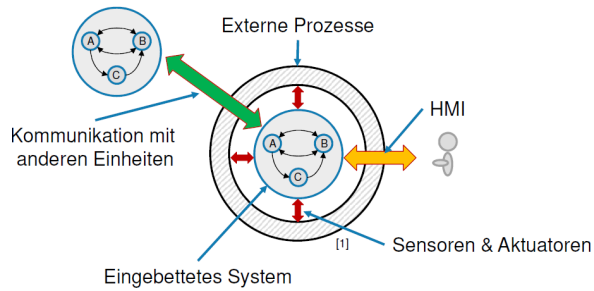


Embedded Systems

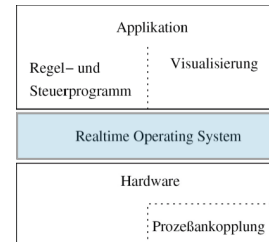
Einführung

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

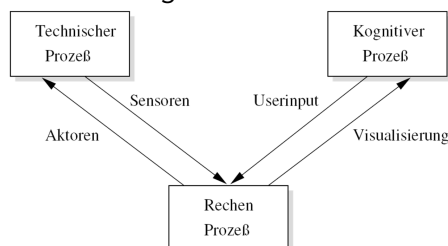
Klassifikation: *Technischer Prozess*

- **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*

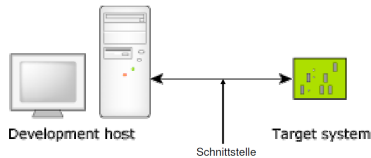
- **Steuerung**: Kein geschlossener *Regelkreis*, Rechenprozess reagiert nicht auf sich ändernde Sensorwerte im technischen Prozess
- **Regelung**: Geschlossener *Regelkreis*, Sensor- bzw. Messwerte werden verwendet, um Stellgrößen daraus zu berechnen

Self-Hosted-Entwicklung

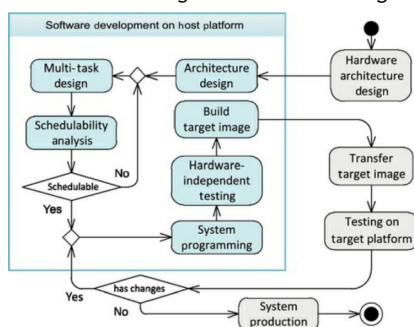
- Entwicklungsumgebung und Zielsystem sind identisch (so wie wir es alle kennen)

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

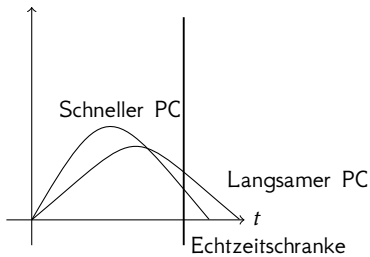
Weitere Tools

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

Echtzeitbetrieb

Kriterien für Echtzeitsysteme

Auslastung

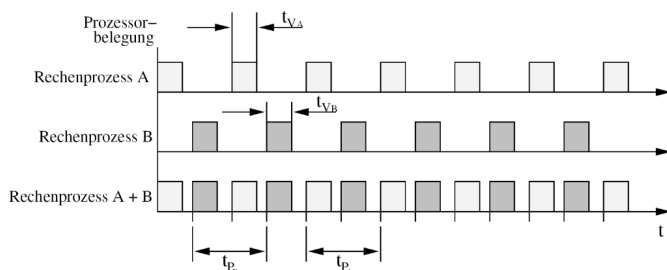


- Schnelligkeit bzw. Geschwindigkeit ist **nicht** wichtig im Kontext einer harten Echtzeitschranke ein schnellerer PC ist zwar häufiger vor der Schranke fertig, aber eben auch nicht zu 100%
- Wichtig dagegen sind:
 - Pünktlichkeit (*Ober- und Untergrenze*) bzw. Rechtzeitigkeit (*Nur Obergrenze*) (*timeliness*)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



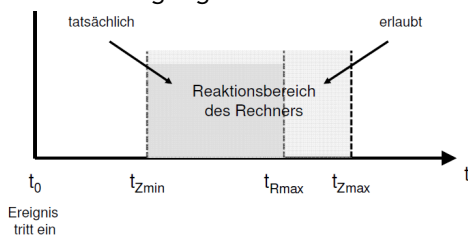
- t_V := Verarbeitungszeit
- t_P := Prozesszeit, Abstand zwischen zwei Anforderungen (*Jobs*) desselben Typs, wenn t_P konstant handelt es sich um einen *zyklischen* bzw. *periodischen* Prozess
- $\rho = \frac{t_V}{t_P}$:= Auslastung
 - $\rho_A = \frac{t_{VA}}{t_{PA}}$
 - $\rho_B = \frac{t_{VB}}{t_{PB}}$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

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

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*)

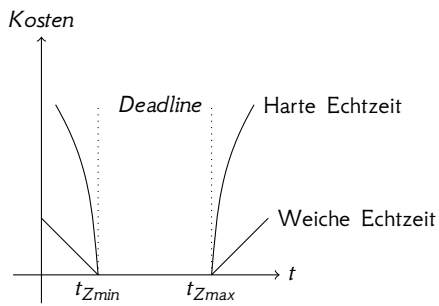
Echtzeitbedingung: Pünktlichkeit



- 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

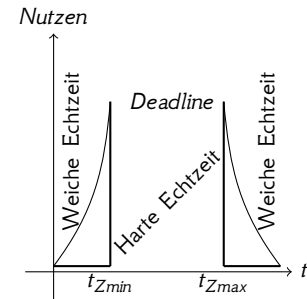
- **2. Echtzeitbedingung:** Um Aufgaben rechtzeitig zu erledigen, muss die Reaktionszeit zwischen der minimal und maximal zulässigen Reaktionszeit liegen: $t_{Zmin} \leq t_{Rmin} \leq t_R \leq t_{Rmax} \leq t_{Zmax}$

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

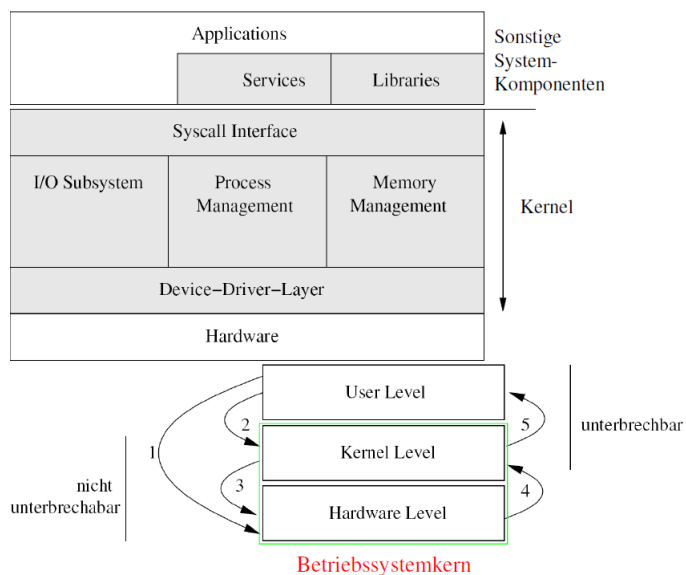


Echtzeitbetriebssysteme

Aufgaben und Anforderungen

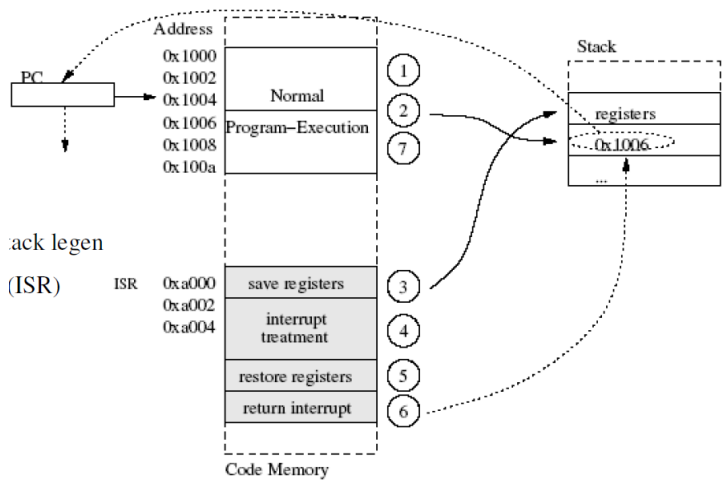
- Steuern und Überwachen: Ausführung der Benutzerprogramme & Verteilung der Betriebsmittel (Speicher, Prozessor, Dateien)
- Stellt dem Benutzer die Sicht einer einfacher als die Hardware zu bedienenden *virtuellen Maschine* zur Verfügung
 - Aus Sicht des Benutzers steht der Rechner ihm allein zur Verfügung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

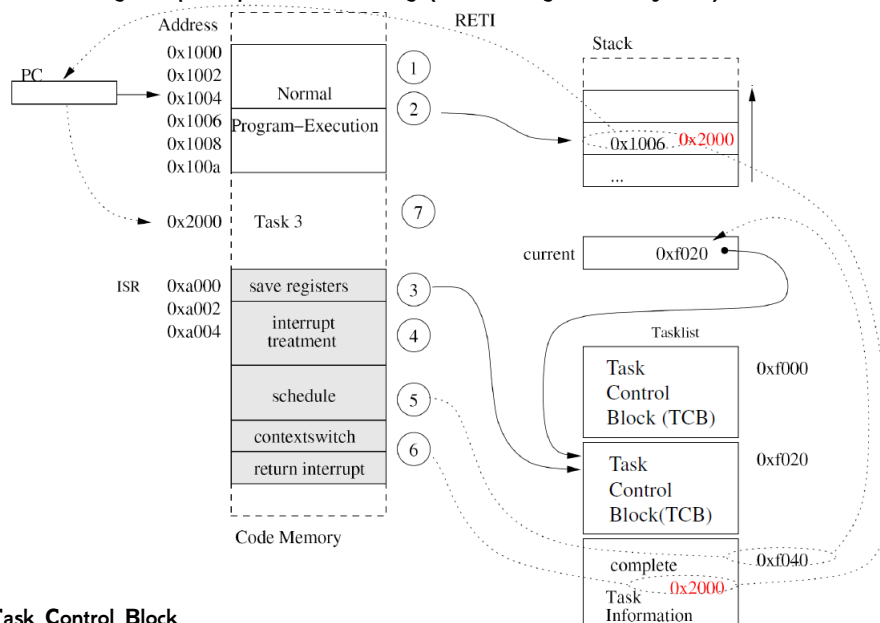


- Ein *Betriebssystem* besteht aus aufbauenden *Systemkomponenten* (Dienstprogramme, Werkzeuge) und einem *Betriebssystemkern*
- (1): *Hardware-Interrupt*
- (2): *Software-Interrupt (Systemcall)*
- (3): *Hardware-Interrupt* (während eines Systemcalls)
- (4): *Hardware-Interrupt (Scheduler wird aufgerufen)*
- (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* Betriebssystem

- (1): CPU arbeitet Programm ab
- (2): Interrupt während der Programmarbeitung:
 - 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*, *Befehlszähler*) 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* höherpriorie 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 priorie 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

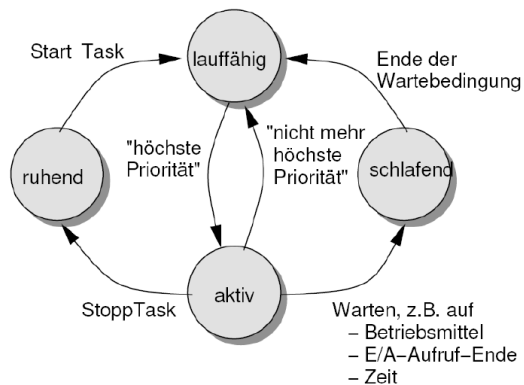
- Beinhaltet: Priorität, Maschinenzustand (Register, Stack), Task-Zustand, Zeit-Quantum, Verwaltungsdaten für Betriebsmittel (*Filedeskriptor*), Speicherabbildungstabellen für virtuellen Speicher (*Prozessadressraum* → *realer Speicher* (Code, Data, Stack))

```

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];   /* 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 */

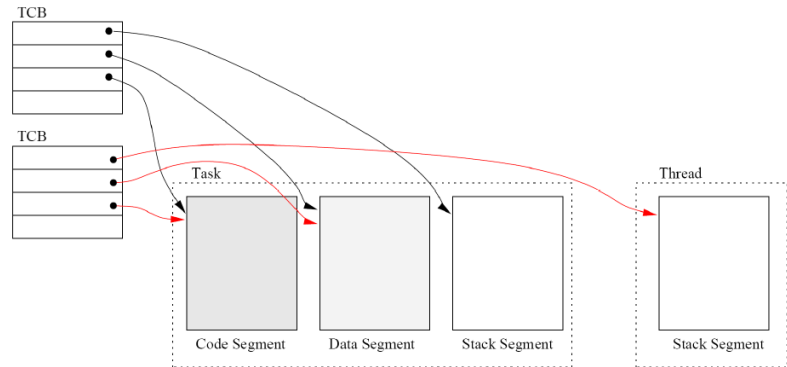
```

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)