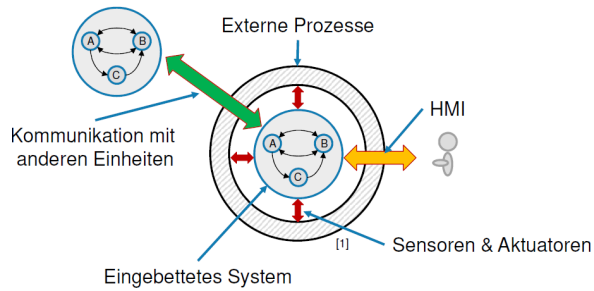


Embedded Systems

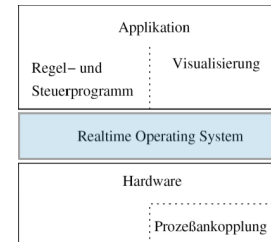
Einführung

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 *Echtzeitfähigkeit*
- **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, Mikrophon) erfassen Zustandsgrößen, *Aktoren* (z.B. Motoren, Relais, Ventile) beeinflussen sie

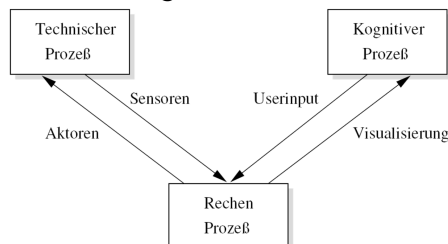
Klassifikation: *Technischer Prozess*

- **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*

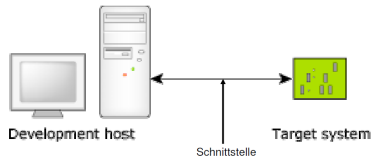
- **Steuerung**: Kein geschlossener *Regelkreis*, Rechenprozess reagiert nicht auf sich ändernde Sensorwerte im technischen Prozess
- **Regelung**: Geschlossener *Regelkreis*, Sensor- bzw. Messwerte werden verwendet, um Stellgrößen daraus zu berechnen

Self-Hosted-Entwicklung

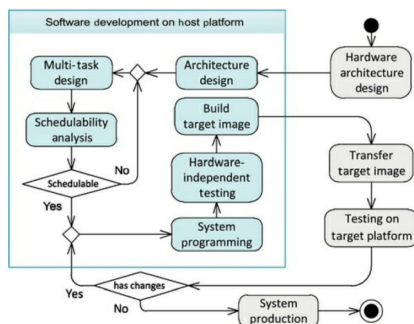
- Entwicklungsumgebung und Zielsystem sind identisch (so wie wir es alle kennen)

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*, *Variablenbelegung*) an gewählten *Breakpoints*
 - Ohne Remote-Debugger:* Konsolenausgaben per *printf*, *LEDs* 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: *Kemel-Tracer*

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

Tools: *Stack-Monitor*

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

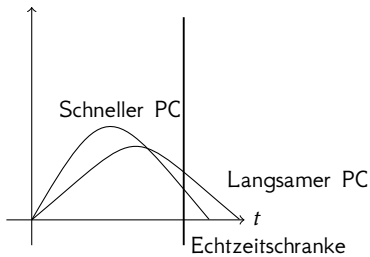
Weitere Tools

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

Echtzeitbetrieb

Kriterien für Echtzeitsysteme

Auslastung

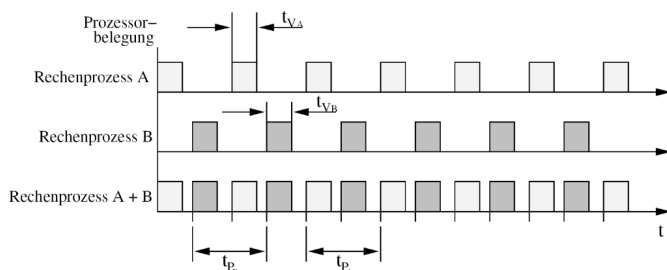


- Schnelligkeit bzw. Geschwindigkeit ist **nicht** wichtig im Kontext einer harten Echtzeitschranke ein schnellerer PC ist zwar häufiger vor der Schranke fertig, aber eben auch nicht zu 100%
- Wichtig dagegen sind:
 - Pünktlichkeit (*Ober- und Untergrenze*) bzw. Rechtzeitigkeit (*Nur Obergrenze*) (*timeliness*)
 - 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

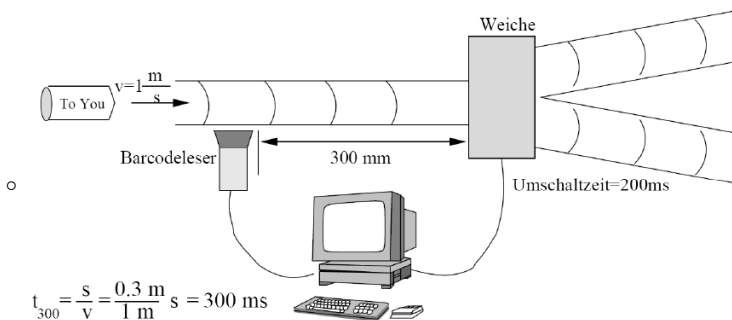
- Verarbeitungszeit von Aufgaben berücksichtigen, bei mehreren Aufgaben Reihenfolge der Abarbeitung planen
- Reihenfolge entscheidend für fristgerechte Ergebnisse
- Priorität von Aufgaben gemäß ihrer Wichtigkeit als Planungsgrundlage
- Unterbrechung einer Aufgabe muss durch einen Prozess zur Bearbeitung höher-priorer Aufgaben möglich sein
⇒ 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
 - $\rho_A = \frac{t_{vA}}{t_{pA}}$
 - $\rho_B = \frac{t_{vB}}{t_{pB}}$
 - $\rho_{A+B} = \frac{t_{vA}}{t_{pA}} + \frac{t_{vB}}{t_{pB}}$
- $\rho = \sum_{i=0}^n \frac{t_{vi}}{t_{pi}} :=$ Gesamtauslastung bei n Prozessen

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



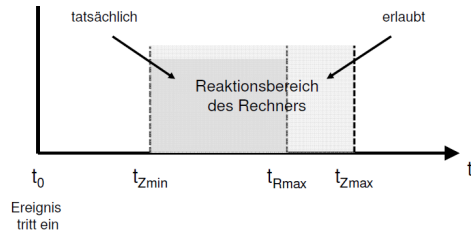
$$t_{300} = \frac{s}{v} = \frac{0,3 \text{ m}}{1 \text{ m/s}} = 300 \text{ ms}$$

$$t_{Z_{\max}} = t_{300} - t_U = 100 \text{ ms}$$

Art von Rechenprozessen

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

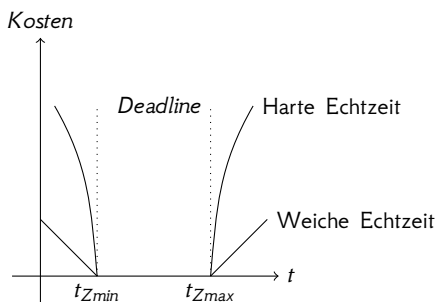
Echtzeitbedingung: Pünktlichkeit



- 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

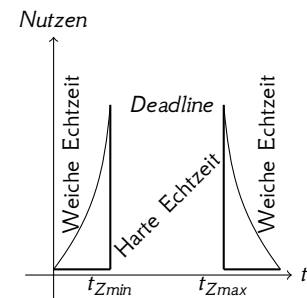
- **2. Echtzeitbedingung:** Um Aufgaben rechtzeitig zu erledigen, muss die Reaktionszeit zwischen der minimal und maximal zulässigen Reaktionszeit liegen: $t_{Zmin} \leq t_{Rmin} \leq t_R \leq t_{Rmax} \leq t_{Zmax}$

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) ⇒ 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

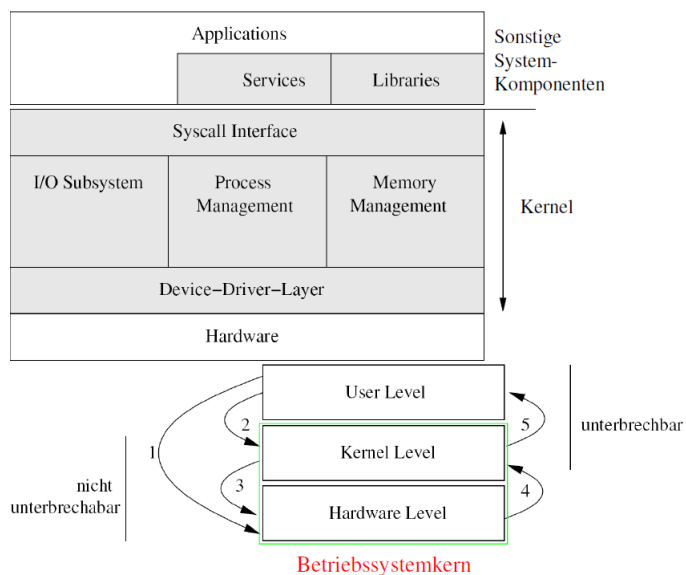


Echtzeitbetriebssysteme

Aufgaben und Anforderungen

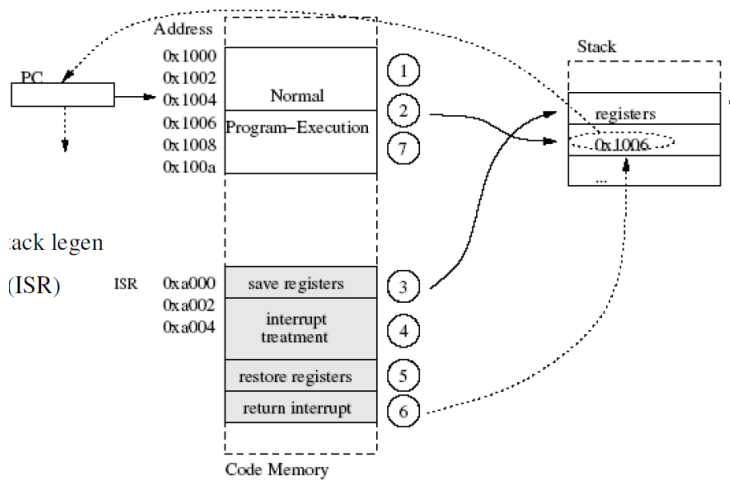
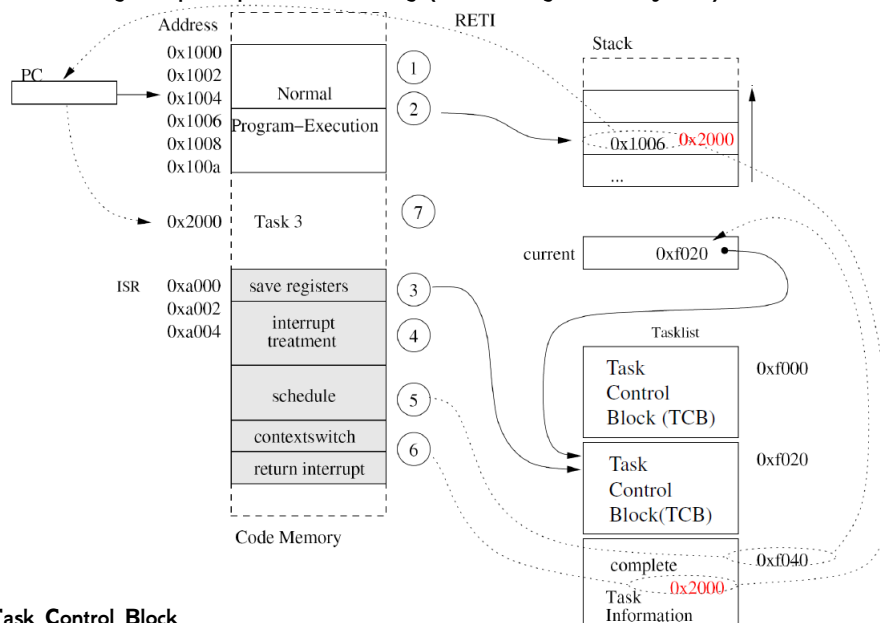
- Steuern und Überwachen: Ausführung der Benutzerprogramme & Verteilung der Betriebsmittel (Speicher, Prozessor, Dateien)
- Stellt dem Benutzer die Sicht einer einfacher als die Hardware zu bedienenden *virtuellen Maschine* zur Verfügung
 - Aus Sicht des Benutzers steht der Rechner ihm allein zur Verfügung
 - 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: Mikrokern-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* BetriebssystemUnterbrechung *mit* präemptivem Scheduling (Multitasking Betriebssystem)

- **Präemptiv** (bei RTOS): Rechnerkern wird bei Interrupt der aktuell rechnende Task entzogen *wenn* höherpriori 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 prioriem 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

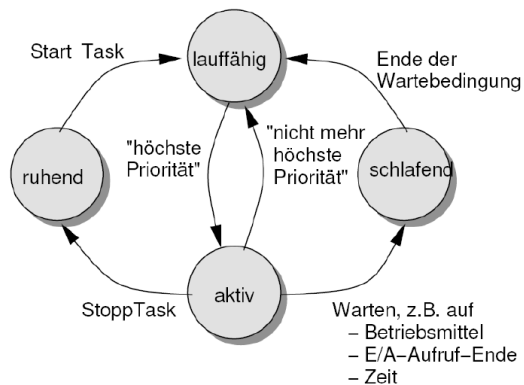
- Beinhaltet: Priorität, Maschinenzustand (Register, Stack), Task-Zustand, Zeit-Quantum, Verwaltungsdaten für Betriebsmittel (*Filedeskriptor*), Speicherabbildungstabellen für virtuellen Speicher (*Prozessadressraum* → *realer Speicher* (Code, Data, Stack))

```

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 fds / 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 */

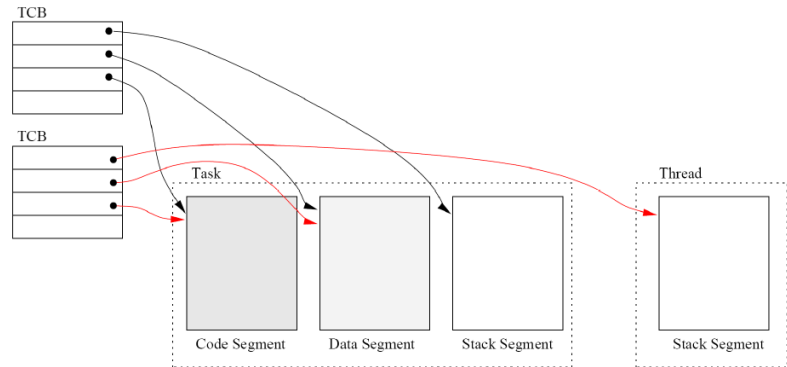
```

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 scheitern
- 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, Parameter), *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))

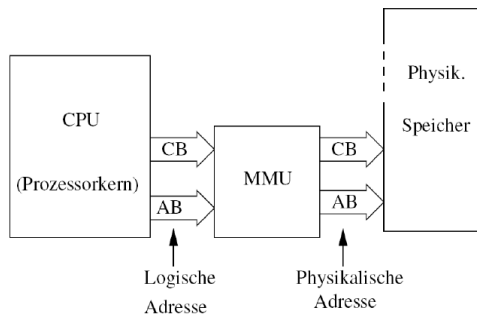
```
int pthread_join(pthread_t thread, void **thread_return):
```

- Aufruf blockiert, bis Thread sich beendet (Ergebnis steht dann in *thread_return*)

```
int pthread_detach(pthread_t tth):
```

- 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 → 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
→ schnelles Laden da keine Veränderung des Executables erforderlich (nur initiale Konfiguration der MMU)
 - Verwendung von *Shared Libraries*: Mehrere *Tasks* teilen sich (Code-) Segment
→ beliebige, virtuelle Adresse durch Linker vergeben, Abbildung auf bereits geladene Shared Library durch MMU
⇒ 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
→ Es ist nicht deterministisch, wann & wie lange eine Page gewapped 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 → *Seite-Fehlt-Hardware-Interrupt* → *Kernel* lädt Seite von HDD

I/O

o Aufgaben:

- Aus **Applikationssicht**: Schnittstelle für einheitlichen Zugriff auf unterschiedlichste Hardware-Ressourcen
- Aus **Hardware-sicht**: Umgebung, um Hardware einfach & systemkonform in Kernel zu integrieren
- **Zusätzlich**: Realisiert Organisationsstrukturen auf Hintergrundspeicher (*Filesystem*)

o Schnittstellenfunktion:

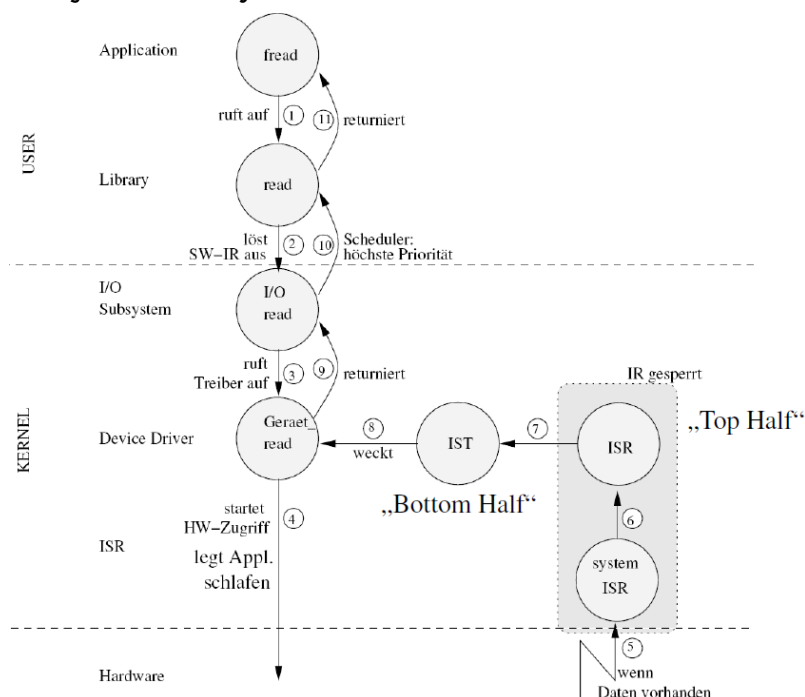
- *Peripherie*-Zugriffe abgebildet durch:
 - Lesen und Schreiben (*read* und *write*): *read(fd, buf, count)*; bzw. *write(fd, &value, sizeof(value))*;
 - Konfigurieren und Einstellen der Betriebsart per *ioctl(fd, request, ...)*
 - Öffnen und Schließen (*open* und *close*)
 - *Open*-Funktion des Gerätetreibers 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:

- o *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()*{...}
- o Codezeile *open('/dev/carrera', O_RDWR)*; führt dazu, dass die bei */dev/carrera* bzw. in *carrera_table* hinterlegte Funktion aufgerufen wird: *carrera_open()*{...}
- o Entfernen eines *Devices* per *rmmdev 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!

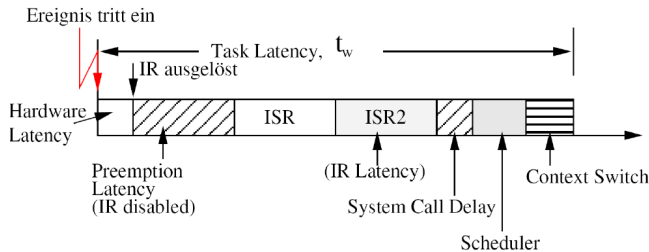
Gerätezugriff im Betriebssystem-Kern

- o Geräte-Schnittstelle wird von Applikation benutzt während Treiber Funktionen realisieren, die von *Kernel* selbst aufgerufen werden (z.B. *ISRs* oder Funktionen in *Kernel*- oder *Timer*-Queues)
- o **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*)
- o **Top Half**: wird gleich im Interrupt ausgeführt (befüllen der *Kernel-Queue*)
- o **Bottom Half**: wird in *Kernel-Queue* zurückgestellt
 - *Kernel-Queue* wird abgearbeitet, bevor *return* in *User Space*
- o **Kernel-Queue**: Liste an (Treiber-)Funktionen, die der *Kernel* in bestimmten Zuständen abarbeitet, z.B. *open()* oder *init_module()*
- o Linux-Kernel wird aufgerufen:
 - Nach Abarbeitung aller Interrupts (auch *Bottom Half*, hochpriorie 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 *ISRs* 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. 10ms)
- **ISR:** Verarbeitungszeit der *Interrupt-Service-Routine* (Systemanteil), bei QNX mit 200MHz Pentium: 1.4µs
- **System Call:** Verzögerung wenn Interrupt während eines *System-Calls* 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 Drives
- 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 1ms) 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 *ISRs* 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: Echtzeithuhren (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 *Zeitserver*s (*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* ≠ 0: Timer startet Intervallweise, *it_value* ≠ 0: Timer startet einmal in Absolutzeit *it_value*
 - *it_interval* ≠ 0 und *it_value* ≠ 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)*

Programmierertechniken bei Echtzeitbetriebssystemen

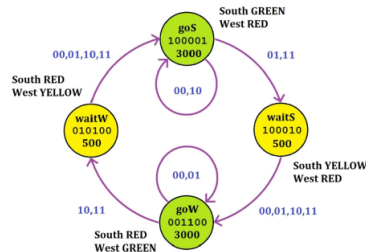
Verschiedene Programmiersprachen

- o **Assembler**: selten, bei zeitkritischen Anwendungen; **C**: Die Programmiersprache für *Embedded Systems*

C++: Seltener als C, tiefes Wissen für Effizienz erforderlich; **ADA**: Schöne, typischere Sprache, schwer *hardwarenah* programmierbar

Java: weit verbreitet, hohe Produktivität, *aber*: nicht deterministisch durch Garbage-Collection / Memory-Management → RT-Java

Endliche Zustandsautomaten (*Finite State Machines*)

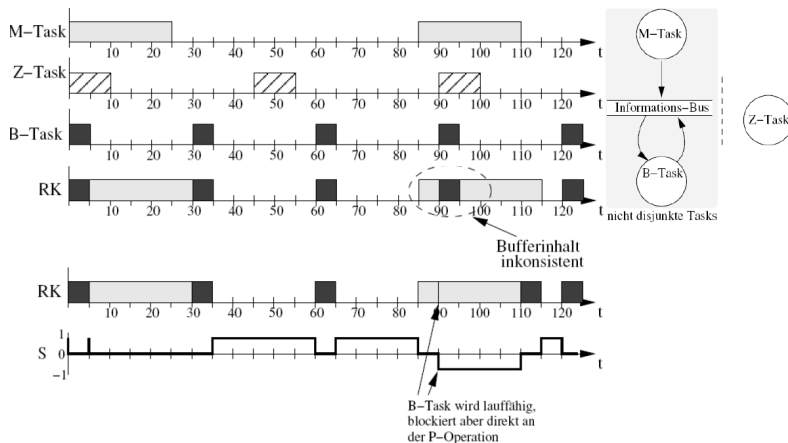


- o Weit verbreitetes *Paradigma* bei Ereignis-orientierten Systemen
- o Bestehen aus einer Anzahl an **Zuständen**, **Ereignissen**, **Transitionen** & **Ausgaben**
- o System befindet sich zu *einer* Zeit in genau *einem* Zustand
- o Eintretendes Ereignis führt zu Zustandsübergang in einen *Folgezustand*
- o Übersichtlich Implementierbar: allgemeiner Interpreter (*switch-case*) oder Beschreibung (*Automatentafel*)
- o Geeignet für **digitale** (also *diskrete*) Ereignisse bzw. Zustände (**nicht** für analoge, kontinuierliche Zustände)
- o Überprüfung des *Abdeckungsgrades* als auch der Laufzeit (deterministisch) möglich

Richtlinien zur Entwicklung eingebetteter Systeme

- o Zerlegung einer Aufgabe auf mehrere (*konkurrierende* und *kooperierende*) Tasks zur schritt haltenden Verarbeitung (*Multitasking*)
- o Unterbrechbarkeit (*Preemption*) der Tasks
- **Kontrollfluss**: Synchronisation zwischen Tasks mit *Semaphore* & *Events*
- **Datenaustausch**: Inter-Prozess-Kommunikation (*IPC*) zwischen Tasks
- **Scheduling**: Priorisierung der Tasks

Kontrollfluss am Beispiel der *Pathfinder Mars-Mission*



- o **Z-Task**: Periodische Zustandsüberwachung (Temperatur, Ströme), hat **mittlere** Priorität
 - o **M-Task**: Erfassung meteorologischer Messwerte, hat **niedrigste** Priorität
 - o **Bus-Management-Task**: Datenaustausch über Informations-Bus, hat **höchste** Priorität
 - o **RK**: Belegung des Rechnerkerns
 - o **S**: Wert der *Semaphore* für den Informationsbus
 - o **Konkurrierende Prozesse**: M- und B-Task konkurrieren um Zugriff auf Daten (sind **nicht** disjunkt)
 - o **Kooperierende Prozesse**: Prozess liefert Daten für anderen (*Erzeuger-Verbraucher-Modell*) - ebenfalls **nicht** disjunkt
 - o **Disjunkte Prozesse**: Ablauf eines Prozesses/Threads unabhängig von anderen (ebenfalls disjunkten) Prozessen
- ⇒ Die Wirkung der gegenseitigen Beeinflussung von *nicht* disjunkten, parallelen Prozessen ohne Synchronisation ist *nicht* vorhersagbar und im Regelfall auch *nicht* reproduzierbar

Kritischer Bereich

Race Condition

- o Unsynchronisierter Zugriff mehrerer Threads bzw. auf Prozesse auf *dieselben* Daten:
 - Ergebnis/Konsistenz vom Prozessfortschritt abhängig (z.B. *Ringpuffer*) → Race conditions *unbedingt* vermeiden
 - Lösung: *Gegenseitiger Ausschluss* (Mutual Exclusion) → höchstens *ein* Prozess tritt in kritischen Abschnitt ein, abgesichert durch **Mutex**

Listing 1.1: Prozess 1

```
i := read(counter); /* Liest 0 */
/* Context-Switch zu Prozess 2 */
i++;
write(counter, i);
/* Schreibt ebenfalls 1 zurück */
```

Listing 1.2: Prozess 2

```
i := read(counter); /* Liest 0 */
i++;
write(counter, i);
/* Schreibt 1 zurück*/
```

Binäre Semaphore vs Mutex

- Binäre Semaphore (hat Werte 0 & 1): exklusive Verwendung zum Erzeugen von Ereignissen im Erzeuger-Verbraucher-Problem
- Mutex: exklusive Verwendung als *Rekursiver Mutex*:
 - Wenn der Prozess bereits einen Mutex inne hat, der bei einem weiteren kritischen Abschnitt (z.B. über einen Funktionsaufruf) wieder benötigt wird, darf der Prozess eintreten, da er den Mutex ja bereits besitzt. Das erste Freigeben des Mutex (z.B. am Ende der aufgerufenen Funktion) ist nicht wirksam, er besitzt ja immer noch den Mutex für den ersten kritischen Abschnitt, erst nach dem Verlassen diesen Abschnitts wird der Mutex freigegeben:

Listing 1.3: Funktion 1

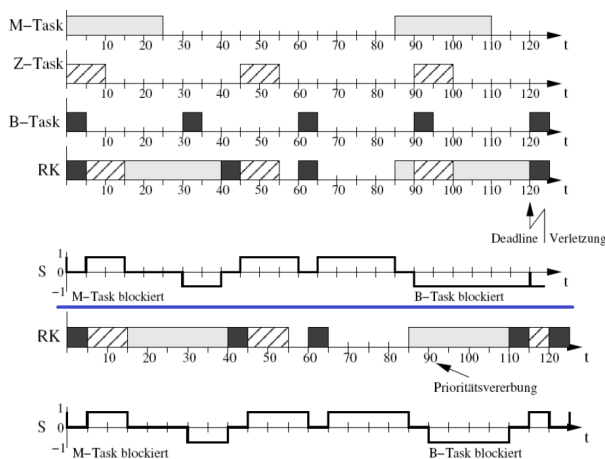
```
func init() {
    lock();
    /* do something */
    unlock();
}
```

Listing 1.4: Funktion 2

```
func main() {
    /* Mutex initial genommen */
    lock();
    init();
    /* Mutex wieder freigeben */
    unlock();
}
```

Semaphore

- Synchronisationswerkzeug: Zur Signalisierung bei Erzeuger/Verbraucher (Erzeuger *gibt* Semaphore, Verbraucher *nimmt* Semaphore)
- Bei gegenseitigem Ausschluss: Semaphore nehmen bei Betreten und geben bei Verlassen
- Integer-Variable*, wird zu Beginn auf Maximalwert N initialisiert (= Anzahl Prozesse, die kritischen Bereich betreten dürfen)
- P-Operation** (Semaphore nehmen): Bei Zugriff auf Semaphore wird Wert um 1 verringert & Prozess schlafen gelegt, falls Wert negativ
- V-Operation** (Semaphore geben): Bei Freigabe einer Semaphore wird Wert um 1 erhöht & ein wartender Prozess geweckt, falls Wert ≤ 0

Kritischer Bereich und Prioritätsinversion am Beispiel der *Pathfinder Mars-Mission*

- Niedrig-priore M -Task läuft, belegt Semaphore
- Hoch-priore B -Task fängt an zu laufen, erkennt, dass Semaphore belegt → legt sich wieder schlafen
- Z -Task läuft an, verdrängt M -Task aufgrund höherer Priorität, läuft bis fertig
- Anschließend läuft zuvor gelaufener Z -Task wieder an, nach einiger Zeit gibt es *Watchdog-Error* da B -Task zu lange nicht drann kam
- Prioritätsinversion:**
 - Tasks unterschiedlicher Priorität: **Hoch** (B -Task), **Mittel** (Z -Task), **Niedrig** (M -Task)
 - Hoch** & **Niedrig** mit gemeinsamen, kritischen Bereich
 - Niedrig** ist im kritischen Bereich, wenn **Hoch** ebenfalls betreten will
 - Hoch** wartet, bis **Niedrig** Bereich wieder verlässt
 - Mittel** verdrängt **Niedrig** → **Hoch** muss auf Beenden von **Mittel** warten
- ⇒ **Mittel** ist somit höherprior als **Hoch**

Lösung der Prioritätsinversion durch Prioritätsvererbung (*Priority Inheritance Protocol*)

- Sobald **Hoch** kritischen Bereich betreten will, ist es wichtig, dass dieser wieder freigegeben wird, **Niedrig** aber *zu unwichtig*
- Lösung: Priorität von **Hoch** auf **Niedrig** für die Zeit der Abarbeitung des kritischen Bereiches übertragen
- Allgemein:** Falls Job **Hoch** Semaphore anfordert, welche gegenwärtig von Job **Niedrig** gehalten wird, dann wird Priorität von **Niedrig** auf die Priorität von **Hoch** angehoben, sobald der Job **Niedrig** Semaphore wieder freigibt, bekommt er initiale Priorität zurück

Lösung der Prioritätsinversion durch *Priority Ceiling Protocol*

- Anheben der Priorität des Tasks **Niedrig** auf Priorität des höchstpriorierten Tasks, mit dem er sich Ressourcen teilt (kann Deadlocks verhindern)
- Schwer zu implementieren da kritische Bereiche und Tasks, die diese verwenden, dem Scheduler bekannt sein müssen

Deadlock

- Critical Sections* können zu Verklemmungen (*Deadlocks*) führen (**Task A** ruft $P(S_1)$ auf, **Task B** ruft $P(S_2)$ auf, **Task A** wartet nun auf S_2 , **Task B** auf S_1)

Schreib-/Lese Locks

- Race-Condition nur dann kritisch, wenn Rechenprozesse Daten *modifizieren*, reines *Lesen* unkritisch → *Spezielle Semaphore* erlauben parallelen Lesezugriff, Schreibzugriff **nur** exklusiv: Rechenprozess teilt bei Anforderung des Mutexes mit, ob er Lesen oder Schreiben will

1. Kritischer Abschnitt frei → Zugriff gewährt
2. Lesender Prozess belegt Critical Section → Schreibender Prozess wird blockiert, lesender zugelassen
3. Schreibender Prozess belegt Critical Section → Prozess wird abgewiesen, egal ob lesend oder schreibend

Weitere Schutzmaßnahmen

- Semaphore **nur** zum Schutz kritischer Abschnitte für Prozesse im **User-Bereich**
- Kernel-Ebene: **Unterbrechungsschutz** (Interrupts für die Zeit des Zugriffs auf kritischen Abschnitt sperren, Latenzzeiten kurz halten)
- Kernel-Ebene bei *Multiprozessorsystemen* (mehrere *ISRs* können *echt parallel* bearbeitet werden, *Unterbrechungsschutz* hilft nicht):
 - Spinlock**: Gemeinsame Variable entscheidet, ob Critical Section betreten werden darf (*Busy waiting* - Polling eines Flags, falls besetzt)
 - Spinlock kehrt im Einprozessorsystem nicht zurück - **nicht verwenden**

Methode	User Level	Einprozessorsystem: Kernel Level	SMP: Kernel Level
Unterbrechungssperre	Nein	Ja	Nein
Spinlock	Nein	Nein	Ja
Semaphore	Ja	Ja	Ja

Events

- Task wartet auf ein Event (Ereignis), andere Task setzt/signalisiert Event (mit *Semaphore* oder *Message-Queue* realisiert)

Condition Variable

- Ist ein *Event*, das an die Änderung einer Bedingung (z.B. Überschreiten eines Puffer-Füllstandes) geknüpft ist, **3 Dinge** zusammen:
 - **Condition Variable** := Event, wenn Bedingung sich ändert
 - **Global (shared) Variable** := Variable, über die Bedingung formuliert ist
 - **Mutex** := sichert Bereich { Überprüfen der Bedingung/Warten auf Veränderungssignal } ab

```
pthread_mutex_t spar_lock;
pthread_cond_t spar_aenderung;
unsigned int erspartes;

void ausgeben(int menge) {
    pthread_mutex_lock(&spar_lock);
    while(erspartes < menge) {
        /* gibt 'spar_lock' frei, damit einzahlen überhaupt
         * möglich ist, ansonsten gäbe es 'Deadlock'
         */
        pthread_cond_wait(&spar_aenderung, &spar_lock);
    }
    erspartes += menge;
    pthread_mutex_unlock(&spar_lock);
}

void einzahlen(int menge) {
    pthread_mutex_lock(&spar_lock);
    erspartes += menge;
    pthread_cond_broadcast(&spar_aenderung);
    pthread_mutex_unlock(&spar_lock);
}
```

Signale

- *Signal* führt zu Unterbrechung des Programmablaufs innerhalb einer Applikation:
 - Programm wird abgebrochen (*exit()*)
 - Programm reagiert mit implementierten *Signal – Handler()* (ähnlich zu einer *Interrupt-Service-Routine*)
- *Signal* kann ausgelöst werden durch:
 - Programm
 - Ereignisse innerhalb des Betriebssystems selbst: Zugriff auf nicht vorhandenen Speicherbereich (→ *Segmentation-Fault-Signal*)
 - ggf. Abfangen um noch wichtige Daten zu sichern

Signale	Events
Kommen <i>asynchron</i> zum Programmablauf werden <i>asynchron</i> verarbeitet Ähnlich zu <i>Interrupt-Service-Routine</i>	Kommen <i>synchron</i> zum Programmablauf werden <i>synchron</i> verarbeitet <i>Rendezvous</i> -Charakter

- **Signale** führen außerdem zum sofortigen Abbrechen eines gerade aktiven, blockierenden *System-Calls*
- Jeden Systemcall überprüfen, ob durch Signal unterbrochen (*errno = EINTR*) & neu aufsetzen falls ja
- *sigaction(signum, struct sigaction *act, sigaction):* *act enthält Adresse des *Signal-Handlers*, der bei Signal von OS aufgerufen wird
- *kill(pid_t, sig):* einem Task oder einer Task-Gruppe wird ein Signal geschickt

Message Queues

- Zum Transport von Daten zwischen Tasks (*Inter-Prozess-Kommunikation*)
- Daten mit unterschiedlicher, nach oben begrenzter Größe von Task A nach Task B senden
- Über Filesystem mit *Namen*, *task-intern* per *Message Queue ID* identifiziert
- Blockierendes Senden (Queue voll) & Empfangen (Queue leer)

Shared Memory

- Gemeinsamer Speicherbereich (= globale Variable(n)), Realisierung *trivial* - aber *Critical Section* nicht vergessen
- Realisierung z.B. über OS-Filesystem, da Speicheradresse von Task zu Task unterschiedlich sein kann, sollte sie innerhalb von Datenstrukturen relativ zu Beginn des Speicherbereichs angegeben werden

Sockets

- Kommunikation über Rechnergrenzen hinweg (aber auch *lokal*) möglich
- Verbindung identifiziert über *Rechnername/IP* + *Port-Nummer*: *Server* öffnet Port (bei TCP zusätzlich Handshake), *Client* öffnet Verbindung & sendet an Server

Busy-Loops

- Echtzeitsteuerung ist nur mit der Regelung **eines einzigen** technischen Prozesses beschäftigt (*volatile* bei Pollen nicht vergessen)
- Rechnerprozess pollt ständig technischen Prozess nach Anforderungen; ist einfach programmierbar, kein OS nötig, sehr häufig vertreten

```
while(True) {
    Lese Wert ein
    Bearbeite Aufgabe
    Warte (100ms)
}
```

Time-Triggered

- Applikation prüft zyklisch, ob bestimmte Bedingung erfüllt ist (Zeitbasis aus dem Betriebssystem, Peripherie ist **rein passiv**)
- Abtastzeit so kurz wählen, dass: kein Zustand verloren geht (*Abtasttheorem*) / Reaktion rechtzeitig erfolgen kann (**2. Echtzeitbedingung**)

```
while(True) {
    Sende Ereignis in 100ms
    /* Verarbeitungszeit */
    Lese Wert ein
    Bearbeite Aufgabe
    /* Wartezeit */
    Warte auf Ereignis
}
```

Event Triggered

- Applikation reagiert auf Ereignisse/(Software-)Interrupts, die von anderen Tasks oder dem OS (z.B. *Treiber*) generiert werden
 - Im Regelfall am effizientesten; Zeitbasis (*isochroner Modus* von Echtzeitbuss) häufig exakter als im Betriebssystem
 - *Determinismus* hängt von zeitlicher Verteilung der externen Events ab (z.B. Überlastung des Systems wenn Benutzer ganz oft Knopf drückt)
- Kein Einfluss auf Häufigkeit der Events (ggf. **nicht echtzeitfähig**)

Verfahren	Charakteristik
<i>Busy-Loop</i>	100% Auslastung, unabhängig vom Bedarfsfall
<i>Time-Triggered</i>	Auslastung abhängig vom gewählten Abtastintervall, unabhängig vom Bedarfsfall
<i>Event-Triggered</i>	Auslastung abhängig vom Bedarfsfall

Programmierstil

- Zeitkritische Tasks mit **möglichst wenig** Systemaufrufe
- Zeitaufwändige Aufgaben in eigene Tasks auslagern ⇒ Werden durch Scheduling des Threads in einzelne Aufgaben zerlegt
- Rekursion vermeiden **da Stack stark beschränkte Ressource** ist ⇒ iterativ programmieren
- Keine Schleifen die im schlimmsten Fall **endlos** auf Event warten ⇒ Abbruchbedingung mit maximaler Anzahl an Durchläufen
- Datenstrukturen statisch oder zur Initialisierungszeit allozieren - nicht erst zur Laufzeit ⇒ **kein new**
 - nicht-deterministisches Verhalten der Speicherverwaltung & eventuell kein Speicher mehr verfügbar
- Rückgabewerte von Funktionen und Return-Codes grundsätzlich **im kompletten Definitionsbereich** auswerten

Echtzeitnachweis

Aufbau und Eigenschaften

- Formaler Nachweis, dass **1. & 2. Echtzeitbedingung** erfüllt ist
- Man geht vom **schlimmst-möglichen Fall** aus
 - Prozess- bzw. Zykluszeiten möglichst kurz
 - Verarbeitungszeit möglichst lang
 - Alle Ereignisse treten gleichzeitig auf ($t = 0$)
- Ablauf
 - Relevante Kenndaten des Prozesses ermitteln
 - Anforderungen, Minimale Prozesszeit, Minimale & Maximale Reaktionszeit t_{Zmin} , Abhängigkeiten zwischen Ereignissen
 - Maximale Verarbeitungszeit t_{Vmax} (**WCET**) für jede Anforderung
 - Auslastungsbedingung überprüfen
 - Rechtzeitigkeitsbedingung verifizieren (Bestimmung von t_{Rmin} & t_{Rmax})

Abschätzen der *Worst Case Execution Time*: Messen

- Messen durch Anweisungen im Code

```
while(true) {
    /* Warten auf Ereignis */
    Zeitstempel t1 setzen
    /* Verarbeitung */
    Zeitstempel t2 setzen
}
```

- Vorteile: Sprachunabhängig & einfach realisierbar
- Nachteile
 - Aussagekraft der Messung abhängig von vielen Randbedingungen (Cache, Prozesszustand, Schleifen)
 - Theoretisch wären sämtliche Kombinationen aus Inputdaten erforderlich (100% Test-Abdeckung)
 - Produktiver Code auf Zielplattform (erst spät in der Entwicklung) oder Simulator erforderlich

Abschätzen der *Worst Case Execution Time*: Analyse

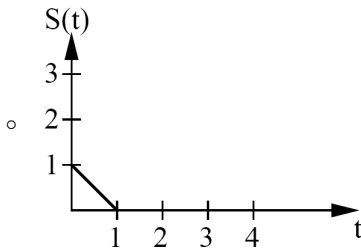
- Aus Quellcode wird Strukturgraph des Codestücks erstellt, mit Beschreibung der Hardware wird längster Pfad durch Graphen gesucht
 - Vorteile
 - Analyse auch nach Codemodifikation schnell durchführbar
 - Frühzeitig ohne Zielhardware möglich
 - Größtmögliche Abdeckung des Inputs
 - Nachteile
 - Komplexe Analysewerkzeuge erforderlich
 - Verifikation der Ergebnisse (stimmt die Hardwarebeschreibung?)

Abschätzen der *Best Case Execution Time* (BCET)

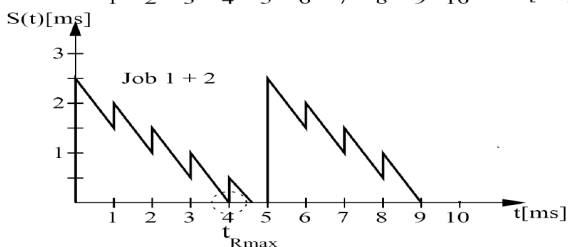
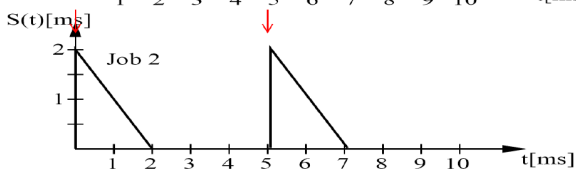
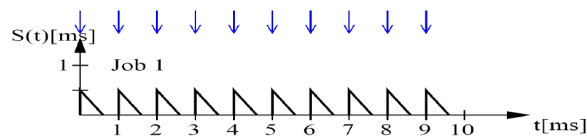
- Erforderlich wenn Codesequenz nicht vor einer minimal zulässigen Reaktionszeit t_{Zmin} abgearbeitet worden sein darf
- BCET = kürzester Pfad im Strukturgraphen
- Im Gegensatz zur WCET misst man bei möglichst geringer Systemlast

Grafische Bestimmung der maximalen Reaktionszeit t_{Rmax}

- Kürzere Prozesszeit entspricht höhere Priorität (**Rate Monotonic**), d.h. der Prozess mit der längsten Prozesszeit ist derjenige, mit der geringsten Priorität
- Bekannt sein muss
 - zulässige Reaktionszeiten t_{Zmin} und t_{Zmax} für jeden Job
 - Verarbeitungszeiten t_v , BCET und WCET für jeden Job
- Voraussetzung: **prioritätengesteuertes (preemptives) Scheduling** & Ereignisse sind **unabhängig** voneinander



- **X-Achse:** Zeit
 - S = aktuelle Systemlast
 - Rechenzeitanforderungen im Falle des *Worst Case*
 - Bei ungestörter Abarbeitung bildet sich eine mit Steigung -1 fallende Gerade
 - Bei jeder Anforderung wird die Gerade um Verarbeitungszeit t_v nach oben verschoben
 - Der Schnittpunkt der Gerade mit der X-Achse entspricht der **maximalen Reaktionszeit** des **niedrigpriorsten** Prozesses
 - Um maximale Reaktionszeit des nächst-niedrigpriorsten Prozesses zu erhalten: X-Achse um t_v des niedrigprioreren Prozesses **nach oben** verschieben
 - Zu jedem Zeitpunkt lässt sich ablesen, wie viel Rechenarbeit noch zu leisten ist
- Es reicht aus, den Graph **eine Hyperperiode** lang zu betrachten, da dies der maximale Zyklus ist, ab dann beginnt alles von vorne
- Eine Hyperperiode ist das kleinste gemeinsame Vielfache der Prozesszeiten aller Prozesse

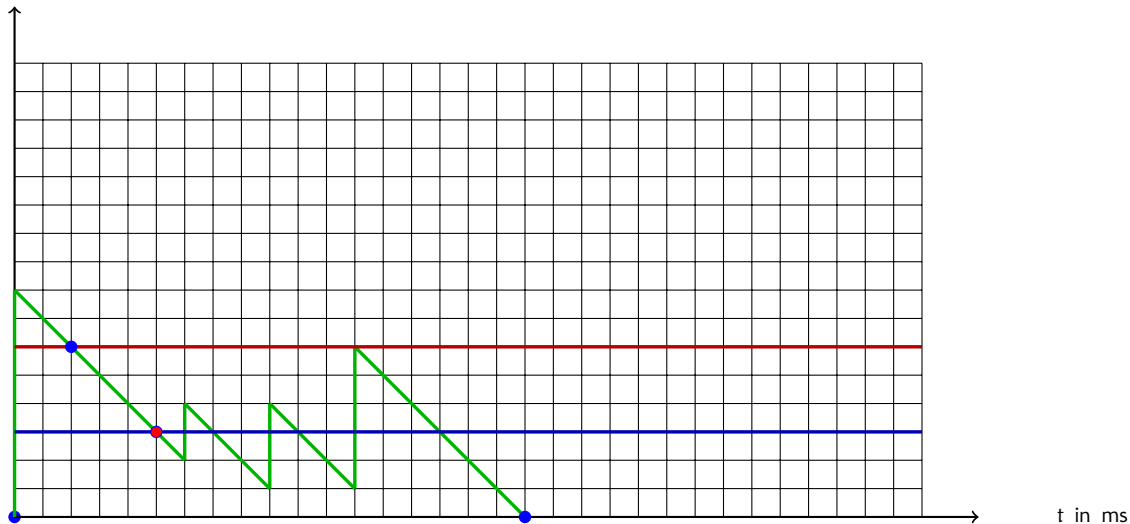


- **Job-1:** $t_v = 0.5ms$, $t_p = 1ms$
- **Job-2:** $t_v = 2ms$, $t_p = 5ms$
- Bei **4ms** ist der erste Schnittpunkt des Graphen mit der X-Achse, d.h. die maximale Reaktionszeit für den niedrigpriorsten Prozess (= Job-2) ist 4ms, da $4 < 5$ ist das **System echtzeitfähig**
- Die **Hyperperiode** ist $kgV(1,5) = 5$, d.h. das System muss maximal 5ms betrachtet werden

Anforderung	t_v	t_p	t_{Zmax}	t_{Zmin}
1	10ms	30ms	30ms	0ms
2	15ms	45ms	45ms	0ms
3	15ms	60ms	60ms	0ms

mit einer **Hyperperiode** $kgV(30, 45, 60) = 180ms$

$S(t)$



- o Durch den Schnittpunkt des grünen Graphen mit der X-Achse bei $t = 90ms$ ergibt sich für die maximale Reaktionszeit für Job-3 = 90ms, da $90ms > 60ms$ ist das System **nicht echtzeitfähig**
- o Die blaue Gerade ist die Verschiebung der X-Achse nach oben, wenn Job-3 nicht berücksichtigt wird
 - die maximale Reaktionszeit für Job-2 = 25ms, da $25ms < 45ms$ ist das System **echtzeitfähig**
- o Die rote Gerade ist die Verschiebung der X-Achse, wenn weder Job-3 noch Job-2 berücksichtigt werden
 - die maximale Reaktionszeit für Job-1 = 10ms, da $10ms < 30ms$ ist das System **echtzeitfähig**

Hinreichende Schedulingbedingung

- o Voraussetzungen
 - Scheduling mit statischen Prioritäten (**Rate Monotonic**) - wie oben
 - Zyklische Tasks ohne Abhängigkeiten
 - $t_{Zmax} = t_p$
 - Falls $t_{Zmax} < t_p$: Maximal zulässige Reaktionszeit durch entsprechende Prozesszeit modellieren: $\min(t_{Zmax}, t_p)$
- o Aussage: Falls Bedingung für Auslastung erfüllt, wird jeder Task innerhalb seiner Prozesszeit fertig, somit ist **Rechtzeitigkeit erfüllt**
 - Falls nicht, kann die Rechtzeitigkeit eventuell trotzdem erfüllt sein
 - Es ist dann keine genauere Analyse (graphisch, mathematisch) erforderlich

$$\mu = \sum_{k=1}^i \frac{t_{V_k}}{\min(t_{Zmax}, t_{p_k})} \leq i \cdot (2^{\frac{1}{i}} - 1)$$

- o Mit zunehmender Prozesszahl $i \rightarrow \infty$ nähert sich die Auslastungsgrenze **69.3%** an

