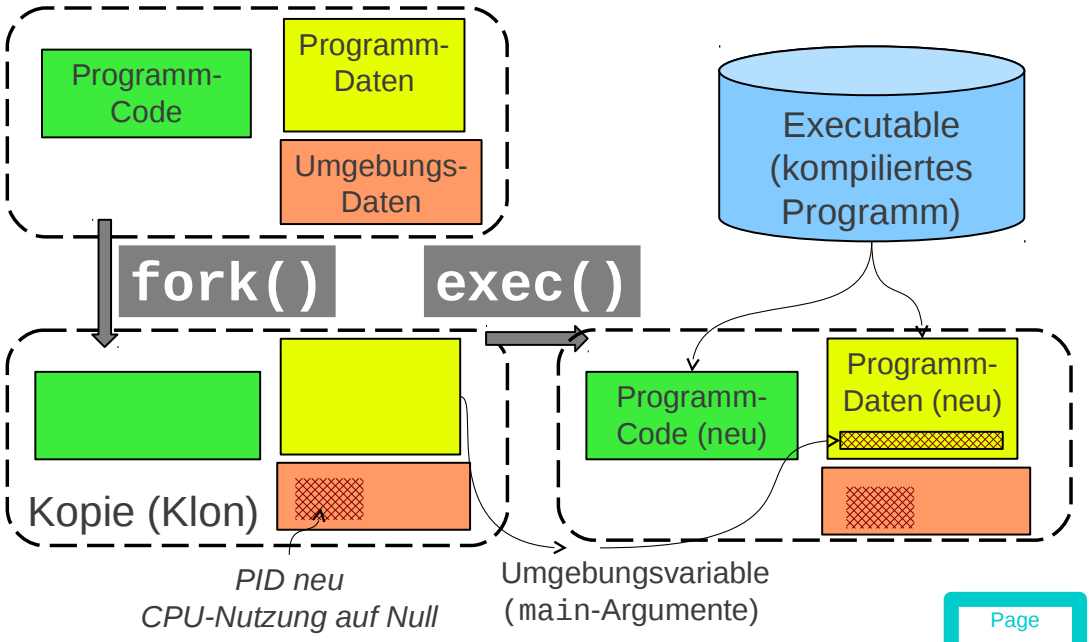

Prozess-Struktur



Prozess-Hierarchie



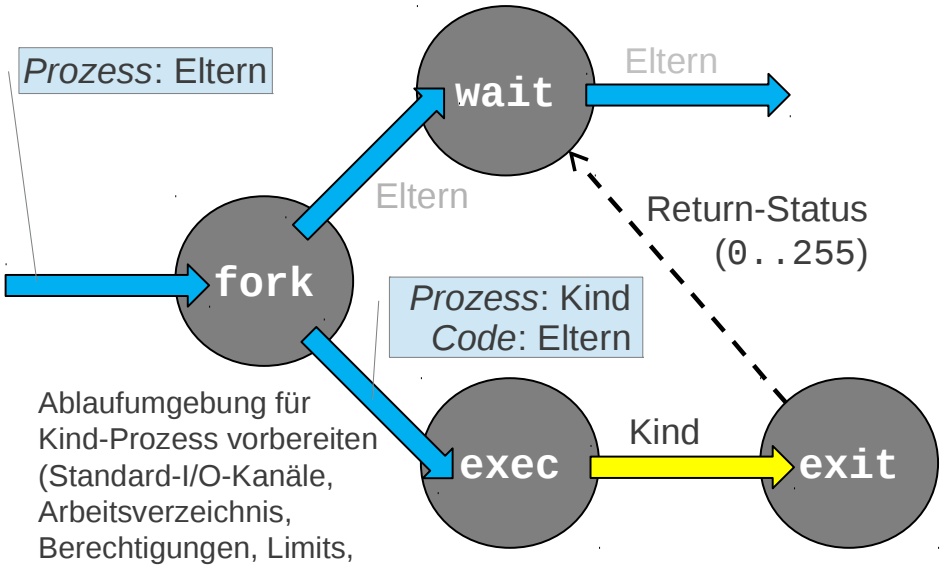
Prozess-Erzeugung



Der „Copy on Write“-Trick

- Benötigt ein neu erzeugter Prozess eine Hauptspeicher-Kopie seines Erzeugers, wird der Kernel
  - die MMU so programmieren, dass die logischen Adressen des neuen Prozesses auf dieselben physikalischen Adressen abgebildet werden, wie die des Erzeugers,
  - anschließend die gemeinsam genutzten Speicherseiten auf „write protected“ setzen,
  - bei einer Veränderung durch einen der beiden Prozesse dementsprechend eine „protection violation“ erhalten
  - und **dann erst die betreffende Seite kopieren**.
- Da zwischen fork und exec im Kindprozess nur relativ wenig an den Programmdaten verändert wird, ist die Prozesserzeugung durchaus effizient.

Warum heißt „fork“ fork?



System-Call (Beispiel-Code)

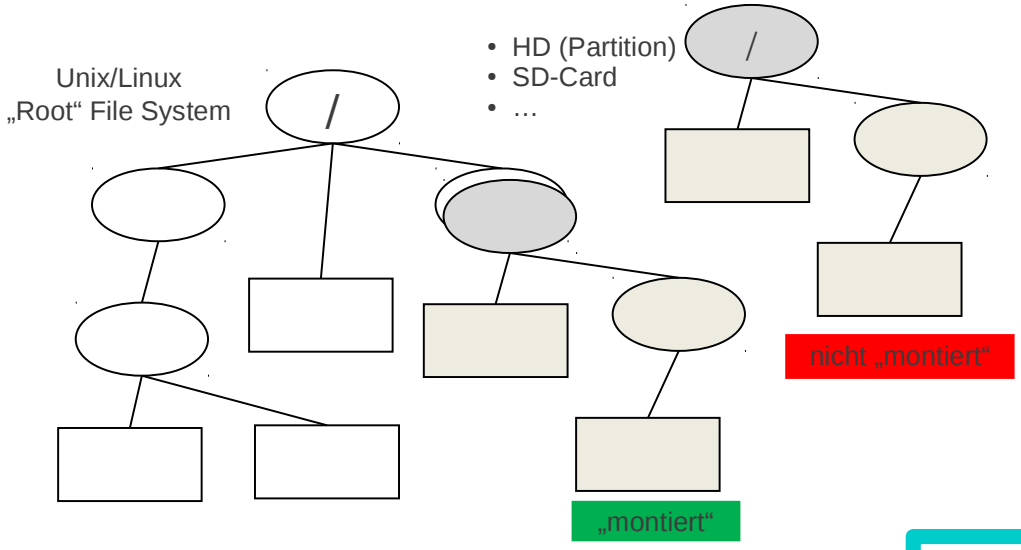
```
/*
 * Neuen Prozess erzeugen (ohne Fehlerbehandlung)
*/
void makeChild(const char *prog) {
    int pid, rstat;
    if ((pid= fork()) > 0} {
        /* Elterprozess */
        wait(&rstat);
    } else {
        /* Kindprozess */

        execl(prog, prog, (const char *)0);
    }
}
```

Bei Bedarf: Ablaufumgebung für Kindprozess modifizieren (z.B. I/O-Umlenkung)

Betriebssystem-Kommandos

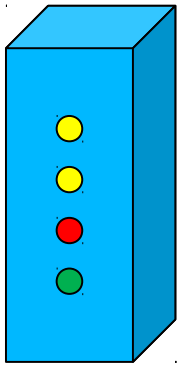
- Klassisches Unix:
  - Jedes „Kommando“ ist ein eigenes Programm …
  - … und wird von „Shell“ mit fork/exec gestartet
  - Overhead (wegen „Copy on Write“-Trick)
    - weniger beim Zeitbedarf sondern
    - mehr beim Platzbedarf der „Executables“
- Eventuell sinnvoll bei Linux auf „Embedded Devices“:
  - alle „wichtigen“ Kommandos in ein einziges „Executable“ und
  - dieses unter verschiedenen (Verweis-) Namen bereitstellen
  - Realisiert durch „Busybox“.
  - **⇒Spart vor allem Speicherplatz im File-System**
- Ebenso „Sparversion“ anderer Standardprogramme sinnvoll:
  - z.B. ssh/scp-Implementierung „dropbear“.

Datei-Systeme



Fiktives Gerätetreiber-Beispiel

- LED-Steuerung über Textausgabe
  - Zuordnung: `A..D` von oben nach unten
  - Codes: `0` = aus, `1` = ein, `2` = blinken



```
# Programmierbeispiel Shell
…
echo –n A0B1C2D1 >/dev/keyleds
```

```
/* Programmierbeispiel C */
…
int fd= open("/dev/keyleds", O_WRONLY);
write(fd, "A0B1C2D1", 8);
close(fd);
```

Berechtigungssystem

- Entworfen als „Multi-User / Multi-Tasking“-System
  - kann Unix nicht nur mehrere parallele Prozesse ausführen,
  - sondern dabei auch unterschiedliche Privilegien dieser Prozesse berücksichtigen (bzw. der Personen, die den jeweiligen Prozess gestartet haben).
- Beim Ablauf von Linux auf „Embedded Devices
  - mag dieser Punkt weniger wichtig erscheinen,
  - könnte aber beispielsweise für unterschiedlich privilegierte Service-Zugänge über telnet/ssh benutzt werden.
- Sehr sparsame Implementierung
  - **kaum Overhead wenn eine Embedded-Linux Device kein Berechtigungssystem erfordert**

UID und GID

- Beruht auf
  - User ID (eindeutige numerische Kennung eines Nutzers) und
  - Group ID (eindeutige numerische Kennung der Gruppenzugehörigkeit).
  - Mit der UID=0 sind besondere Privilegien verbunden (= System-Administrator, „Super-User“, „Root“ …).
- Prozesse
  - besitzen die UID und GID ihres Erzeugers,
  - vererben diese an erzeugte Prozesse und
  - können (nur dann) ihre UID/GID verändern, wenn sie privilegiert sind.
- Dateien
  - bekommen die UID und GID des Prozesses, der sie angelegt hat.
  - Privilegierte Prozesse können die UID /GID einer Datei beliebig ändern.
  - Evtl. können Eigentümer ihre Dateien „verschenken“ (systemabhängig).

Zugriffsrechte: read, write, execute

- „Normale Dateien“ und „Devices“:
  - **r**: Inhalt kann gelesen werden
  - **w**: Inhalt kann geschrieben/verändert/erweitert werden
  - **x**: kann als Programm ausgeführt werden
  - Kompilierte Programme („Executables“) sind auch ohne zusätzliches **r** ausführbar - sie können dann aber nicht kopiert werden
  - Bei Skripten ist **r** notwendig und ausreichend
  - und natürlich **x** für das Interpreter-Programm (z.B. `tclsh`)
- Verzeichnisse
  - **r**: Enthaltene Files können aufgelistet werden.
  - **w**: Neue Files können angelegt / vorhandene gelöscht werden
  - **x**: Verzeichnis-Inhalt (und Unterverzeichnisse) ist zugänglich
- **Besondere Rechte für Executables: SUID / SGID:**
  - Während der Ausführung gelten **nicht** die Rechte des Eltern-Prozesses
  - sondern die des Eigentümers bzw. der Gruppe dieser Datei

Entscheidungsfindung und Darstellung

- Der Zugriff auf eine Datei geht immer von einem Prozess aus:
  - (Prozess UID == Datei UID) ? ⇒ Eigentümer (user)
  - (Prozess GID == Datei GID) ? ⇒ Gruppenmitglied (group)
  - Ansonsten ⇒ Rest der Welt (other)
- 3 Rechte (`r`, `w`, `x`) und 3 Kategorien (u, g, o):
  - ⇒ 9 Kombinationen
  - Symbolische Darstellung z.B. `rwxr-xr-x` oder `rw-r----`
  - Alternativ Oktalnotation: `755` oder `640`
    - Entspricht interner Repräsentation der Berechtigungen
    - SUID oktal `4000`, SGID oktal `2000` (Bit-Oder bzw. Addieren)

Unix/Linux-Vernetzung

- Seit langer Zeit „Standard“ unter Unix
  - Klassische Umsetzung
    - Remote-Login: telnet, rsh
    - Dateitransfer: ftp, rcp
  - Mittlerweile besser (= mehr Sicherheit)
    - Remote-Login: ssh
    - Dateitransfer: scp, sftp
- Netzwerk-Dateisysteme
  - NFS (Network-Filesystem)
    - Entwickelt und erstmals implementiert von SUN (~ 1985)
    - Unterstützung im Kernel oder durch externen Server/Client
    - Lokales Eingliedern exportierter Verzeichnisse durch „Montieren“
  - SAMBA (Windows-Welt)

GUI-Programmierung

- Seit langer Zeit Standard unter Unix
  - Window-System heißt „X11“
    - Ursprung Mitte der 1980er
    - Design-Konzept: Netzwerkfähigkeit
  - Breite Hardware-Unterstützung
- Programmierschnittstelle
  - Direkte Nutzung der „Xlib“ aufwändig und umständich
  - Besser „fortschrittlichere C++-Libraries“: z.B.
    - „Qt“ (kde – Konzept lose gekoppelter Komponenten)
    - Gnome/GTK (quell-offen und kostenfrei)
  - Oder Nutzung aus Skriptsprachen (Tcl/Tk, Python, Ruby …)
  - Weitere Alternative: Java (AWT, SWT …)
- **OS-Unabhängigkeit leicht realisierbar!**

Skript-Sprachen

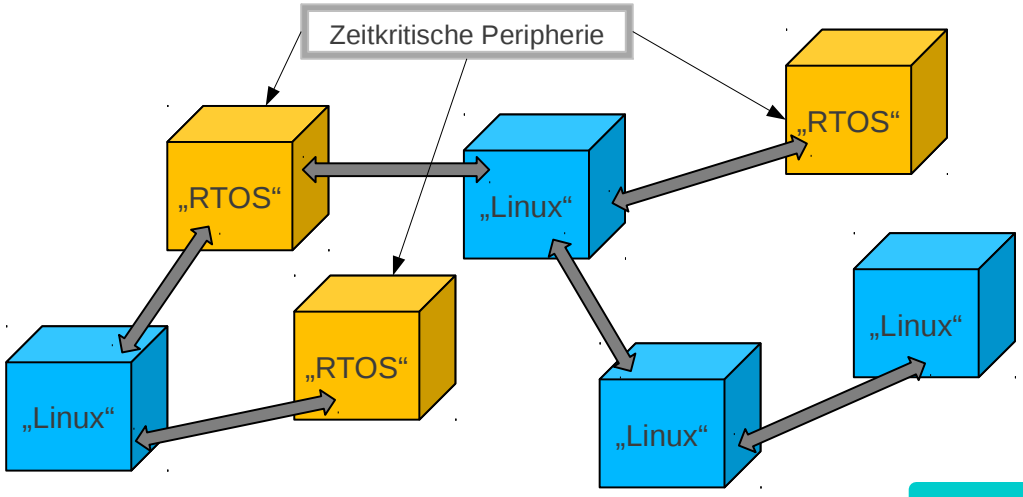
- Klassiker: Tcl/Tk
  - Shell-ähnlich – aber „aufgeräumter“ …
  - … dennoch nicht jedermanns Sache
  - Objektorientierte Variante verfügbar: itcl
  - Speziell für Embedded Targets: Evolane Tcl-Engine
- Moderner: Python, Ruby …:
  - Unterstützt „Prozedurale Programmierung“
  - Unterstützt „Objekt-Orientierte Programmierung“
  - Unterstützt „Funktionale Programmierung“
- In allen Fällen gilt:
  - Zusammenspiel mit C/C++-Modulen einfach realisierbar
  - ⇒ SWIG (Software Wrapper Interface Generator)

Unix/Linux und die Echtzeit

- Unix wurde nicht unter dem Aspekt der „Echtzeit-Fähigkeit“ entworfen:
  - Es ist kein „träges“ System (ganz im Gegenteil) …
  - ...aber Spanne zwischen durchschnittlichen und
  - schlechtesten Fall ist mitunter groß
- Linux ist in dieser Hinsicht sehr ähnlich zu Unix ...
  - Reaktionszeiten sind im Allgemeinen nicht einfach absolut verlässlich kalkulierbar, ausgenommen
    - Gerätetreibern
    - speziell modifizierte Varianten
- **… wenngleich sich gerade in diesem Bereich in den letzten Jahren einiges getan hat!**

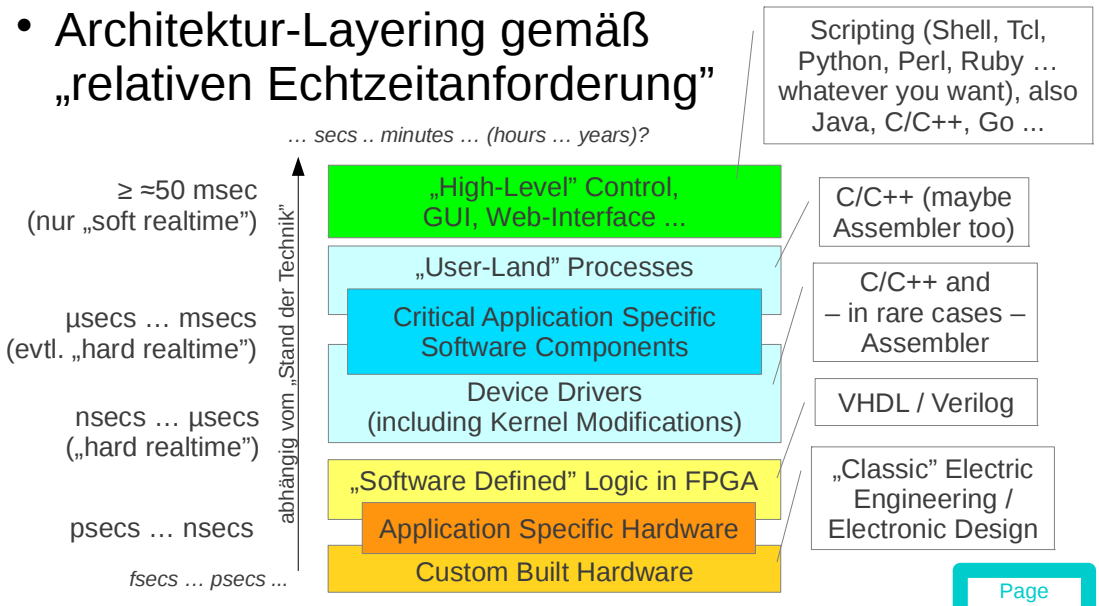
Möglicher Ausweg (1)

- Vernetzte „Nodes“ unterschiedlicher Systeme



Möglicher Ausweg (2)

- Architektur-Layering gemäß „relativen Echtzeitanforderung”



Zusammenfassung (1)

- Unix war das Vorbild für Linux
- Es mag ein „altes“ Betriebssystem sein, aber …
- … die Übernahme seiner Struktur macht Linux
  - skalierbar, übersichtlich und
  - für viele Aufgaben geeignet
- Hardware-Voraussetzung ist eine MMU
  - Speicherbereiche verschiedener Prozesse werden damit gegeneinander abgeschottet ⇒ **Sicherheit!**
  - gilt **nicht** für „µC-Linux“-Variante ⇒ **weniger sicher**
- Echtzeit ist keine „typische“ Unix-Domäne
  - aber realisierbar in Gerätetreibern
  - und speziellen Linux-Varianten

Zusammenfassung (2)

- Zwei der zentralen Abstraktionen sind
  - „Prozess“ und
  - „Datei“.
- Prozesse bilden eine Baumstruktur
  - Jeder Prozess hat einen „Erzeuger“ ...
  - … außer dem mit Ende des Boot-Vorgangs gestarteten “Initializers”
- Das Dateisystem ist hierarchisch strukturiert
  - Externe Datenträger werden transparent „montiert“
  - Die Kommunikation mit Gerätetreibern erfolgt oft über sogenannte „Device Files“ (`/dev/…`)

Zusammenfassung (3)

- Die meisten Betriebssystem-Kommandos
  - sind eigenständige kleine Programme und
  - evtl. zusammengefasst in der „Busybox“.
- Die Programmiersprache C/C++ wird immer unterstützt:
  - Zugriff auf Dienstleistungen des Kernels erfolgt ausschließlich durch C-Funktionsaufrufe;
  - der Kernel ist selbst größtenteils in C geschrieben.
- Generell ist auch „Skript-Programmierung möglich:
  - Es gibt diverse „Shell“-Programme …
  - … und viele Skript-Sprachen werden unterstützt.
- Vernetzung und GUI sind eine Selbstverständlichkeit!

Epilog

- Vielleicht hatten die Unix-Entwickler doch eine Ahnung, wie langlebig ihr System sein würde.
- Für die Systemzeit verwendeten sie nämlich eine 32-Bit-Variable – `signed long` (damals).
- Damit zählen sie die Sekunden beginnend mit dem *1. Januar 1970, 0 Uhr GMT*
- Eine vorzeichenbehaftete 32-Bit Variable läuft bekanntlich bei dem Wert *2.147.483.647* über.
- Das entspricht welchem Datum: _____?

Das war‘s

Noch Fragen?

Danke für Ihre Aufmerksamkeit