

# Master Studiengang

---



**h\_da**

HOCHSCHULE DARMSTADT  
UNIVERSITY OF APPLIED SCIENCES

**fbi**  
FACHBEREICH INFORMATIK

## Realtime Systeme (Teil 6/2)

Prof. Dr. Ralf S. Mayer,  
Prof. Dr. Peter Altenbernd  
Fachbereich Informatik, h\_da

- Echtzeit-Betriebssysteme
  - Definitionen
  - Architekturen
    - UNIX
    - Kernel-Konzepte
  - Kernel Aufgaben
  - Kernel Konzepte
  - Anforderungen an RTOS
  - Struktur eines einfachen RTOS
  - Semaphore
  - Prozesskommunikation
- Designaspekte eines Echtzeitsystems

# Definition RTOS

---

Ein Echtzeitbetriebssystem oder auch RTOS (real-time operating system) ist ein Betriebssystem, welches für die Einhaltung von Zeitbedingungen optimiert wurde. Im Vergleich zu anderen Betriebssystemen, verläuft das Weiterleiten und Verarbeiten von Systemnachrichten und Signalen schneller. Echtzeitbetriebssysteme sind in der Regel deutlich kompakter und kommen häufig in eingebetteten Steuerungsrechnern zum Einsatz.

Ein **Echtzeitbetriebssystem** oder auch **RTOS** (real-time operating system) ist ein Betriebssystem mit zusätzlichen Echtzeit-Funktionen für die Einhaltung von Zeitbedingungen und die Vorhersagbarkeit des Prozessverhaltens.

# Aufgaben eines Kerns (Kernel)

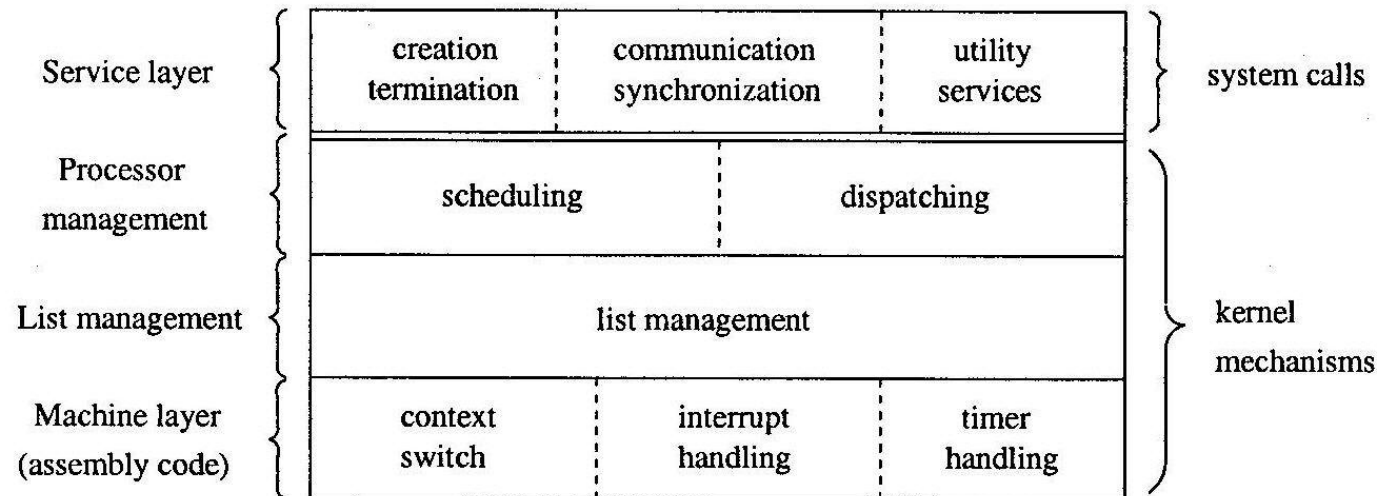
---

- Dienste für den Zugriff auf Hardware
  - Arbeitsspeicherverwaltung
  - Verwaltung der Ein- und Ausgabeprozesse
  - Unterste Ebene der Gerätesteuerung (Treiber)
  - Interrupt-Bearbeitung
  - Ausführung privilegierter Befehle (nur im Systemmodus ausführbar)

... eventuel zusätzliche Aufgaben wie:

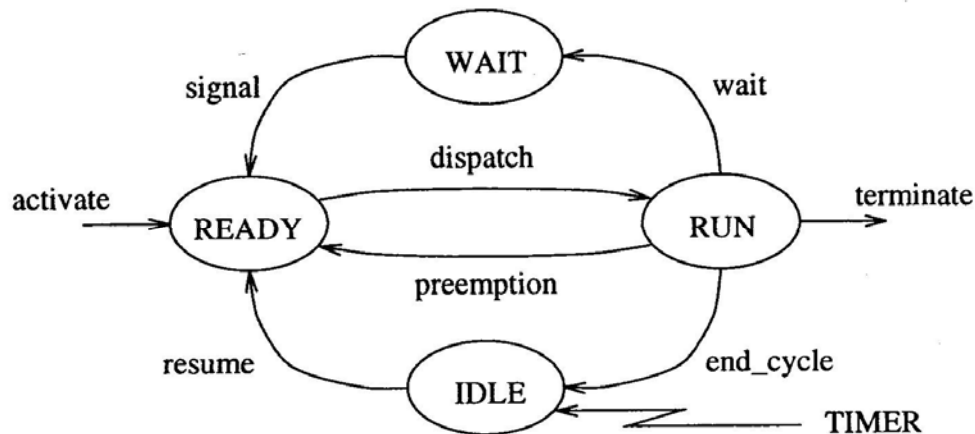
- Prozess-Management
- Interrupt-Bearbeitung
- Prozess-Synchronisierung
  - Semaphore  
die ein Ressourcen-Zugriffsprotokoll unterstützt
- Kommunikation zwischen Prozessen

# Struktur eines einfachen RT OS



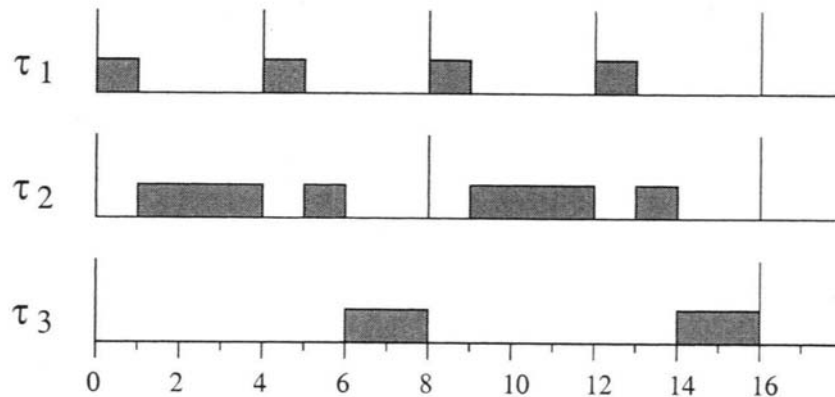
- **Maschinenebene:** Interagiert direkt mit der physikalischen HW → Assembler. Erledigt im wesentlichen: Context-Switch, Interrupt- und Timer Handling
- **Listenmanagement:** Verfolgt Status der Tasks, fügt neue Tasks ein, entfernt Tasks
- **Prozessor-Management:** Planung der Ausführung (schedule), Abfertigung (dispatch) von Operationen
- **Dienste-Ebene:** alle für den Benutzer sichtbaren Dienste und Systemaufrufe

# Prozesszustände



- **Running.** Task in diesem Zustand wird vom Prozessor ausgeführt
- **Ready.** Task ist bereit zur Ausführung. Alle Tasks in diesem Zustand sind in der *ready queue*
- **Waiting.** Task in diesem Zustand ist in einem Synchronisationszustand und wartet auf ein Event. Tasks in diesem Zustand werden in die zu dem *Semaphor* gehörende Queue eingeordnet. Bei einem *signal* wird der obere Task in der Queue in die *ready queue* eingeordnet.
- **Idle.** Periodischer Task, der seine Ausführung beendet hat macht dies durch *end cycle* bekannt und wird in die *idle queue* eingeordnet. Wird vom Kernel rechtzeitig wieder geweckt und in die *ready queue* eingeordnet.

# Scheduling



Beispiel:  
Scheduling von drei Tasks  
nach RMS

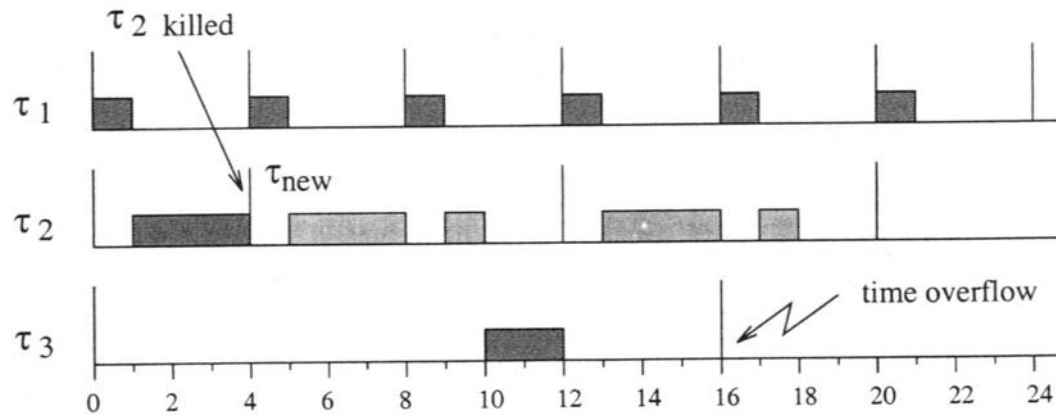
In Echtzeitsystemen muss auch die dynamische Erzeugung und Beendigung von Tasks berücksichtigt werden. Ein neuer Status muss eingeführt werden, um die Prozessor-Bandbreite zu erhalten, welche für die garantierte Ausführung von Echtzeit-Tasks vorgesehen ist.

Der Auslastungsfaktor  $U_k$  (utilisation factor) eines abgebochenen Tasks (z.B. durch kill) kann nicht sofort von der totalen Prozessorauslastung abgezogen werden, weil er möglicherweise schon die Ausführung anderer Tasks verursacht haben könnte.

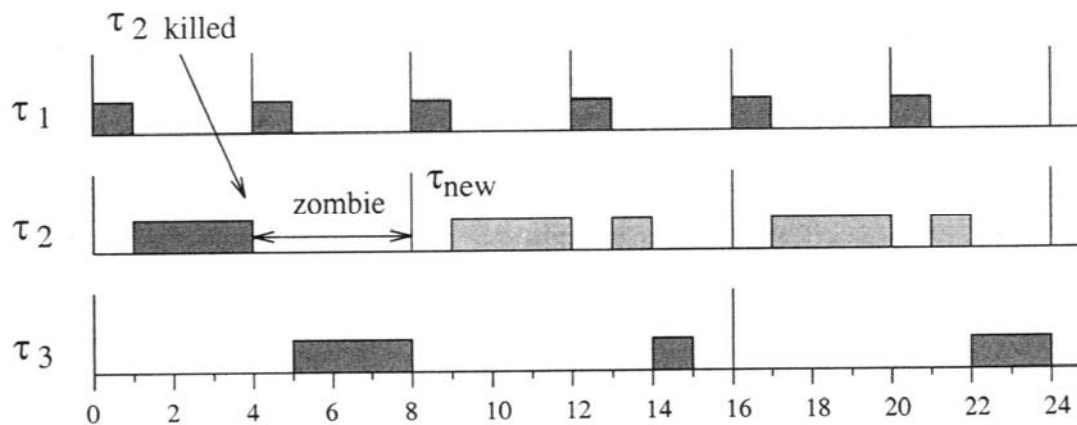
Um den Garantie-Test konsistent zu halten, kann  $U_k$  erst am Ende einer Periode abgezogen werden.

Daher wird u.a. der Status Zombie eingeführt.

## Scheduling (2)

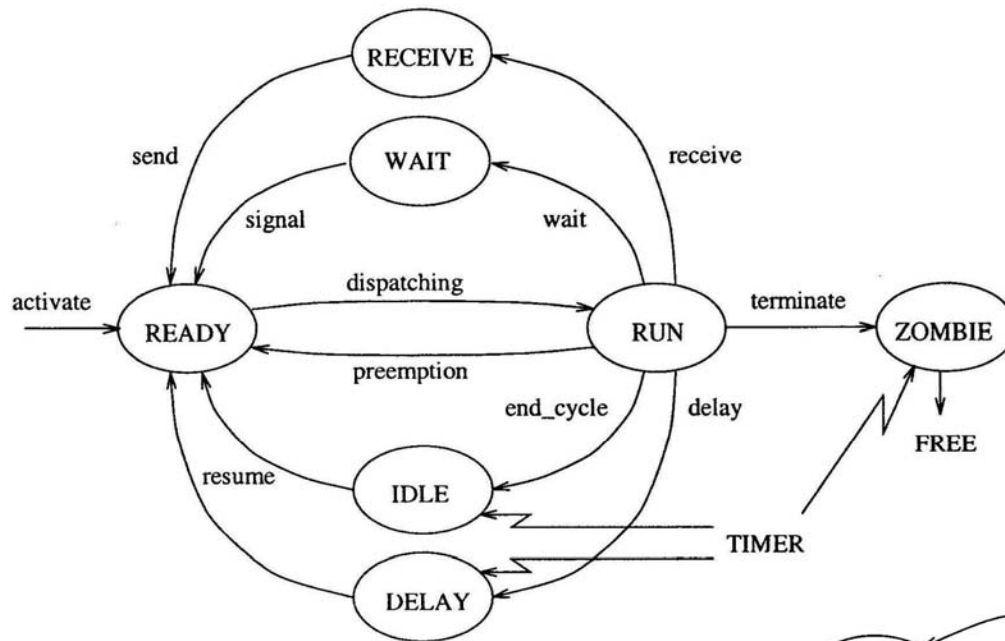


Entfernung eines  
 abgebrochene Tasks  
 und **sofortiges**  
**Einfügen** eines  
 neuen kann zu  
**Überschreitung der**  
**Deadline** führen



Entfernung eines  
 abgebrochene Tasks  
 und Einfügen eines  
 neuen **am Ende**  
**einer Ausführungs-**  
**periode** ermöglicht  
 erst die **garantierte**  
**Ausführbarkeit**

## Scheduling (3) vollständigere Beschreibung



**Delay** für verzögerte Tasks

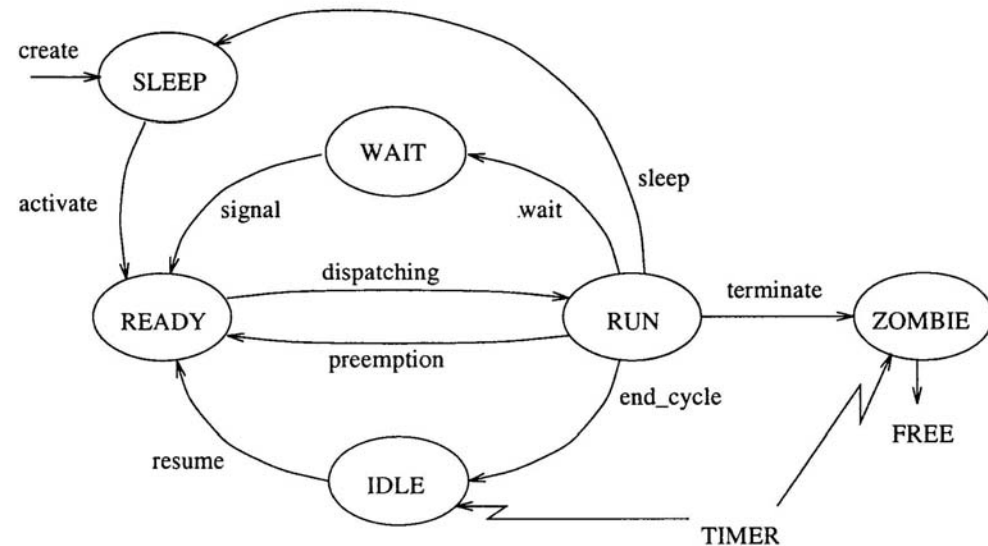
**Receive.** Task wartet auf Nachricht

**Zombie.** Erst nach Beendigung der Periode wird Datenstruktur deallokiert und Task verlässt vollständig das System

**Aperiodische Tasks** werden mit *activate* und *sleep* gehandhabt.

*create* erzeugt alle Datenstrukturen

Vereinfachte Darstellung für weitere Betrachtung →



## Task Control Block

task identifier
task address
task type
criticality
priority
state
computation time
period
relative deadline
absolute deadline
utilization factor
context pointer
precedence pointer
resource pointer
pointer to the next TCB

```

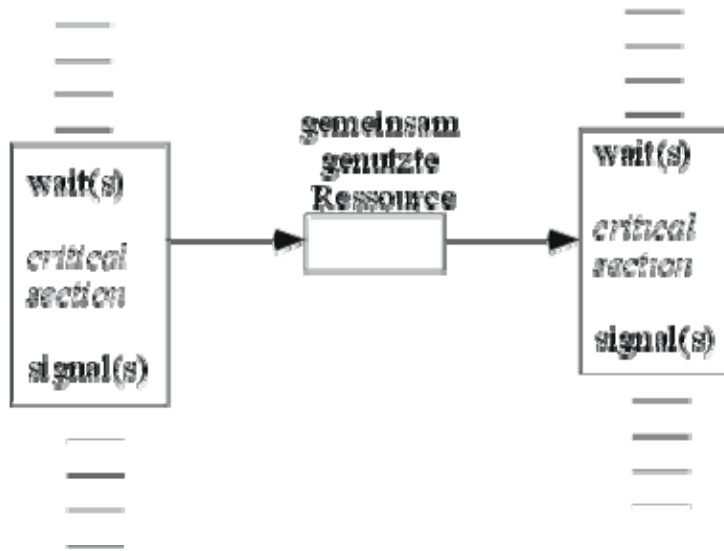
typedef int    queue;           /* head index    */
typedef int    sem;            /* semaphore index */
typedef int    proc;           /* process index  */
typedef int    cab;           /* cab buffer index */
typedef char*  pointer;        /* memory pointer */
  
```

```

struct tcb {
    char    name[MAXLEN+1];    /* task name      */
    proc    (*addr)();         /* first instruction address */
    int     type;              /* task type      */
    int     state;             /* task state     */
    long    dline;             /* absolute deadline */
    int     period;            /* task period    */
    int     prt;               /* task priority  */
    int     wcet;              /* worst-case execution time */
    float   util;              /* task utilization factor */
    int     *context;          /* pointer to the context */
    proc    next;              /* pointer to the next tcb */
    proc    prev;              /* pointer to previous tcb */
};
  
```

# Datenstrukturen

Zugriff auf gemeinsam genutzte Ressourcen erfolgt über Semaphore.



```

struct scb {
    int    count;           /* semaphore counter    */
    queue  qsem;           /* semaphore queue      */
    sem    next;           /* pointer to the next  */
};
  
```

Listen und  
Listen-  
verwaltung

```

struct tcb  vdes[MAXPROC];      /* tcb array */
struct scb  vsem[MAXSEM];      /* scb array */
  
```

```

proc    pexe;                  /* task in execution */
queue   ready;                 /* ready queue */
queue   idle;                  /* idle queue */
queue   zombie;               /* zombie queue */
queue   freetcb;              /* queue of free tcb's */
queue   freesem;              /* queue of free semaphores */
float   util_fact;            /* utilization factor */
  
```

# Zeitmanagement

```

unsigned long   sys_clock;           /* system time           */
float           time_unit;          /* unit of time (ms)     */
  
```

<i>tick</i>	<i>lifetime</i>
1 ms	50 days
5 ms	8 months
10 ms	16 months
50 ms	7 years

***sys\_clock*** ist die Anzahl *timer interrupts* während der Lebensdauer des Systems.

Kleine *time\_unit*

→ gute Aktivierungsrate des Systems

→ erhöht Systemoverhead

Optimal: GGT aller Task-Perioden

***Harte*** Tasks können *periodisch* oder *aperiodisch* sein → EDF

Nicht-Echtzeit-Tasks (NRT) im Hintergrund ausgeführt → FPS

Vermeidung zweier Queues, Transformierung von NRT, so dass deren Deadlines immer größer als die der HARD-Tasks sind:

$d_i^{\text{NRT}} = \text{MAXDEADLINE} - \text{PRT\_LEV} + P_i$ , wobei

$\text{MAXDEADLINE} = 2^{31} - 1$ ;

PRT\_LEV: Anzahl der Prio-Level des Kernel;

$P_i \in [0, \text{PRT\_LEV} - 1]$

# Initialisierung des Systems

```
void    ini_system(float tick)
{
    proc    i;

    time_unit = tick;
    <enable the timer to interrupt every time_unit>
    <initialize the interrupt vector table>

    /* initialize the list of free TCBs and semaphores */
    for (i=0; i<MAXPROC-1; i++) vdes[i].next = i+1;
    vdes[MAXPROC-1].next = NIL;
    for (i=0; i<MAXSEM-1; i++) vsem[i].next = i+1;
    vsem[MAXSEM-1].next = NIL;

    ready = NIL;
    idle = NIL;
    zombie = NIL;

    freetcb = 0;
    freesem = 0;
    util_fact = 0;

    <initialize the TCB of the main process>
    pexe = <main index>;
}
```

# Erzeugen eines neuen Task

```
/*-----*/  
/* create -- creates a task and puts it in sleep state */  
/*-----*/  
proc    create(  
    char    name[MAXLEN+1],    /* task name      */  
    proc    (*addr)(),         /* task address   */  
    int     type,              /* type (HARD, NRT) */  
    float   period,           /* period or priority */  
    float   wcet)             /* execution time  */  
{  
    proc    p;  
    <disable cpu interrupts>  
    p = getfirst(&freetcb);  
    if (p == NIL) abort(NO_TCB);  
    if (vdes[p].type == HARD)  
        if (!guarantee(p)) return(NO_GUARANTEE);  
    vdes[p].name = name;  
    vdes[p].addr = addr;  
    vdes[p].type = type;  
    vdes[p].state = SLEEP;  
    vdes[p].period = (int)(period / time_unit);  
    vdes[p].wcet = (int)(wcet / time_unit);  
    vdes[p].util = wcet / period;  
    vdes[p].prt = (int)period;  
    vdes[p].dline = MAX_LONG + (long)(period - PRT_LEV);  
    <initialize process stack>  
    <enable cpu interrupts>  
    return(p);  
}
```

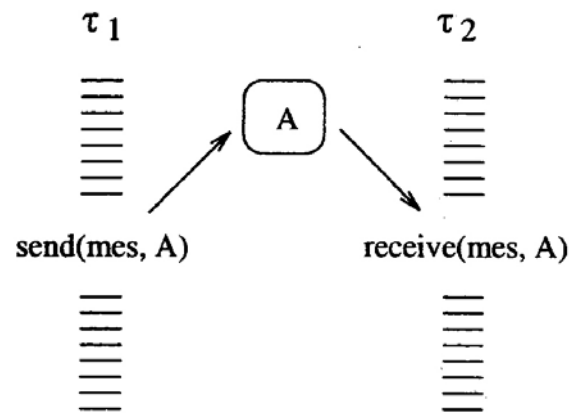
# Intertask Kommunikation

- Intertask Kommunikation ist ein kritischer Aspekt von Realtime Systemen
- Benutzung von shared Ressourcen für die Nachrichtenvermittlung kann
  - Prioritätsinversion
  - unbegrenzte Blockierung von Tasks zur Folge haben
- Ansätze
  - synchrone Modelle
  - asynchrone Modelle
- synchrone Modelle können zu unvorhersagbarem Verhalten in Echtzeitsystemen führen
  - Mögliche Lösungsansätze:
    - Überführung von synchronen Interaktionen in Vorgänger-Nachfolger-Beziehungen
    - asynchrone Modelle

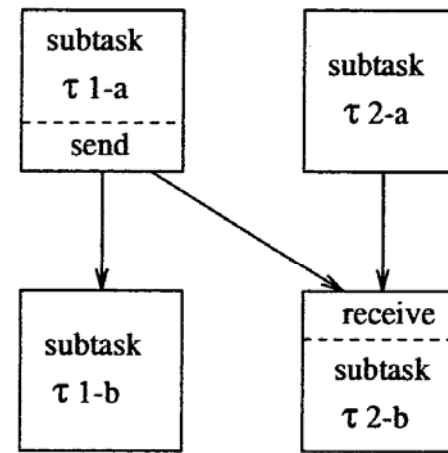
# Synchrone Task-Kommunikation

## „Rendez-vous“

- Sender und Empfänger müssen sich synchronisieren, d.h. sich zeitlich treffen (rendez-vous)
  - startet der Sender zuerst, muss er warten, bis der Empfänger die Nachricht abholt
  - startet der Empfänger zuerst, muss er warten, bis der Sender die Nachricht sendet
- Probleminderung:
  - Zerlegung von Sende- und Empfangstasks in Subtasks
  - senden nur am Ende eines Tasks-Abschnitts
  - empfangen nur am Anfang eines Tasks-Abschnitts

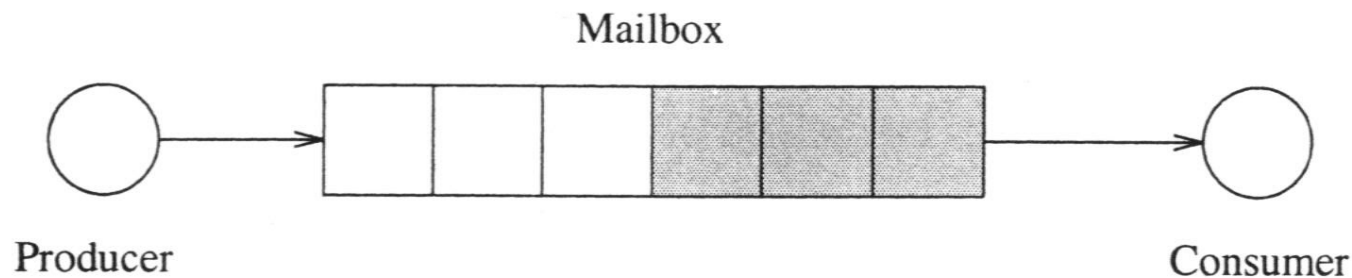


(a)



(b)

# Asynchrone Task-Kommunikation



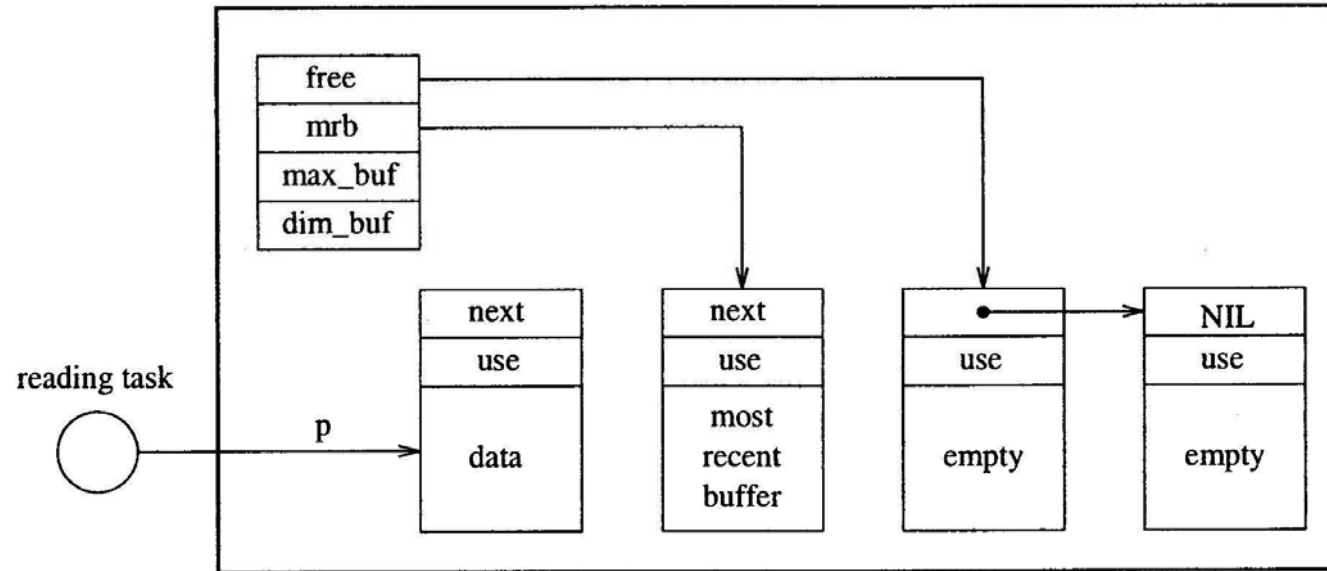
## 1. Mailbox

- wird in dem meisten kommerziellen Echtzeitsystemen verwendet
- besteht aus einem **shared memory** Puffer, der eine feste Anzahl Plätze für Nachrichten vorsieht und nach **FIFO** Prinzip arbeitet
- Problem:
  - ist die Mailbox **voll**, wird der **Sender blockiert**
  - ist die Mailbox **leer**, wird der **Empfänger blockiert**
  - wenn sendender Task eine Periodendauer von  $T_1$  hat und empfangender Task eine Periodendauer von  $T_2$  tritt bei  $T_1 \neq T_2$  irgendwann eine **Blockierung** ein

## 2. **Cyclical Asynchronous Buffer** (CAB) (Clark 1989)

- Entworfen für Kooperation von periodischen Aktivitäten wie Kontrollschleifen und Sensor-Datenaufnahme
- Stellt einen one-to-many Kommunikationskanal dar
- Eine Nachricht wird nicht verbraucht (d.h. entfernt) sondern durch eine neuere überschrieben.
- Sender kann niemals blockieren
- Sobald eine Nachricht im CAB ist, kann eine Empfänger nicht mehr blockieren
- Nachricht kann mehrfach gelesen werden
- Eine nicht rechtzeitig abgeholte Nachricht kann verloren gehen, d.h. durch eine neuere ersetzt werden.
  - → nicht geeignet Nachrichten Historie
  - → meist gut geeignet für Kontrollanwendungen (bei denen in der Regel nur aktuelle Daten von Interesse sind)

# CAB Implementierung



**Schreiben** einer Nachricht:

```

buf_ptr = reserve( cab_id );
<copy message in buf_ptr>
putmes(buf_ptr, cab_id);
  
```

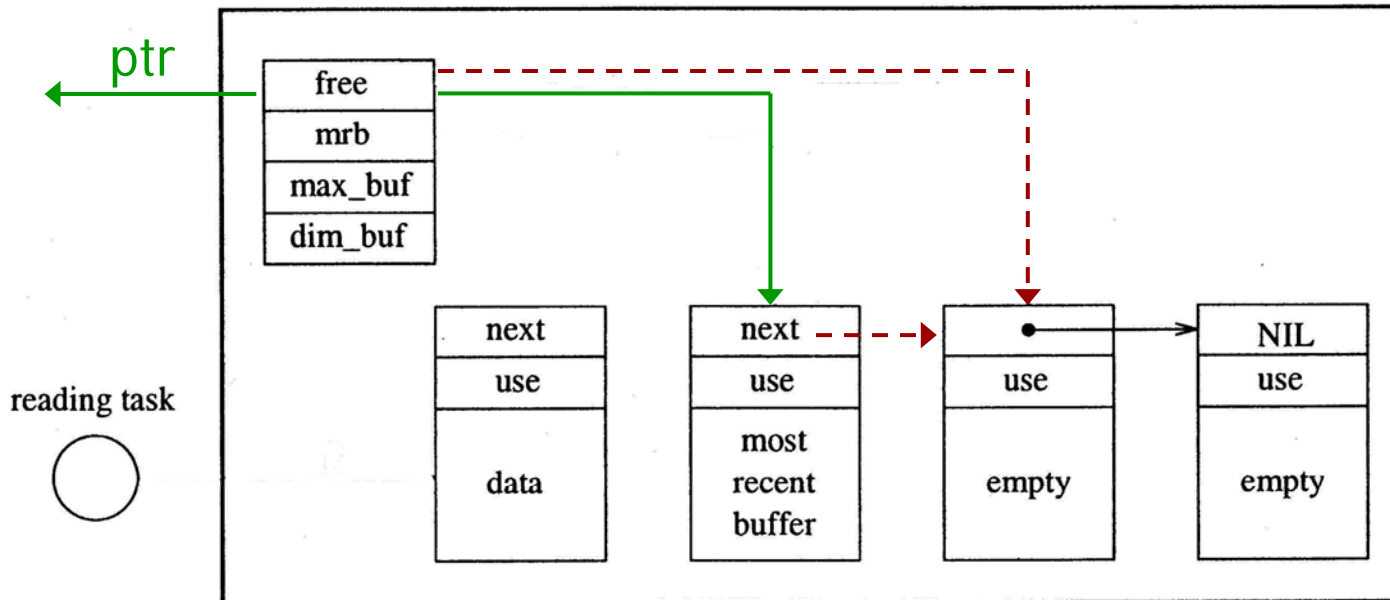
**Lesen** einer Nachricht:

```

mes_ptr = getmes( cab_id );
<use message >
unget(mes_ptr, cab_id);
  
```

**Funktionen** sind **nicht unterbrechbar** um Konsistenz und Integrität der Strukturen zu gewährleisten.

# CAB Implementierung (reserve)

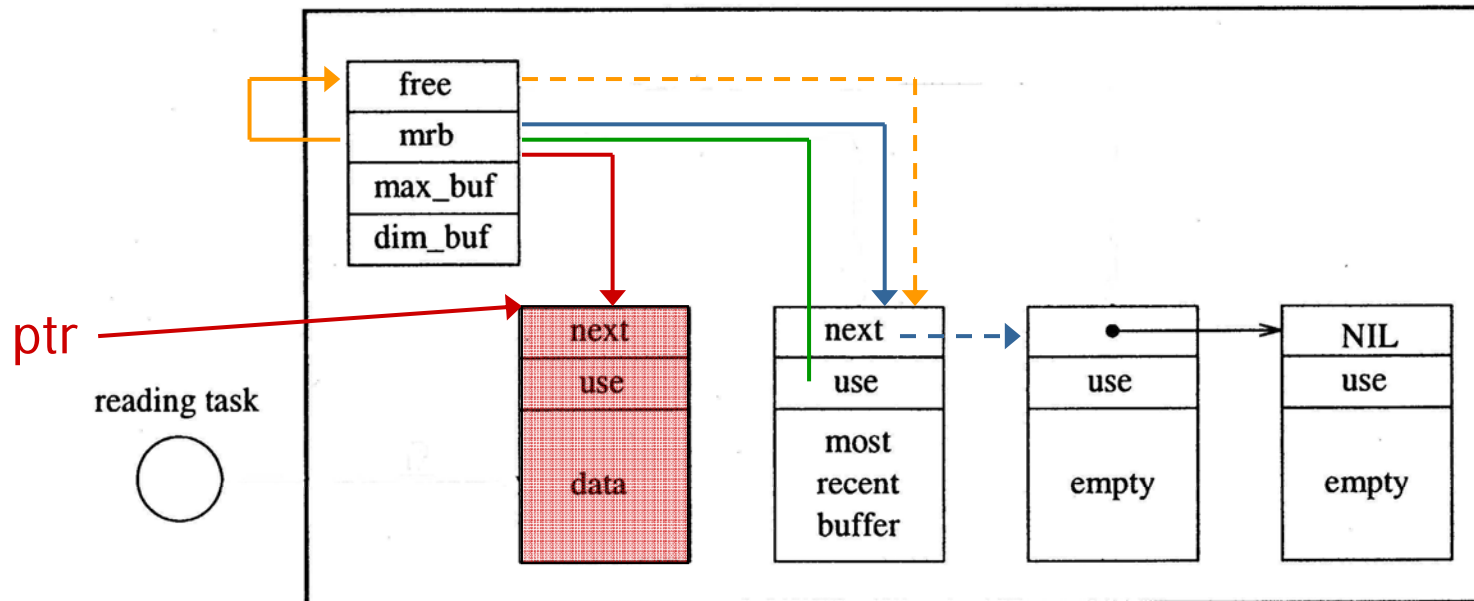


`ptr = reserve(CAB c);`

freier Buffer wird zurückgegeben, jetzt benutzt

CAB-Struktur zeigt mit `free` auf den nächsten freien Puffer

# CAB Implementierung (putmes)



```
void putmes(CAB c, pointer ptr);
```

```
wenn c.mrb.use == 0
```

```
  c.mrb.next = c.free;
```

```
  c.free = c.mrb;
```

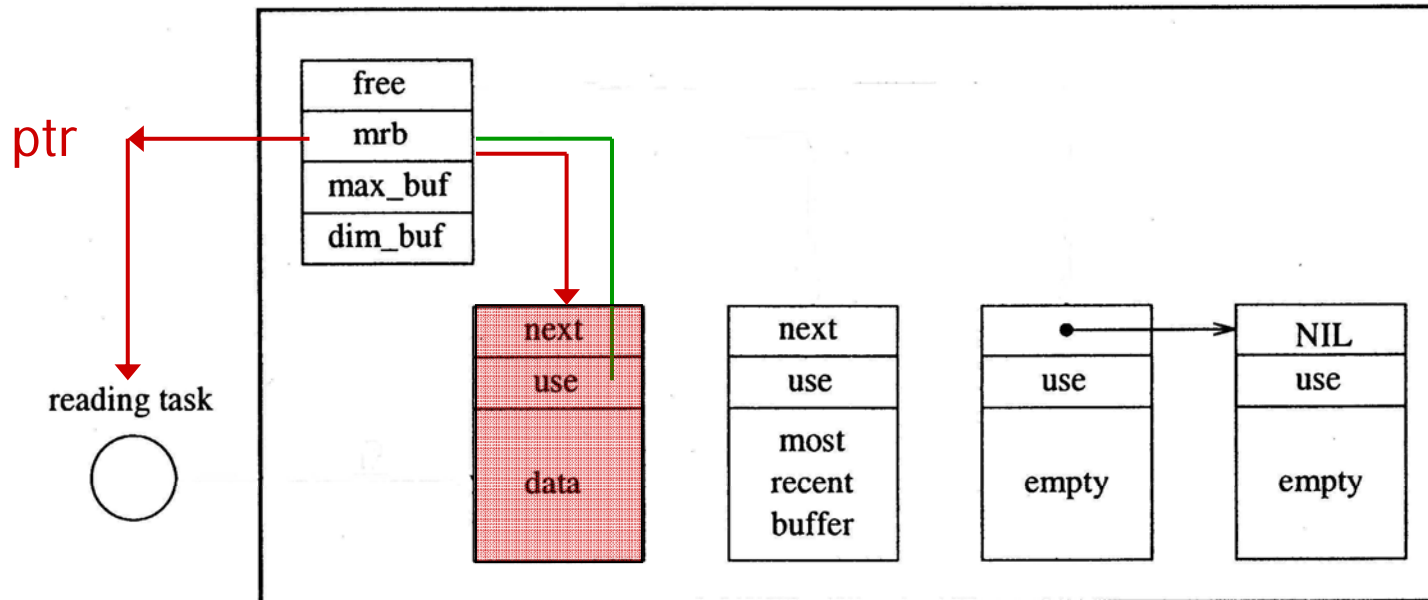
```
  /* if not accessed by a recipient */
```

```
  /* deallocate mrb */
```

```
  /* insert in free queue */
```

```
c.mrb = ptr;
```

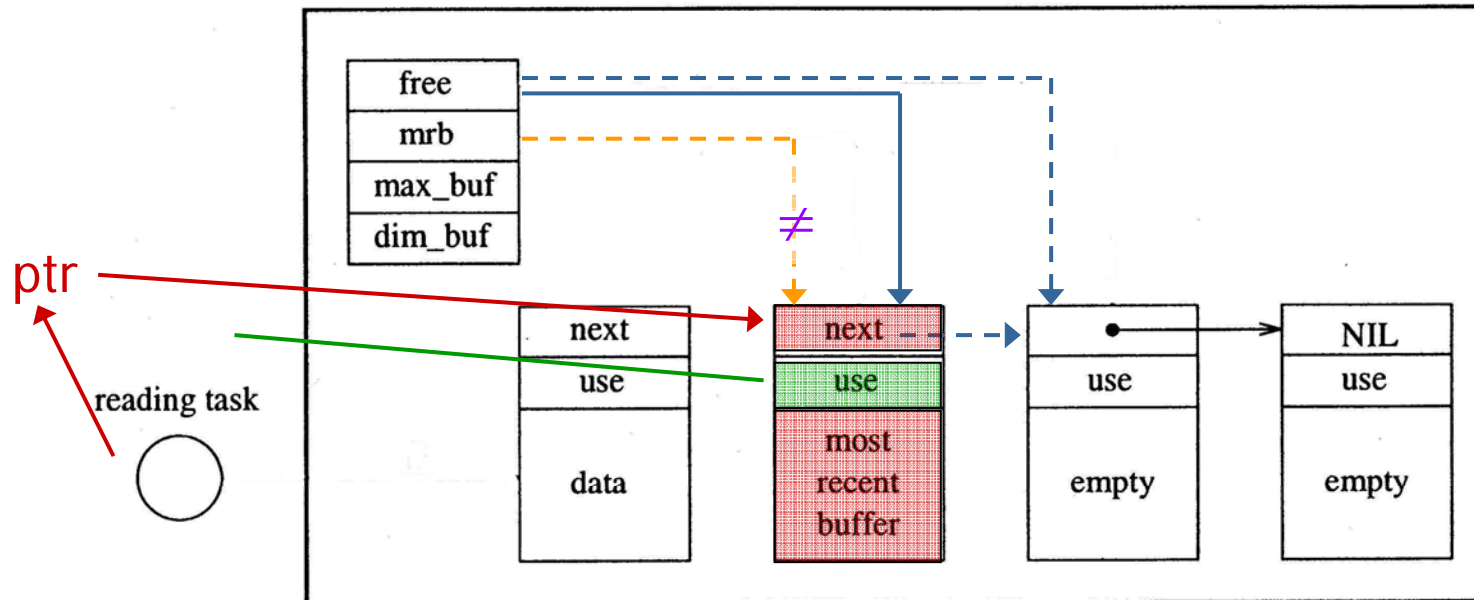
# CAB Implementierung (getmes)



```

ptr = getmes(CAB c);
pointer p = c.mrb;      /* get the most recent buffer */
p.use = p.use + 1;     /* increment the counter */
return(p);
  
```

# CAB Implementierung (unget)



```

void unget(CAB c, pointer ptr);
p.use = p.use - 1; /* decrement the counter */
if (p.use == 0) && (p != c.mrb) { /* if not accessed, not mrb */
    p.next = c.free; /* deallocate mrb */
    c.free = p; /* insert in free queue */
}
  
```

# System Overhead durch Betriebssystem

Das **Betriebssystem benötigt Rechenzeit** um alle Kernel-Aufgaben zu behandeln:

- Queuing von Tasks
- Kontext Wechsel
- Update interner Datenstrukturen
- Intertask Kommunikation: Senden und Empfangen von Nachrichten
- Bedienung von Interrupt-Anfragen

**Normalerweise** ist die Prozessorzeit für das OS gegenüber den anderen Tasks **vernachlässigbar** bezüglich Garantietests

In einigen Fällen jedoch, wenn **Anwendungstasks kleine Ausführungszeiten** haben und **enge Zeitbedingungen**, kann die Kernel-Aktivität nicht mehr vernachlässigt werden und **signifikante Interferenzen** erzeugen.

Vorhersagbarkeit kann dann nur noch durch **Betrachtung des Systemoverheads** in der Analyse der Schedulability erreicht werden.

# Overhead Faktoren

Einer der wichtigsten Beiträge zum Systemoverhead ist die **Context Switching** Zeit:

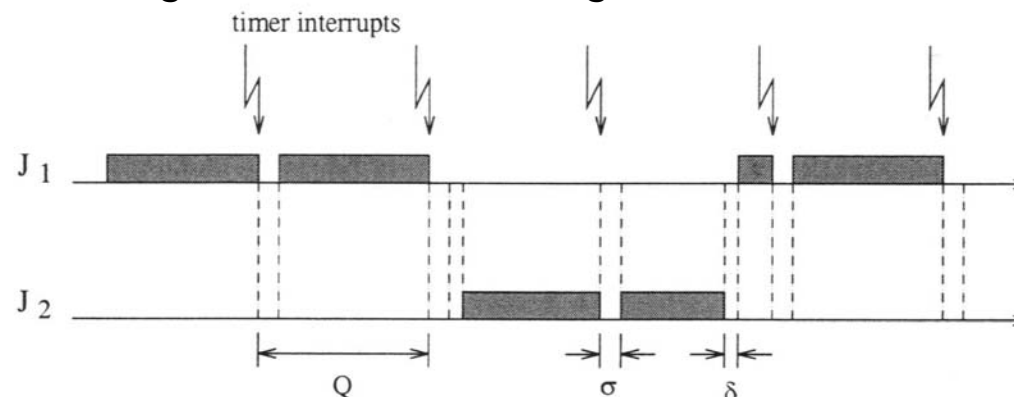
- ist intrinsische Grenze für den Kernel
- nicht abhängig vom Scheduling Algorithmus
- nicht abhängig von der Struktur der Tasks

Ein weiterer wichtiger Beitrag ist Prozessorzeit zur Behandlung der **Timer Interrupts**. Sei

- $Q$  der Systemtakt
- $\sigma$  die worst case Ausführungszeit der entsprechenden Treiber
- $\delta$  die Ausführungszeit eines Kontext Wechsels

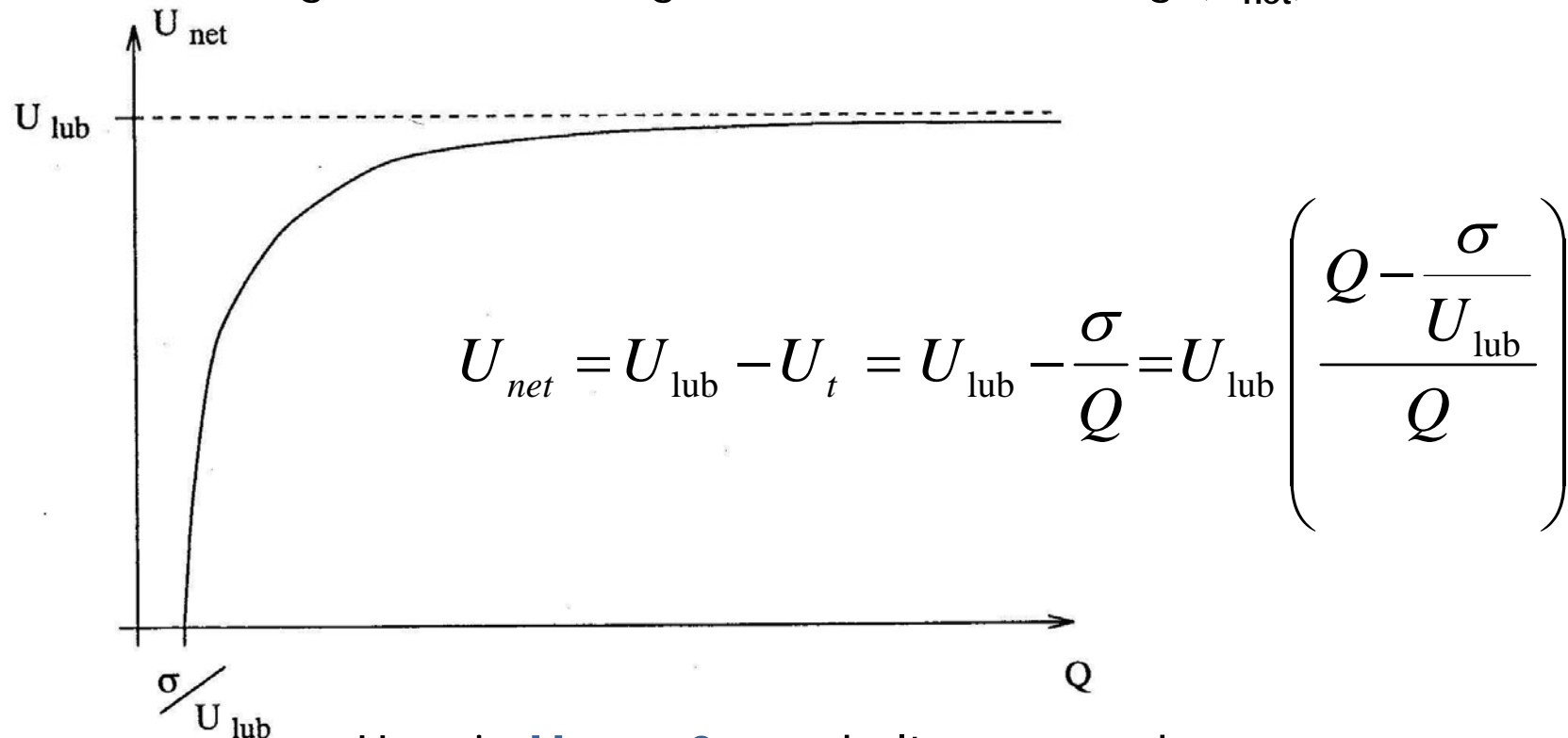
berechnet sich der Utilization Faktor  $U_t = \sigma/Q$

Dieser muss zum gesamten Auslastungsfaktor aller Tasks addiert werden.



# Auslastungs Faktor

Die Addition von  $U_t$  zum Auslastungsfaktor der übrigen Tasks ist gleichbedeutend mit der Reduktion des *Least Upper Bound* (lub) mit  $U_t$ . Daraus ergibt sich die mögliche Netto-Auslastung ( $U_{net}$ ) zu:



Um ein  $U_{net} > 0$  zu erhalten, muss der **Systemtakt  $Q$  stets größer als  $(\sigma/U_{lub})$**  sein

# Auslastungsfaktor Beispiel

Das folgende Beispiel gibt eine Einschätzung von der Leistungsminderung durch den System Overhead.

Ausgehend von einem EDF Scheduling Algorithmus und der Annahme, dass  $U_{lub} = 1$ , sowie die Ausführungszeit für die Timer Interrupt Routine eine Ausführungszeit von  $\sigma = 100 \mu\text{s}$  hat wird die Auswirkung der Größe des Systemtakts  $Q$  auf  $U_{net}$  betrachtet:

$$\begin{array}{lll} Q = 10 \text{ ms} & \rightarrow & U_{net} = 0.99 \\ Q = 1 \text{ ms} & \rightarrow & U_{net} = 0.90 \\ Q = 200 \mu\text{s} & \rightarrow & U_{net} = 0.5 \end{array}$$

Dies bedeutet, dass bei einem GGT aller Task-Perioden von  $200 \mu\text{s}$  in einem solchen System ein Utilization Faktor  $U = 0.6$  für ein solches Ensemble von Echtzeittasks **nicht mehr garantiert** werden könnte.

# Schedulability Analyse mit Overhead

Der Overhead durch Kernel-Mechanismen kann durch zusätzliche Terme in den Ausführungszeiten  $C_i$  der einzelnen Tasks  $\tau_i$  berücksichtigt werden.

Insbesondere die Zeit  $\delta$ , welche für Kontext Wechsel (context switches), beispielweise durch Systemaufrufe (system calls) wird auf diese Weise berücksichtigt.

Ebenso werden implizite Kontext Wechsel, welche vom Kernel veranlasst werden, dem verdrängten (preempted) Task in Rechnung gestellt.

In diesem Fall muss in der Analyse der Schedulability die Anzahl Preemptions  $N_i$  für jeden Task abgeschätzt werden. Dies kann meist offline aus den einzelnen Zeitbedingungen des Tasks  $\tau_i$  abgeschätzt werden.

$$N_i = \sum_{k=1}^{i-1} \left\lfloor \frac{T_i}{T_k} \right\rfloor.$$

(Diese Abschätzung der oberen Grenze für die Preemptions ist eher zu pessimistisch. Für spezielle Scheduling Algorithmen können bessere gefunden werden)

## Sched. Analyse mit Overhead (2)

- Ausführungszeiten  $C_i$  und Periodendauer  $T_i$  des einzelnen Tasks  $\tau_i$
- Zeit  $\delta$ , welche für Kontext Wechsel (context switches)
- Anzahl Preemptions  $N_i$
- Gesamt Utilization Faktor  $U_{tot}$  für den Task Set

$$U_{tot} = \sum_{i=1}^n \frac{C_i + \delta N_i}{T_i} + U_t = \sum_{i=1}^n \frac{C_i}{T_i} + \left( \delta \sum_{i=1}^n \frac{N_i}{T_i} + U_t \right).$$

$$U_{tot} = U_p + U_{ov},$$

- Utilization Faktor  $U_p$  für alle periodischen Tasks
- Korrekturfaktor  $U_{ov}$ , der die Zeit für Timer Handling und den Overhead für Preemption berücksichtigt.

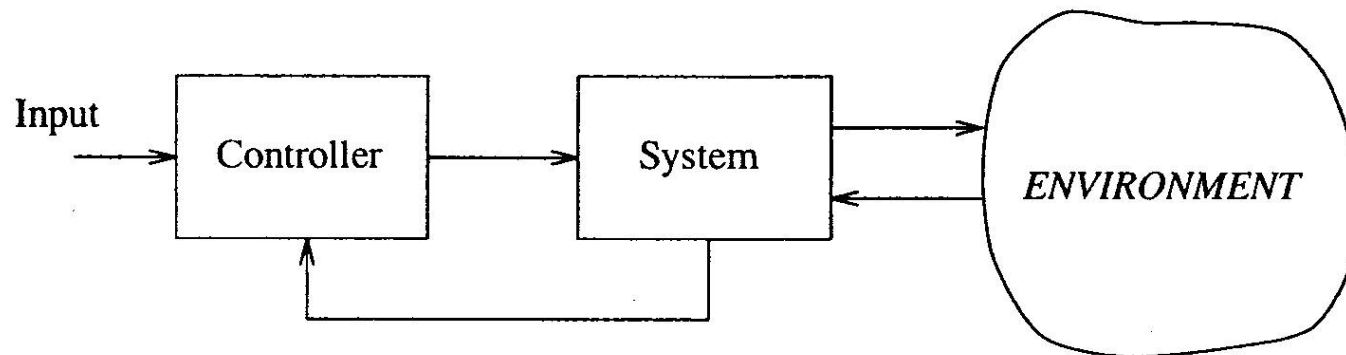
$$U_{ov} = U_t + \delta \sum_{i=1}^n \frac{N_i}{T_i}.$$

Weitere Betrachtungen für Interrupt  
 Behandlung siehe  
 → *Jeffay* und *Stone*, z.B. in [1]

# Design von Echtzeitsystemen

Ein Echtzeitsystem ist durch die folgenden Komponenten charakterisiert

- das zu kontrollierende **System**:  
z.B. eine Fabrik, ein Fahrzeug, ein Roboter oder ein Gerät, welches ein bestimmtes Verhalten realisieren soll
- die **Kontrolleinheit**:  
in unserem Fall ein Rechnersystem, welches dem zu kontrollierenden System richtige Werte für die gewünschten Steuerungs-Ziele liefert
- die **Umgebung**:  
die äußere Welt in der das System operiert



# Arten kontrollierter Systeme

---

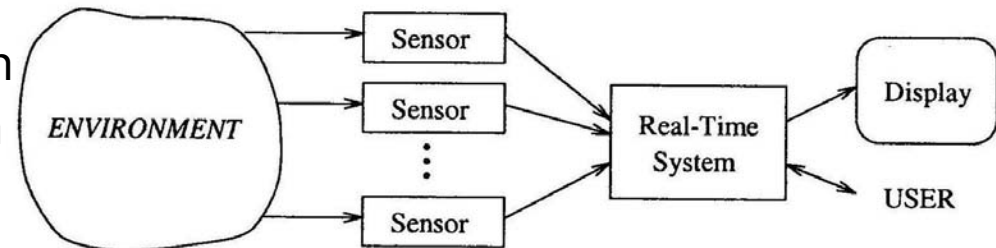
Allgemein kann man kontrollierte Systeme folgendermaßen untergliedern:

- **Monitor-Systeme** (monitoring systems)
- **offene Kontroll-Systeme** (open-loop control systems)
- **rückgekoppelte Kontroll-Systeme** (closed-loop control systems, feedback control systems)

# Monitoring Systeme

## Monitor Systeme

- verändern nicht ihre Umgebung
- Benutzen Sensor-Daten um
  - eigenen Status festzustellen
  - Sensordaten zu verarbeiten
  - Anzeige von Ergebnissen



## Typische Beispiele:

- Radar-Verfolgung
- Flugverkehrskontrolle (**Air Traffic Control**)
- Überwachung, Alarm (Umwelt, Medizin usw.)

Viele dieser Anwendung erfordern **periodische Erfassung** von mehreren Sensoren, möglicherweise mit **unterschiedlichen Sampling Raten**.

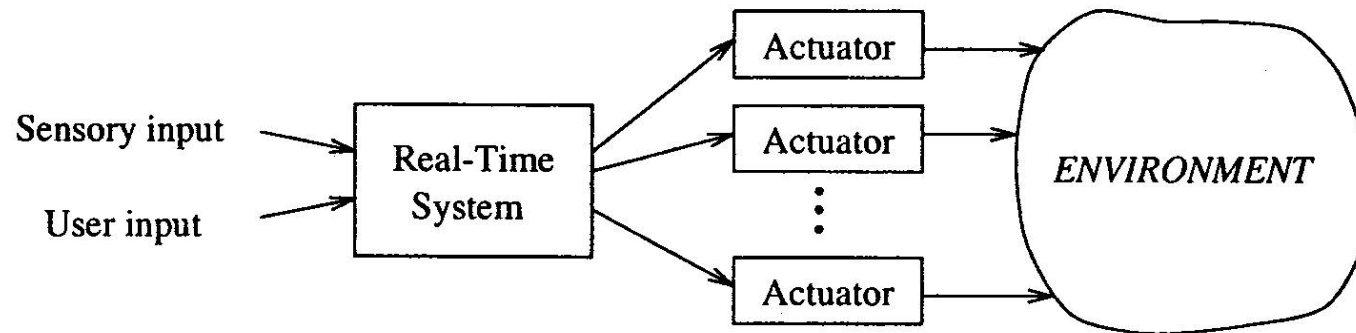
Da **kritische Bedingungen** erfasst werden müssen, ist die Verwendung eines **harten Echtzeitsystems** erforderlich.

Der Task Set kann **offline** analysiert werden um die Schedulability unter den vorgegebene Zeitbedingungen zu testen

# Offene Kontroll Systeme

Offene Kontroll-Systeme interagieren mit der Umgebung

- ausgeführte Aktionen hängen nicht strikt vom aktuellen Zustand der Umgebung ab.
- Sensor-Daten werden benutzt, um Aktionen zu planen
- Keine Rückkopplung (feedback) zwischen Sensor und Aktuator
- Aktionen können unabhängig von neueren Sensordaten ausgeführt werden.



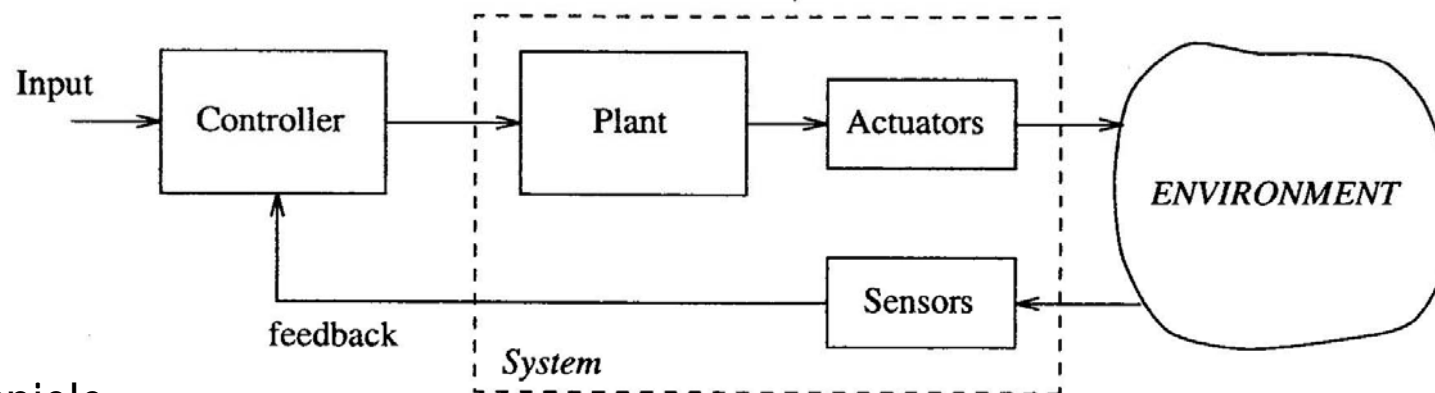
Beispiel:

Ein Roboter-System mit Kamera identifiziert ein Objekt und seinen Ort und sendet die Koordinaten an die Roboter-Steuerung. Diese berechnet die notwendige Aktion, um das Objekt aufzunehmen. Hierfür ist **kein Echtzeitsystem erforderlich**. Erfolg der Aktion hängt nur von der Rechengeschwindigkeit und der richtigen Berechnung der Bewegung ab.

# Rückgekoppelte Kontroll Systeme

Rückgekoppelte Kontroll-Systeme haben häufige Interaktionen mit der Umgebung in beide Richtungen

- ausgeführte Aktionen hängen strikt von neuen Sensor-Daten ab
- Sensor und Kontrolle hängen fest zusammen, oft mit mehreren Feedback-Pfaden
- Sensoren häufig direkt auf Aktuatoren montiert um Aktion direkt anhand der Sensordaten zu korrigieren.



Beispiele:

- Biologische Systeme (Mensch, Tiere)
- Moderne „fly-by-wire“ Flugzeuge  
Basisbefehle werden vom Piloten gegeben. Diese werden in eine Serie von Input-Daten umgewandelt. Das System berechnet anhand von Sensor-Daten Bewegung und Stellung der Steuerruder und die Triebwerksleistung

# Vorgehen beim Entwurf von RT-Systemen

Für kritische Echtzeitsysteme müssen, neben klassischen Designaspekten, folgende Punkte im Detail betrachtet werden:

- **Strukturierung** der Anwendung in eine Anzahl **konkurrierender Tasks**
- **Festlegung** der **korrekten Timing-Bedingungen** für jeden Task
- Wahl einer **geeigneten Betriebsumgebung**, welche erlaubt, **erforderliche Zeitbedingungen zu garantieren**
  
- Identifizierung von **harten** und **weichen Echtzeitbedingungen** (siehe 1. Teil der Vorlesung)
- **Analyse der Umgebung**, beispielsweise physikalische Gegebenheiten (wie beispielsweise in den Praktika durchgeführt)
- Spezifikation der **Sensoren** und **Aktuatoren** bezüglich
  - Reaktionsgeschwindigkeit, Verzögerungen etc.
  - Umgebungsbedingungen (Temperaturbereich, Feuchtigkeit, ...)
  - Ausfallsicherheit
  - ...

# Konkrete Entscheidungen

- Wahl eines Prozessors
- Platine
  - Eigenentwicklung der Platine
  - Evaluation Board
- alternativ: kommerzielles Produkt mit dem Wunschprozessor (PDA), dann HW verstehen und abändern, eigenes OS aufspielen ...
- Peripherie: Anpassung des digitalen/analogen I/O:
  - diskrete Digitalbausteine
  - FPGA (Field Programmable Gate Array - einzeln rekonfigurierbarer Logik-Chip)
  - CPLD (CMOS Programmable Logic Device)
  - spezifische Chipsätze (GPS, WLAN, Bild-/Tonverarbeitung, DFT, div. Algorithmen)
- Entwicklungsumgebung:
  - RT-OS
  - Compiler/Linker/Debugger (GNU, insight, ...)
  - JTAG, on-chip debugging
  - Versionsverwaltung (CVS, ...)
  - Dokumentationshilfen (doxygen, ...)
- Laborausstattung
  - Oszilloskop
  - Multimeter
  - Lötkolben

- [1] Hard Real-Time Computing Systems; Giorgio C. Buttazzi; Springer; 2005
- [2] Real-Time Systems and Programming Languages; Alan Burns, Andy Wellings; Pearson-Addison Wesley; 3rd Ed 2001
- [3] Real-Time Design Patterns, *Robust Scalable Architecture for Real-Time-Systems*; Bruce Powel Douglas; Addison-Wesley; 2003
- [4] Embedded System Design on a Shoestring, *Achieving High Performance with a Limited Budget*; Lewin A.R.W. Edwards; Newnes; 2003