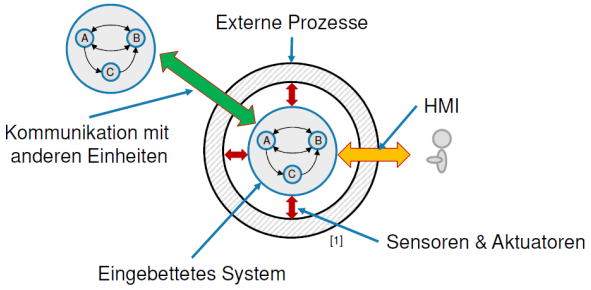
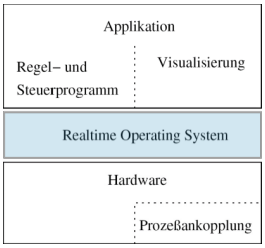
Embedded Systems
Einführung

Eingebettetes System



- integrierte, elektronische Schaltung mit spezifischer Aufgabe, mit der ein Benutzer nur indirekt in Verbindung kommt
- Teil eines Gesamtsystems mit stark beschränkten Ressourcen, bestehend aus Hard- und Software (teilweise ohne Betriebssystem)
- HW/SW-Codedesign z.B. mit *VHDL*, allgemein *tool*- bzw. *modellbasierter* Entwurf
- *75 %* verwenden ein Betriebssystem (Tendenz steigend)
  - *25 %* mit *Main-Loop*

- Ist eine Kombination aus Hard- und Softwarekomponenten, die in einen technischen Kontext zur *Steuerung*, *Regelung* und *Überwachung* eines Systems eingebunden sind
- Es verrichtet vordefinierte Aufgaben, oftmals mit *Echtzeitberechnungs*-Anforderungen
- **Speicherprogrammierbare Steuerung** (SPS): Verwendet zur Fabrikautomatisierung, Verkehrsleitung
- **Standardarchitektur** auf einem PC: Preiswerte Hardware (allerdings oft nicht *industrietauglich*) preiswerte Software, häufig ohne *Echtzeitfaehigkeit*
- **Industrie-PC**: Unterstützt Echtzeitbetriebssysteme



Definition: *Technischer Prozess*

- Prozess, in dem Zustandsgrößen durch *technische* Hilfsmittel festgestellt und beeinflusst werden
  - *Prozess* definiert als Gesamtheit von aufeinander einwirkenden Vorgängen in einem System, durch die Information verändert wird
- *Sensoren* (z.B. Thermometer, Kamera, Mikrofon) erfassen Zustandsgrößen, *Aktoren* (z.B. Motoren, Relais, Ventile) beeinflussen sie

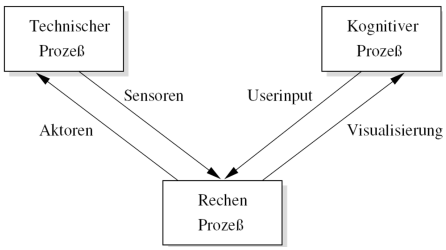
Klassifikation: *Technischer Prozess*

- **Fließprozess (Regler):** physikalische Größe mit stückweise kontinuierlichem Wertebereich, ablaufende Vorgänge sind zeit- und ortsabhängig, z.B. chemische Reaktoren, Energieerzeugung in Kraftwerken
- **Folgeprozess (State Machine):** Binäre, diskrete Informationselemente werden gemeldet oder ausgelöst, z.B. Ampel- oder Aufzugsteuerung
- **Stückprozess (Datenbank):** Informationselemente werden einzeln identifizierbaren Objekten (Stücken) zugeordnet z.B. Transport- oder Ladevorgänge, Fertigung

Definition: *Rechenprozess*

- *Task* als Instanz zur dynamischen Abarbeitung eines Programms zur Berechnung von Ausgabewerten aus Eingabewerten über Umformen, Transportieren oder Speichern von Information

Definition: *Kognitiver Prozess*



- Umformen, transportieren oder verarbeiten von Information im menschlichen Bediener
- Einflussnahme des Bedieners auf den Rechenprozess über *Man Machine Interface* (MMI)

Definition: *Steuerungssystem*

- Umfasst zur Steuerung erforderliche Rechenprozesse sowie deren Hard- bzw. Software
- Aufgaben:
  - Erfassen von Zustandsgrößen
  - Koordinaten & Überwachung der Prozessabläufe

Definition: *Steuerung* und *Regelung*

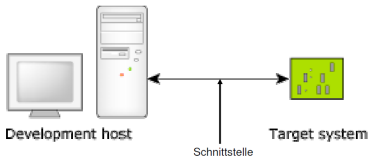
- **Steuerung:** Kein geschlossener *Regelkreis*, Rechenprozess reagiert nicht auf sich ändernde Sensorwerte im technischen Prozess
- **Regelung:** Geschlossener *Regelkreis*, Sensor- bzw. Messwerte werden verwendet, um Stellgrößen daraus zu berechnen

Self-Hosted-Entwicklung

- Entwicklungsumgebung und Zielsystem sind identisch (*so wie wir es alle kennen*)

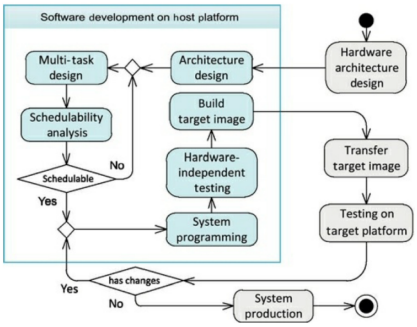
Host-Target-Entwicklung

- *Self-Hosted-Entwicklung* oft nicht möglich da Hardware proprietär oder zu leistungsschwach für Entwicklungsumgebung



- **Host:** Entwicklungsrechner, enthält *Cross-Compiler*, *Remote-Debugger*, *Target*-Libraries und -Betriebssystem
  - *Cross-Compiler:* Erzeugt *Image*, dass eigentliche Applikation sowie Betriebssystem- und Laufzeitkomponenten + *Startupcode* enthält
  - *Remote-Debugger:* Auf dem Host läuft GUI mit *Debug-Info*, über *JTAG* etc. sieht man den Systemzustand des Targets (*Stack*, *Variablen*belegung) an gewählten *Breakpoints*
  - *Ohne* Remote-Debugger: Konsolenausgaben per *printf*, *LED*s blinken lassen
- **Schnittstelle:** Zum *Downloaden* der Applikation auf das *Target* oder fürs *Debugging*, verschiedenste Variationen möglich (z.B. *Ethernet*, *USB*, *JTAG*, *Flash*, …)
- **Target:** System, für das entwickelt wird
  - *Boot-Monitor:* Programm auf dem Target, über das Software geladen und gestartet werden kann, erfolgt über ähnliche Schnittstellen wie die *Host-Target*-Entwicklung an sich

Softwareentwicklung in einem *Host-Target* System



- **Object File:** Symboltabelle
- **Linker:** Symbol-Auflösung und *Relocation*
- **Executable File:** Code & Daten zur Ausführung, Umsetzung auf virtuellen Speicher
- **Shared object file:** Code und Daten zum (dynamischen) linken mit anderen *object files*
- **Relocatable file:** Code und Daten zum linken mit anderen *object files* um Executable zu erstellen
- **Dynamic Linker:** Laden von *shared* Libraries

Tools: *Kernel-Tracer*

- Zeigt *Signale*, *Task-Zustaende*, *Semaphoren*, *Interrupts*

Tools: *Stack-Monitor*

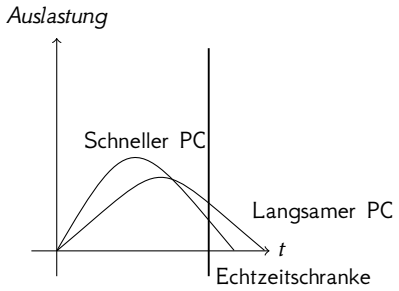
- Zeigt maximal verfügbarer *Stack* pro *Task*, aktuelle Auslastung und maximale je gemessene Auslastung (*Hochwassermarke*) des Stacks

Weitere Tools

- Anzeige der Speicherbelegung und *Auslastung* der *CPU*, *Memory-Leak-Detection*, *Code-Coverage*

Echtzeitbetrieb

Kriterien für Echtzeitsysteme



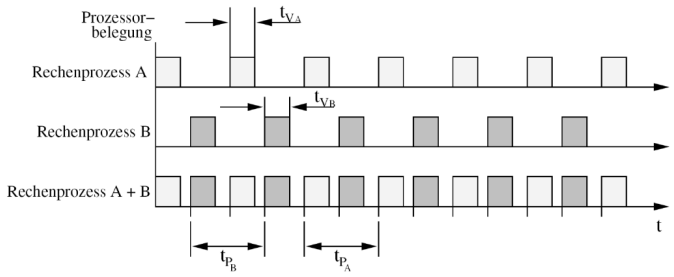
- Schnelligkeit bzw. Geschwindigkeit ist **nicht** wichtig im Kontext einer harten Echtzeitschranke ein schnellerer PC ist zwar häufiger vor der Schranke fertig, aber eben auch nicht zu 100%
- Wichtig dagegen sind:
  - Pünktlichkeit (*Ober- und Untergrenze*) bzw. Rechtzeitigkeit (*Nur Obergrenze*) (*timeliness*)
  - Verfügbarkeit
  - Determinismus (bei gleicher Eingabe im gleichen Zustand liefert das System immer die gleiche Ausgabe)
- Verletzungen von Zeitbedingungen ggf. katastrophal (fristgerechte Bearbeitung von Anforderungen aus einem technischen Prozess)

Vorbedingungen für Echtzeitbetrieb

- Verarbeitungszeit von Aufgaben berücksichtigen, bei mehreren Aufgaben Reihenfolge der Abarbeitung planen
- Reihenfolge entscheidend für fristgerechte Ergebnisse
- Priorität von Aufgaben gemäß ihrer Wichtigkeit als Planungsgrundlage
- Unterbrechung einer Aufgabe muss durch einen Prozess zur Bearbeitung höher-priorer Aufgaben möglich sein

  $\Rightarrow$ Formaler Rahmen zum Nachweis schritthaltender Verarbeitung

Echtzeitbedingung: Auslastung



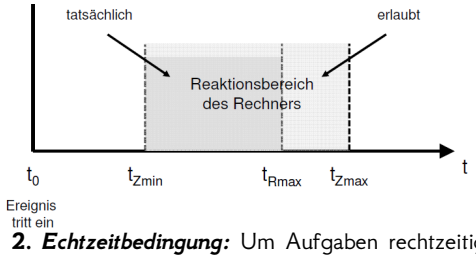
- t_V := Verarbeitungszeit
- t_P := Prozesszeit, Abstand zwischen zwei Anforderungen (*Jobs*) desselben Typs, wenn t_P konstant handelt es sich um einen *zyklischen* bzw. *periodischen* Prozess
- $\rho = \frac{t_V}{t_P}$:= Auslastung
  - $\rho_A = \frac{t_{VA}}{t_{PA}}$
  - $\rho_B = \frac{t_{VB}}{t_{PB}}$
  - $\rho_{A+B} = \frac{t_{VA}}{t_{PA}} + \frac{t_{VB}}{t_{PB}}$
- $\rho = \sum_{i=0}^{n} \frac{t_V i}{t_P i}$:= Gesamtauslastung bei n Prozessen

- **1. *Echtzeitbedingung:*** Gesamtauslastung aller Prozesse $\leq 1 \Leftrightarrow \rho = \sum_{i=0}^{n} \frac{t_V i}{t_P i} \leq 1$

Art von Rechenprozessen

- *zyklisch:* konstanter Abstand zwischen zwei Anforderungen
- *azyklisch:* Keine Untergrenze zwischen zwei Nachrichten, kommen beliebig (*gefaehrlich*)
- *sporadisch:* ähnlich wie *azyklisch* aber mit Untergrenze zwischen zwei Nachrichten (z.B. *Netzwerktreiber*)

Echtzeitbedingung: Pünktlichkeit

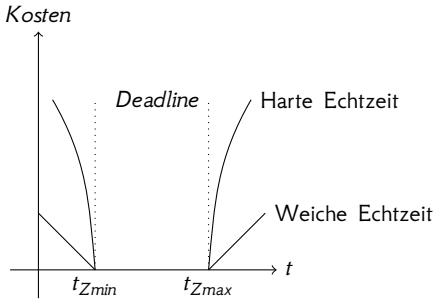


- Aufgabe darf nicht vor spezifizierten Zeitpunkt t_{Zmin} erledigt sein(meist unwichtig oder *trivial*)
- Aufgabe muss spätestens bis Zeitpunkt t_{Zmax} erledigt sein (Rechtzeitigkeit)
- Verbleibende Reaktionszeit: $t_R = t_V + t_W$ (Verarbeitungszeit + Wartezeit)

  Wartezeit := Zeit, bis Rechenkern frei ist

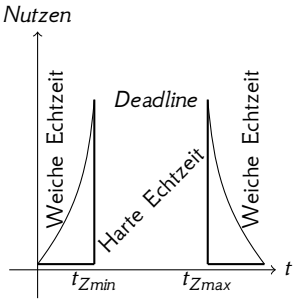
- **2. *Echtzeitbedingung:*** Um Aufgaben rechtzeitig zu erledigen, muss die Reaktionszeit zwischen der minimal und maximal zulässigen Reaktionszeit liegen: $t_{Zmin} \leq t_{Rmin} \leq t_R \leq t_{Rmax} \leq t_{Zmax}$

Harte und weiche Echtzeit



- *Harte* Echtzeit: Verletzung der Rechtzeitigkeit hat *katastrophale* Folgen (z.B. Airbag, Herzschrittmacher)
- *Weiche* Echtzeit: Schlechteres Ergebnis (z.B. *ruckelnde* Videowiedergabe, GPS-Latenz) $\Rightarrow$ Häufig *Graubereich*
- *Kostenfunktion:* Kosten *explodieren* bei Überschreitung der Echtzeitschranke (*Deadline*) im Falle von Harter Echtzeit, bei Weicher Echtzeit steigen Kosten nur *linear* an

- *Nutzenfunktion:* Ergebnisse außerhalb des Intervalls $[t_{Zmin}, t_{Zmax}]$ haben bei Harter Echtzeit nahezu *keinen* Nutzen mehr, bei Weicher Echtzeit ungefähr *lineare* Ab- bzw. Zunahme

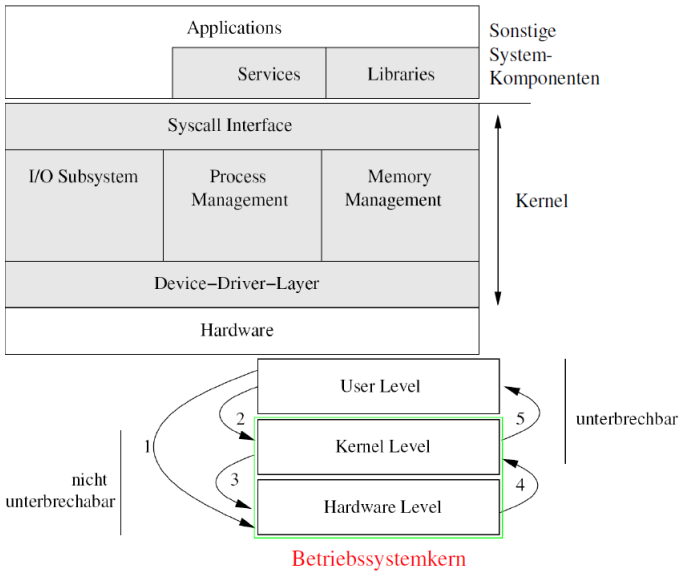


Echtzeitbetriebssysteme

Aufgaben und Anforderungen

- Steuern und Überwachen: Ausführung der Benutzerprogramme & Verteilung der Betriebsmittel (Speicher, Prozessor, Dateien)
- Stellt dem Benutzer die Sicht einer einfacher als die Hardware zu bedienenden *virtuellen Maschine* zur Verfügung
  - Aus Sicht des Benutzers steht der Rechner ihm allein zur Verfügung
  - Einfacher, standardisierter Zugriff auf *Ressourcen* (Speicher, Geräte, Dateien per Gerätetreiber, Dateisystem, Speichermanagement)
- **Zeitverhalten**
  - Schnelligkeit (bei einem *RTOS* insbesondere Realisierung kurzer Antwortzeiten)
  - Zeitlicher Determinismus (Speicherverwaltung und Garbage Collection sind problematisch)
    - Scheduling, IPC und Synchronisation
    - Angabe und Einhalten von Zeitbedingungen, Bereitstellen von *Zeitdiensten*
- **Geringer Ressourcenverbrauch**
  - Hauptspeicher & Prozessorzeit
- **Zuverlässigkeit & Stabilität**
  - Programmfehler dürfen Betriebssystem und andere Programme *nicht* beeinflussen
  - Linux: Treiber & Kernelmodule laufen im *Kernel*-Adressraum
  - QNX: Mikrokernel-Architektur: sogar Treiber haben *eigenen* Adressraum
- **Sicherheit**
  - Datei- und Zugangsschutz
- **Portabilität, Flexibilität und Kompatibilität von Systemkomponenten**
  - Erweiterbarkeit, Einhalten von Standard (z.B. *POSIX*)
  - Möglichkeit für andere Betriebssysteme, geschriebene Programme zu portieren (anpassen, übersetzen, ausführen)
- **Skalierbarkeit**
  - Hinzunehmen oder Weglassen von Betriebssystem-Komponenten möglich machen
  - Geringer Programm- und Datenspeicherbedarf bei kleinen Anwendungen (*Footprint*)
  - Komfort und umfassende Funktionalität bei großen Anwendungen

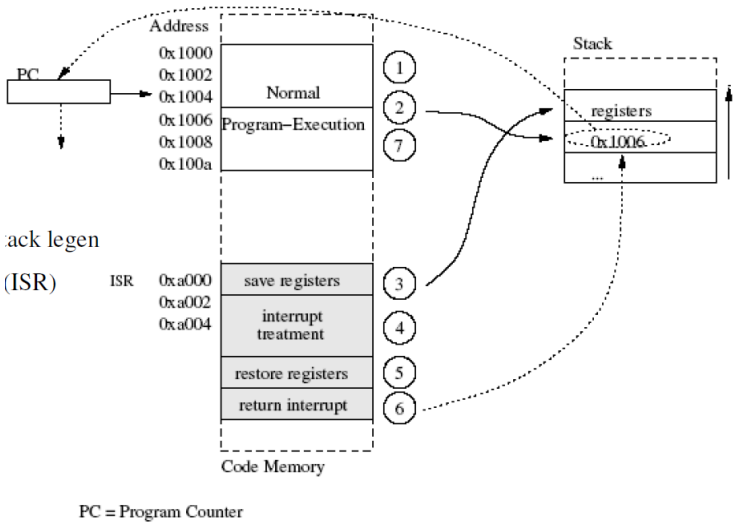
Aufbau und Struktur



- Ein *Betriebssystem* besteht aus aufbauenden *Systemkomponenten* (Dienstprogramme, Werkzeuge) und einem *Betriebssystemkern*
- (1): Hardware-*Interrupt*
- (2): Software-Interrupt (*Systemcall*)
- (3): Hardware-Interrupt (während eines Systemcalls)
- (4): Hardware-Interrupt (*Scheduler* wird aufgerufen)
- (5): Scheduler übergibt CPU einem Task auf *User*-Ebene
- Betriebssystem-Dienste werden fast bei jedem Betriebssystem über Software-Interrupts (*Supervisor Call* / Systemcall) angefordert

Prozessmanagement

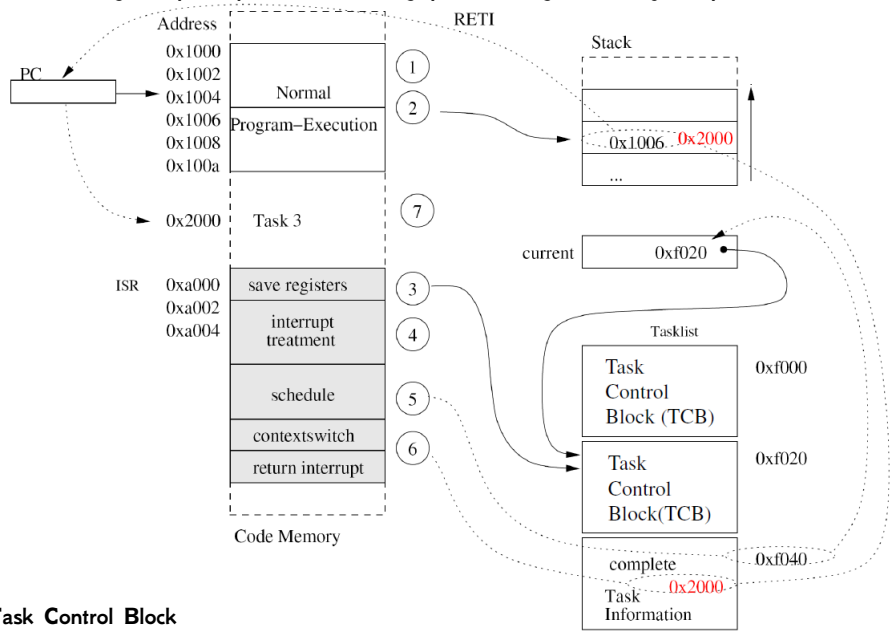
Unterbrechung *ohne* Betriebssystem



PC = Program Counter

- (1): CPU arbeitet Programm ab
- (2): Interrupt während der Programmabarbeitung:
  - Abarbeitung des aktuellen Befehls beenden
  - Befehlszähler und Registerinhalte auf den Stack legen
  - Befehlszähler auf *Interrupt Service Routine* (ISR) legen
- (3): *ISR* rettet von ihr benötigte CPU-Register
- (4): Eigentliche Interrupt-Behandlung
- (5): Gerettete CPU-Register wiederherstellen
- (6): Befehl *Return from Interrupt:*
  - Auf Stack abgelegte Register (*Flags*, *Befehlszaehler*) wiederherstellen
- (7): Normalen Programmablauf fortsetzen
- *Softwareinterrupts (Systemcalls):*
  - Benutzerprogramme fordern Dienste des Betriebssystems an
- *Hardwareinterrupts:*
  - Hardwarekomponenten (Systemuhr, HDD) fordern Dienste des Betriebssystems an

Unterbrechung *mit* präemptivem Scheduling (Multitasking Betriebssystem)



- **Präemptiv** (bei RTOS): Rechnerkern wird bei Interrupt der aktuell rechnende Task entzogen *wenn* höherpriore Task auf Interrupt reagieren muss
  - Interrupt zum Kontextwechsel
  - Retten des Kontextes des unterbrochenen Prozesses *j*
  - eventuelle Auftragsbearbeitung
  - *Scheduler:* Auswahl nächster Rechenprozess *i*
  - Kontext von Rechenprozess *i* laden
  - Return zu PC_i

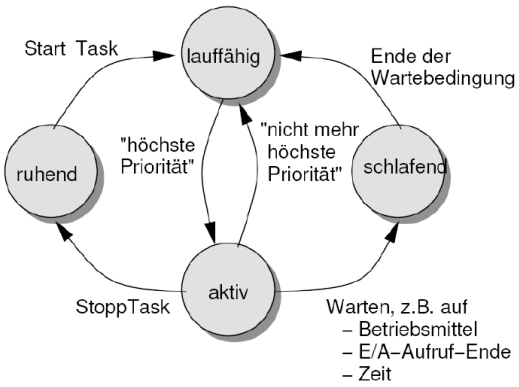
  Bei Interrupt (z.B. *Timer* oder *I/O*) wird Scheduler gestartet und zu höher priorem Prozess gewechselt um obere Reaktionsschranke eines *RTOS* einhalten zu können (Interrupt-Sperre im *Kernel* so kurz wie möglich, obere Schranke einhalten *wichtig*)
- **Nicht-präemptiv** (normales Betriebssystem): Scheduling nur bei Systemcall oder zeitgesteuert, *nicht* bei Interrupt

Task Control Block

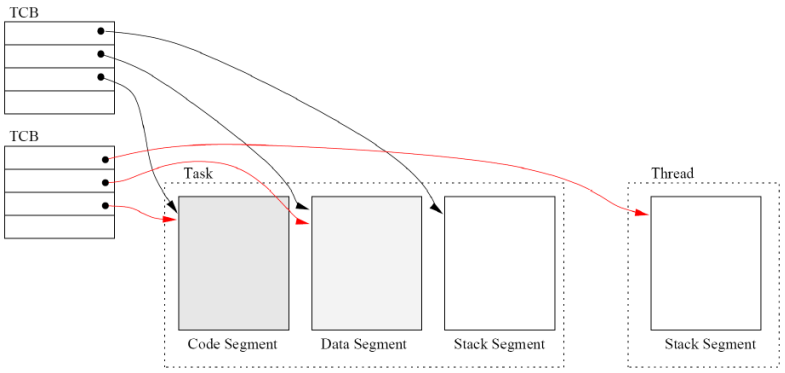
- Beinhaltet: Priorität, Maschinenzustand (Register, Stack), Task-Zustand, Zeit-Quantum, Verwaltungsdaten für Betriebsmittel (*Filedeskriptor*), Speicherabbildungstabellen für virtuellen Speicher (*Prozessadressraum* → *realer* Speicher (Code, Data, Stack))

```
char*                   name;                   /* task name */
uint                    status;                 /* status of task */
uint                    priority;               /* task's current priority */
uint                    prioNormal;             /* task's normal priority */
FUNCPTR                 entry;                  /* entry point of task */
struct sigtcb   *pSignalInfo;   /* ptr to signal info for task */
uint                    taskTicks;              /* total number of ticks */
uint                    taskIncTicks;   /* number of ticks in slice */
struct __sFile  *taskStdFp[3]; int taskStd[3];  /* stdin, stdout, stderr fps / fds */
char                    **ppEnviron;    /* environment var table */
int                             envTblSize;             /* number of slots in table */
int                             nEnvVarEntries; /* num env vars used */
EXC_INFO excInfo; REG_SET regs;         /* exception info & register set */
```

Task-Zustände



- **Tasks und Threads:**
  - *Leichtgewichtige Prozesse* um Aufwand für *Kontextwechsel* zu minimieren
  - Mehrere *Threads* teilen sich fast *kompletten* Task-Kontext
  - Lediglich *Stack* (mit Program Counter) und Thread-Status unterschiedlich
  - sind effizient zu erzeugen und zu schedulen
  - gemeinsamer (*kein* getrennter) Prozessadressraum
  - gemeinsame Betriebsmittel wie Files / Devices
  - gemeinsamer *globaler Speicher*, oft aus Effizienzgründen verwendet

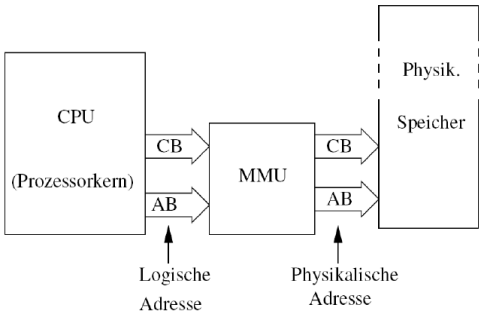


- Thread erzeugen:

```
#include <stdio.h>
void parent() {
        printf("The parent process has ID %d\n", getpid());
}
void child() {
        printf("The child process has ID %d\n", getpid());
        return;
}
int main(int argc, char **argv) {
        if( fork() != 0) {
                parent();
                wait();
        } else {
                child();
        }
        printf("Exit process %d\n", getpid());
        exit( 0 );
}
```

- *int pthread_create(pthread_t * thread, pthread_attr_t * attr, void * (*start_routine)(void*), void * arg)*:
  - Erzeugt neuen Thread (Einstiegsfunktion *start_routine* mit Argument *arg*)
  - Thread wird nebenläufig mit aufrufenden Thread abgearbeitet
    - Beenden mit *pthread_exit* oder beenden von *start_routine*
  - Attribute: *Scheduling* (Art, Parmeter), *Stack* (Größe, Adresse), *JOINABLE / DETACHED*
    - *JOINABLE: Thread Control Block* wird solange aufgehoben, bis *JOIN* auf diesen Thread aufgerufen wird
    - *DETACHED: TCB* wird direkt nach Beendigung des Threads weggeworfen
- *void pthread_exit(void * retval)*:
  - Beendigung des Threads mit *retval*
  - Alternative zu Beenden der *start_routine*
  - Cleanup-Handler aufrufen (Ressourcen freigeben (Speicher, Filedeskriptor))
- *int pthread_join(pthread_t thread, void * *thread_return)*:
  - Aufruf blockiert, bis Thread sich beendet (Ergebnis steht dann in *thread_return*)
- *int pthread_detach(pthread_t th)*:
  - Wenn keiner auf Thread wartet, räumt er sich bei Beendigung komplett auf (Thread Deskriptor, Stack)

Speichermanagement



- Aufgaben einer *Memory-Management-Unit* (MMU): Speicherschutz & Adressumsetzung
- *MMU* ist in Hardware implementiert und wird durch das Betriebssystem mittels *Speicherabbildungstabellen* konfiguriert
- **Speicherschutz:**
  - Jeder Prozess (**nicht** Thread) hat eigenen Prozessadressraum
  - Zugriff nur auf eigene *Daten*-, *Stack*- und *Code*segmente
  - Zugriff auf *nicht* abgebildete Adresse führt zur Interrupt (*Segmentation Fault*) durch *MMU*

Speicherverwaltung *ohne* MMU / *Adressumsetzung*

- Alle Programme sind zur *Link*-Zeit bekannt, Linker kann unterschiedliche Adressbereiche pro Programm zuordnen und *Sprungadressen* (d.h. Funktionsaufrufe) auflösen, es gibt ein Executable (*Image*)
- Wenn mehrere Programme dynamisch zur Laufzeit in den Hauptspeicher geladen werden sollen:
  - unterschiedliche, zur *Lade*-Zeit festgelegte Programmadressen (Sprünge bei Funktionsaufrufen)
  - Der *Loader* ersetzt Adressen zur *Lade*-Zeit eines Programms (z.B. Ersetzen des Symbols einer Funktion *printf*() durch Adresse, unter der Funktion tatsächlich verfügbar durch Symboltabelle - erst zur *Ladezeit* bekannt)
  - Verwendung von *Position Independent Code* (PIC) da natürlich zur *Compile*-Zeit absolute Sprünge nicht bekannt *PIC* bedeutet die Verwendung von *Relativsprüngen* (d.h. anstatt absoluter Sprung von $0x900 \rightarrow 0x1000$ wird relativer Sprung um $0x100$ eingetragen)

Speicherverwaltung *mit Adressumsetzung*

- Einheitlicher, *virtueller* Adressraum für Programme:
  - beginnt bei 0, umfasst kompletten adressierbaren (Adressbusbreite) Adressbereich *für jedes Programm*
  - Linker legt virtuelle Adressen in *Executable* fest, Adressen werden nicht verändert, nur durch *MMU* auf reale abgebildet
    $\rightarrow$ schnelles Laden da keine Veränderung des Executables erforderlich (nur initiale Konfiguration der MMU)
  - Verwendung von *Shared Libraries:* Mehrere *Tasks* teilen sich (Code-) Segment
    $\rightarrow$ beliebige, virtuelle Adresse durch Linker vergeben, Abbildung auf bereits geladene Shared Library durch MMU
    $\Rightarrow$ Reduziert Hauptspeicherbedarf
  - Physikalischer Adressraum meist *kleiner* als virtueller Adressraum
  - Abbildung durch *Swappen* (=Auslagern des zugeordneten, physikalischen Speichers eines *gesamten* Prozesses auf HDD)
  - Abbildung durch *Paging* (=Auslagern selten genutzter Speicherseiten (4kByte-*Pages*) auf HDD)
  - Swappen und Paging führen zu **Nichtdeterminismus**, für *Echtzeitbetriebssysteme* also ungeeignet
    $\rightarrow$ Es ist nicht deterministisch, wann & wie lange eine Page geswapped wird, auch nicht, wie lange das Laden aus HDD in RAM dauert

Adressabbildung

- Virtuelle Adresse (z.b. *32*-Bit lang) besteht aus zwei Teilen: *Seitendeskriptoradresse* [31:12] und *Seitenoffset* [11:0]
  - Seitenoffset (Adresse innerhalb einer 4kByte *Page*) ist äquivalent zur physikalischen Adresse
  - Über den *Seitendeskriptor* wird die Adressierung der *Page* vorgenommen
  - *Seitendeskriptor* enthält *Zugriffsrechte* [15:12] und tatsächliche *Page*-Nummer [11:0], Zugriffsrechte-Flags:
    - **Schreibflag:** 1 := Page darf geschrieben werden, 0 := Schreibzugriff führt zu *Bus-Error*
    - **Datenflag:** 1 := Page darf gelesen werden, 0 := Lesezugriff führt zu *Bus-Error*
    - **Codezugriff:** 1 := Prozessor darf *Pageinhalt* als Befehl ausführen, 0 := Versuch, Inhalt als Code auszuführen führt zu *Bus-Error*
    - **Validflag:** 1 := Seite ist im Hauptspeicher, 0 := Seite ausgelagert $\rightarrow$ *Seite-Fehlt-Hardware-Interrupt* $\rightarrow$ *Kernel* lädt Seite von HDD

I/O

- Aufgaben:
  - Aus **Applikationssicht:** Schnittstelle für einheitlichen Zugriff auf unterschiedlichste Hardware-Ressourcen
  - Aus **Hardwaresicht:** Umgebung, um Hardware einfach & systemkonform in Kernel zu integrieren
  - **Zusätzlich:** Realisiert Organisationsstrukturen auf Hintergrundspeicher (*Filesystem*)
- Schnittstellenfunktion:
  - *Peripherie*-Zugriffe abgebildet durch:
    - Lesen und Schreiben (*read* und *write*): $read(fd, buf, count)$; bzw. $write(fd, \&value, sizeof(value))$;
    - Konfigurieren und Einstellen der Betriebsart per $ioctl(fd, request, \ldots)$
    - Öffnen und Schließen (*open* und *close*)
      - *Open*-Funktion des Gerätetreibers wird aufgrund der *System*-Call Parameter ausgewählt (z.B. *serielle*, *parallele*, oder *analoge* Schnittstelle): Gerät als Pfad mit Dateinamen, Zugriffsart per Flag angeben; gibt *Descriptor* als Referenz zurück $fd = open('Tuer', O\_RDWR)$;
      - *Close*-Funktion gibt *Ressource* wieder frei: $close(fd)$;
  - Anforderung einer *Ressource* beim Betriebssystem ggf. Ablehnen aufgrund:
    - fehlender Zugriffsrechte
    - Ressource bereits belegt

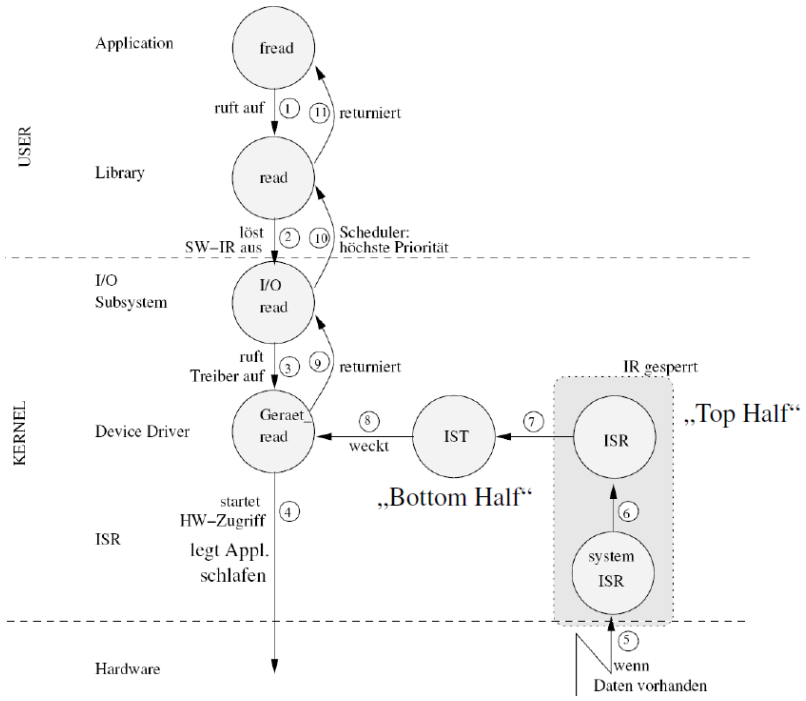
Realisierung von *Treibern*:

- *Device* wird mit Namen sowie primären- und sekundären *Identifier* angelegt: *mknod/dev/carrera* 240 0
  - Führt dazu, dass primären Identifier eine Funktion *init_module*(){...} zum Initialisieren einer *struct carrera_table* zugeordnet wird. Diese Struktur enthält *Funktionspointer* zu *Treiberfunktionen* wie *carrera_open*(){...}, *carrera_close*{...}, *carrera_write*(){...}
- Codezeile $open('/dev/carrera', O\_RDWR)$; führt dazu, dass die bei */dev/carrera* bzw. in *carrera_table* hinterlegte Funktion aufgerufen wird: *carrera_open*(){...}
- Entfernen eines *Devices* per *rmmod carrera*: es wird Funktion *cleanup_module*(){...} aufgerufen, die Zeichenkette */dev/carrera* wieder freigibt

Warum Gerät nicht aus Applikation heraus ansteuern?

- Einheitliche Ressourcenverwaltung von Interrupts & I/O-Bereiche
- Kapseln systemkritischer Teile
  - Hardwarezugriffe aus Applikation erfordern Abbildung der HW-Adressen in Prozessadressraum
  - Hardwarezugriffe sind *sicherheitskritisch* → nur innerhalb eines Treibers durchführen
  - Programmierfehler könnten zum Absturz des gesamten Systems führen
- Überführen des Geräts in sicheren Zustand bei Applikationsfehlern
  - Treiber sind nach Applikationsfehler im BS noch vorhanden
  - Gerät wird bei Beendigung einer Applikation automatisch durch Treiber freigegeben
  - Bei *sicherheitskritischen* Systemen wesentlich!

Gerätezugriff im Betriebssystem-Kern

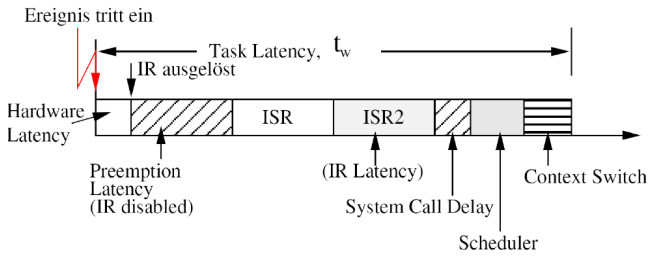


- Geräte-Schnittstelle wird von Applikation benutzt während Treiber Funktionen realisieren, die von *Kernel* selbst aufgerufen werden (z.B. *ISR*s oder Funktionen in *Kernel*- oder *Timer*-Queues)
- **Ziele:** - Verkürzung der *ISR*-Zeit, da währenddessen weitere Interrupts gesperrt
  - Verkürzen der Zeit im *System-Call* (z.B. durch *Kernel-Thread*, der nur im *Kernel* abgearbeitet wird und solange läuft, bis er sich schlafen legt oder seine Funktion mit *return* verlässt - höchste Prio)
  - Periodische Vorgänge in Treibern ermöglichen (*Polling*)
- **Top Half:** wird gleich im Interrupt ausgeführt (befüllen der *Kernel-Queue*)
- **Bottom Half:** wird in *Kernel-Queue zurückgestellt*
  - Kernel-Queue wird abgearbeitet, bevor *return* in *User Space*
- **Kernel-Queue:** Liste an (Treiber-)Funktionen, die der Kernel in bestimmten Zuständen abarbeitet, z.B. *open*(...) oder *init_module*(...)
- Linux-Kernel wird aufgerufen:
  - Nach Abarbeitung aller Interrupts (auch *Bottom Half*, hochpriore User-Threads auch Vorrang)
  - Vor Scheduling & bei jedem Tick der Systemuhr

Gerätezugriff im Betriebssystem-Kern

- *Timer-Queue:* Liste aus Paaren (*Treiberfunktion*, *Zeitpunkt*): Zum Zeitpunkt wird Treiberfunktion abgearbeitet
- *Big Kernel Lock:* Während etwas im Kernel ausgeführt wird, sind alle *ISR*s gesperrt, gibt es heute nicht mehr

Latenzzeit



- **Reaktionszeit:** $t_R = t_V + t_W$ (Verarbeitungs- + Wartezeit)
- **Hardware-Latency:** Zeit bis Hardware-Ereignis als Interrupt über Bus der CPU gemeldet wird (wenige *Gatterlaufzeiten*, im *ns*-Bereich)
- **Preemption-Delay:** *OS* im kritischen, nicht unterbrechbaren Abschnitt (Interrupts gesperrt, bei Standard-OS ca. 10*ms*)
- **ISR:** Verarbeitungszeit der *Interrupt-Service-Routine* (*Systemanteil*), bei *QNX* mit 200MHz Pentium: 1.4μs
- **System Call:** Verzögerung wenn Interrupt während eines *System-Call*s auftritt, System-Call fertig stellen oder abbrechen (bei Standard-OS viele *ms*)
- **Scheduling:** Bei *QNX* mit 200MHz Pentium: 2.9μs
- **Context-Switch:** Aufwand, um anderen Task weiterzubearbeiten (sehr variable)
- Schließlich Weiterbearbeitung höher priorer Tasks

Filesysteme

- Bei *Embedded Systems* kaum Festplatten, viel eher:
  - *RAM*-Filesysteme (beim Starten aus persistentem Speicher geladen, beim Beenden zurückgeschrieben)
  - *EEPROM*, *Flash* oder *Solid State Drive*s
- Organisation mit Filesystemen wie *FAT*, *NTFS*, *Ext3*
  - schneller Zugriff & wenig Overhead für Verwaltungsinformationen
- Verwendung von *Caches*:
  - Daten temporär inkonsistent, kein zeitlicher Determinismus bei Zugriff (man weiß nicht, wann und ob welcher Speicherbereich im Cache liegt)
  - *Sync-Mode* → keine Verwendung von Caches sondern explizit synchronisieren

Synchronisation

- **Zugriffszeit**: Zeit zwischen Auftrag und Erfüllung des Auftrags, Mechanismen zur Synchronisation von Prozessen mit Gerätezugriff
- **Synchroner Zugriff**: implizites Warten, auf Gerät wartender *Task* wird schlafen gelegt bis:
  - Gerät antwortet, Fehlersituation (z.B. *Timeout*) eintritt oder *Task* ein *Signal* erhält
- **Asynchroner Zugriff**: explizites Warten, Zugriff auf Gerät kehrt sofort zurück:
  - Task kann weiterarbeiten; Mitteilung durch OS an Task via Polling / Ereignis (*Signal*) / *Callback*-Funktion: Task holt Ergebnis ab
  - **Probleme:**
    - Mehrere Aufträge können gleichzeitig vorliegen, Dienste müssen sich daher auf selben Auftrag beziehen können
    - ggf. wird *Dienstkette* { Auftragstatus ermitteln } → { Ergebnis holen } nicht bis zum Ende einer Applikation beendet
    - Zustandsverwaltung im OS erforderlich
  - async. Zugriff mit **nicht**-blockierenden Diensten:
    - Leseauftrag → Warten auf Ergebnis → Ergebnis holen (Polling, Signal / Schlafen, Signal / Callback)
    - Höhere Performance (Audio, Video, Netzwerk, Datenbanken) + besseres Antwortverhalten
    - Umgesetzt über Funktionen wie *aio_read*, *aio_write*, *aio_fsync* (z.B. Sync. zwischen Cache und Festplatte)
  - async. Zugriff **mit** Thread und **blockierendem** Aufruf:
    - Thread erzeugen → im Thread blockierend auf Ressource zugreifen → Thread-Ende abwarten
    - Beim gleichzeitigen Warten auf mehrere Kanäle verwenden, Flag *O_NONBLOCK* um Gerät via *open*(...) nicht-blockierend zu öffnen
- **Nicht blockierender Aufruf** (ähnlich zu blockierendem Lesezugriff):
  - Liefert Ergebnis sofort wenn vorhanden, liefert andernfalls { Nichts zum Lesen vorhanden }

Zeitdienste

- **Aufgaben:** zyklische Interruptgenerierung, Zeitmessung, Watchdog (*Zeitüberwachung*), Zeitsteuerung für Dienste
  - Realisierung über *Realzeituhren* / *Timer*, sind **nicht** kontinuierlich sondern **diskret**

Zyklische Interruptgenerierung

- **Systemzeit** (Ticks):
  - Timer generiert zyklisch (z.B. alle 1*ms*) Interrupt, zugehörige *ISR*:
    - ruft Scheduler auf, bearbeitet zeitabhängige Systemdienste (*Weckrufe*), dient als *Softwaretimer*
  - Genauigkeit hängt von Zeitbasis ab
- Zyklische Bedienung von Interface-Karten (Taktgenerierung durch Bus-Controller)

Zeitmessung
- Zuordnung hochgenauer Zeitstempel zu Ereignissen
- Berechnung von Geschwindigkeiten über Differenzzeitmessung (z.B. Fahrzeuge mit Lichtschranken)

Watchdog (Zeitüberwachung):
- Zur Überwachung der Einhaltung von Echtzeitbedingungen
- Zur Zeitüberwachung einfacher Dienste (Datenausgabe, Peripherie)
- Zur Zeitüberwachung von Systemkomponenten, User-Interaktionen (*Totmann* in Lokomotiven) oder des gesamten Systems
- Mechanismus:
  - System muss in regelmäßigen Abständen einen rückwärts laufenden Timer zurücksetzen (d.h. Startwert neu setzen)
  - Wenn System in undefiniertem Zustand und unfähig, Zähler zurückzusetzen, läuft dieser auf Null → löst *Interrupt* aus:
    - Alarm / Fehlermeldung
    - Systemreset (entweder zuerst noch *ISR* aufrufen und dann CPU-Reset oder direkt CPU-Reset)
    - **Problem:** Es gibt Momente, in denen *ISR*s nicht ausgeführt werden (Interrupt Sperre), daher lieber direkt Reset!
- **Erstellen:** *WDOG_ID wdCreate* (*void*), **Löschen:** *wdDelete* (*WDOG_ID*), **Starten:** *wdStart* (*WDOG_ID*, *delay*, *FUNCPTR*, *param*), **Abbrechen:** *wdCancel*(*WDOG_ID*)

Zeitsteuerung für Dienste:
- Ausführen spezifischer Aufgaben in regelmäßigen Abständen (Backups, Messwerte erfassen)

Zeitgeber:
- *Absolut*-Zeitgeber: Uhren (*clocks*), *Relativ*-Zeitgeber: Timer (*timer*)
- Realisierung in Hardware oder per Software via **Vorwärtszähler** / **Rückwärtszähler**
  - **repetitive** Zähler: Zählwert wird nach Ablauf (Zählerstand = 0) selbstständig neu geladen
  - **single shot** Zähler: expliziter Neustart erforderlich

Zugriff auf Zeitdienste - Hardwarelevel:
- *Zeit-bezogene* Hardware: Echtzeituhren (Absolutzeitgeber), Frequenzteiler, Vor- und Rückwärtszähler, Watchdog
- **Absolutzeitgeber:** Stellen Absolutzeit (*HH* : *mm* : *ss* : *ms TT.MM.YYYY*) bereit, *batteriegepuffert*, möglichst genau bei *verteilten* Echtzeitsystemen

Zugriff auf Zeitdienste - Kernellevel:
- Zyklische Timerinterrupts im Betriebssystemkern:
  - Treiber können Tasks für variable Dauer in { wartend } versetzen
  - Module können zyklischen oder einmaligen (zu einem Relativzeitpunkt) Funktionsaufruf zulassen
- **Vorsicht** bei Realzeit im Kernel (Sekunden statt Ticks): Abstand zwischen zwei Ticks kann von System zu System variieren
- **Zählerüberlauf:** Muss in allen Schichten geprüft werden - um darauf basierende Fehler möglichst früh zu erkennen:
  → relative Zeit wird auf kurz vor Überlauf des Timers gesetzt!

Zugriff auf Zeitdienste - Userlevel:
- Absolutzeitanfragen z.B. mit *gettimeofday*(), um Prozess / Thread für definierte Zeit in Zustand { wartend } versetzen: *sleep*(*seconds*)

Kritische Punkte: in *Kernelebene:* Relativzeit führt zu Zählerüberlauf; in *User-Ebene:* Sommer-/Winterzeit, Schalttage, Zeitzone

Zeitdrift:
- Durch ungenaue Uhren, ungenaue Start-Systemzeit → *Zeitsynchronisation* mit Broadcastmessages eines *Zeitservers* (*Mikrosekundenbereich*)
  - Synchronisation mit *Network Time Protocol* (NTP) / DCF77 / statistische Berechnungen

Zeitkorrektur:
- **Problem:** Sprunghafte Korrektur von Absolutzeiten
  - Zurückstellen: Zeitpunkte kommen doppelt vor
  - Vorstellen: Zeitpunkte werden übersprungen

⇒ Abbremsen/Beschleunigen der Systemuhr: *adjtimex*(*struct timex*), keine doppelten/fehlenden Zeitpunkte, Zeit vergeht langsamer/schneller
- Durch Downtime (Serverupdate) entfallen geplante Aktionen → durchgeführte Aktionen protokollieren und bei Neustart nachholen

Repräsentation von Zeit:
- Absolutzeit: *struct time_t* (Jahr, Wochentag, Monat, Tag, Stunde, Minute, Sekunde)
- Absoluttimer: *struct timespec* {*tv_sec*, *tv_nsec*}
- Relativzeiten und Zeitpunkte für Timer: *struct itimerspec* {*it_interval*, *it_value*}
  - *it_interval* $\neq 0$: Timer startet Intervallweise, *it_value* $\neq 0$: Timer startet einmal in Absolutzeit *it_value*
  - *it_interval* $\neq 0$ **und** *it_value* $\neq 0$: Timer startet im Zeitpunkt *it_value* Intervallweise mit *it_interval*

API für Zeitdienste:
- Prozess für *n ns* schlafen legen: *nanosleep*(*timespec request*, *timespec remainder_after_signal*)
- Uhrzeit abfragen/setzen/Auflösung: *clock_gettime*(*clockid_t*, *timespec*)/*clock_settime*(*clockid_t*, *const timespec*)/*clock_getres*(*clockid_t*, *timespec*)
- Timer erzeugen/löschen/Ablauf: *timer_create*(*clockid_t*, *sigevent*, *timer_t*)/*timer_delete*(*timer_t*)/*timer_settime*(*timer_t*, *itimerspec val*, *itimerspec remain*)