Vorlesung Real-Time Systems

6. Echtzeitbetriebssysteme

Übersicht

1. Ziel dieses Kapitels
2. Definition
3. Anforderungen an ein Echtzeitbetriebssystem (RTOS)
4. Unterschiede RTOS – Standard-OS
5. Standards
6. Elemente eines RTOS
7. System Overhead
8. Zusammenfassung
9. Case Study

1. Ziel dieses Kapitels

Ein **Echtzeitbetriebssystem (Real-Time Operating System – RTOS)** hat zunächst die gleichen Aufgaben wie ein Standard-Betriebssystem. Darüber hinaus muss es aber spezielle Anforderungen erfüllen und spezielle Dienste liefern, um die Echtzeitfähigkeit zu gewährleisten.

In diesem Kapitel werden die wesentlichen Unterschiede eines RTOS zu einem Standard-OS beschrieben. Die charakteristischen Konzepte eines RTOS werden vorgestellt.

"Sie müssen das System neu starten, damit Ihre Änderungen wirksam werden"...

2. Definition

Definition RTOS:

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 Systemverhaltens.

Echtzeitbetriebsyssteme sind in der Regel deutlich kompakter als Standard-Betriebssysteme und kommen häufig in eingebetteten Systemen zum Einsatz.

3. Anforderungen an ein RTOS

RTOS unterliegen anderen Anforderungen als Standardbetriebssysteme:

- stabiler Betrieb rund um die Uhr
  - u.U. jahrelang
- definierte Reaktionszeiten
- effizienter Prozesswechsel (geringer Prozesskontext)
- Interruptbehandlung
- echtzeitfähiges Scheduling
- Prozesssynchronisation
  - Semaphore, Mutex, etc.
- echtzeitfähige Prozesskommunikation
- Zeitdienste
  - absolute, relative Uhren, Timeouts, Weckdienste, etc.
- einfaches Speichermanagement

3. Anforderungen an ein RTOS

- Unterstützung für Ein-/Ausgabe
  - z.B. Zugriff auf Hardware-Register
- Unterstützung für Buskommunikation
  - z.B. zum Senden/Empfangen von CAN-Botschaften
- Konfigurierbarkeit
  - z.B. Allokierung von Speicherbereichen, Scheduling Algorithmus
- Skalierbarkeit
  - optionale Komponenten sollten nur bei Bedarf implementiert werden
- minimaler Speicherbedarf

RTOS stehen oft in Form von Sourcecode zur Verfügung, der, abhängig von der gegebenen Konfiguration, teilweise generiert / parametrisiert und dann zusammen mit der Applikation übersetzt und gelinkt wird.

4. Unterschiede RTOS – Standard-OS

Warum kann ein Standard-OS in der Regel nicht für Echtzeitanwendungen verwendet werden?

- Existenz von Monitor und Tastatur (Maus) werden vorausgesetzt
  - bei eingebetteten Systemen meist nicht vorhanden
- Hintergrundspeicher wird vorausgesetzt
  - bei eingebetteten Systemen meist nicht vorhanden
- Das System zeigt nichtdeterministisches Verhalten
  - z.B. wegen Verwendung von Puffern, Fragmentierung von Dateien und Speicher
- ungeeignete Scheduling-Strategien
  - z.B. nur "Round Robin"
- „Timing“-Services ungeeignet
  - zu geringe Auflösung, Schwankungen, ...
- viel zu hohe Hardware-Anforderungen (Prozessor, Speicher, ...)
  - auch selten/nie benötigte Dienstprogramme und Bibliotheken immer vorhanden
- viel zu hohe Lizenzkosten (für die hohen Stückzahlen eingebetteter Systeme)

4. Unterschiede RTOS – Standard-OS

Ziele eines Standard-OS:

- alle Prozesse werden (weitgehend) gleich (fair) behandelt
- der Gesamtdurchsatz des Systems wird optimiert
- weniger Prozesswechsel und lange kritische Bereiche
- blockweise Ein-/Ausgabe, Sammlung "gleicher Aufträge"

Ziele eines RTOS:

- einige Prozesse sind wichtiger als andere und haben höhere Priorität
- kritische Prozesse verdrängen weniger wichtige
- vorhersagbare Laufzeiten sind wichtiger als optimaler Gesamtdurchsatz
- kurze kritische Bereiche
- sofortige Bearbeitung anstehender Aufträge (Ein-/Ausgabe)

5. RTOS Standards

Es gibt kein "typisches" Echtzeitbetriebssystem, da die Anforderungen je nach Einsatzbereich sehr unterschiedlich sind. Anders als bei PC-Betriebssystemen haben sich bislang nicht ein oder zwei RTOS-Produkte durchgesetzt.

Es existieren einige Derivate, z.B.

- RT-Linux (http://www.rtlinux.org)
- RT-Java (kein RTOS im eigentlichen Sinne)
- Windows CE für eingebettete Systeme

Außerdem viele proprietäre Lösungen wie QNX, LynxOS, VxWorks, ...

Dazu gibt es die folgenden **Standards**:

- POSIX [(1)] (ISO/IEC 9945-1:1996): an UNIX angelehnter Standard für Betriebssystem-Schnittstellen mit Ergänzungen für Echtzeit.
- OSEK [(2)] /VDX: „de facto"- Standard der Automobilindustrie. Zielt auf minimale Betriebssystemkerne
- AUTOSAR: basierend auf OSEK, mit einigen Erweiterungen

(1) POSIX: Portables Operations System fuer unIX

(2) OSEK: Offene Systeme und deren Schnittstelle fuer die Elektronik im Kraftfahrzeug

5. RTOS Standards

POSIX = Portable Operating System Interface
wurde durch Arbeitsgruppen in der IEEE definiert und durch ANSI und ISO standardisiert. POSIX hat seine Ursprünge in der UNIX-Welt und soll die Portierung von (stark) betriebssystemabhängigen Programmen vereinfachen.

Der POSIX 1003.1 Standard = ISO/IEC 9945-1:1996 = ANSI/IEEE Std. 1003.1 besteht aus mehreren Teilen. Betriebssysteme unterstützen oft nur Teilmengen des gesamten POSIX-Standards. Es besteht die Möglichkeit, ein Betriebssystem zertifizieren zu lassen. (QNX und VxWorks sind z.B. POSIX-konform).

Bestandteile von POSIX:
asynchr./synchr. I/O, shared memory, RT signals, message queues, priority scheduling, memory locking, semaphores, timers, ...

- POSIX 1003.1 - 1990: Basisdienste eines Betriebssystems
- POSIX 1003.1b - 1993: Real-Time-Erweiterungen
- POSIX 1003.1c - 1995: Threads
- POSIX 1003.1d - 1999: weitere RT-Erweiterungen
- POSIX 1003.1j - 2000: weitere RT-Erweiterungen
- POSIX 1003.1q - 2000: Tracing, Ausführungszeitmessung, ...

Typisches C POSIX Interface:

```
typedef ... pthread_t;  /* details not defined */
typedef ... pthread_attr_t;

int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);

int pthread_attr_setstacksize(..);
int pthread_attr_getstacksize(..);

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
       void *(*start_routine)(void *), void *arg);
  /* create thread and call the start_routine with the argument */

int pthread_join(pthread_t thread, void **value_ptr);
int pthread_exit(void *value_ptr);
  /* terminate the calling thread and make the pointer value_ptr
     available to any joining thread */

pthread_t pthread_self(void);
```

All functions return 0 if successful, otherwise an error number

5. RTOS Standards

OSEK:

- "Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug"
- ist ein Gemeinschaftsprojekt der deutschen Automobilindustrie (u.a. BMW, DaimlerChrysler, VW, Opel, Bosch, Siemens)

VDX:

- "Vehicle Distributed eXecutive"
- PSA, Renault

Der Standard ist auf Skalierbarkeit und Minimalität (8 Bit bis 32 Bit Mikroprozessen) zugeschnitten; er besteht aus drei Teilen:

- **OS**: Operating System: Schnittstellen für Tasks, Scheduling, Alarme, Ereignisse, Ressourcen, Interrupts, ...
- **COM**: Communication: Schnittstellen für Nachrichtenübertragung, Übertragungsmodi, Monitoring, Message Queues, ... zwischen Prozessen und Prozessoren
- **NM**: Network Management standard: Verwaltung eines Netzwerkes, Modi für Erkennung/Behandlung von Ausfällen, ...

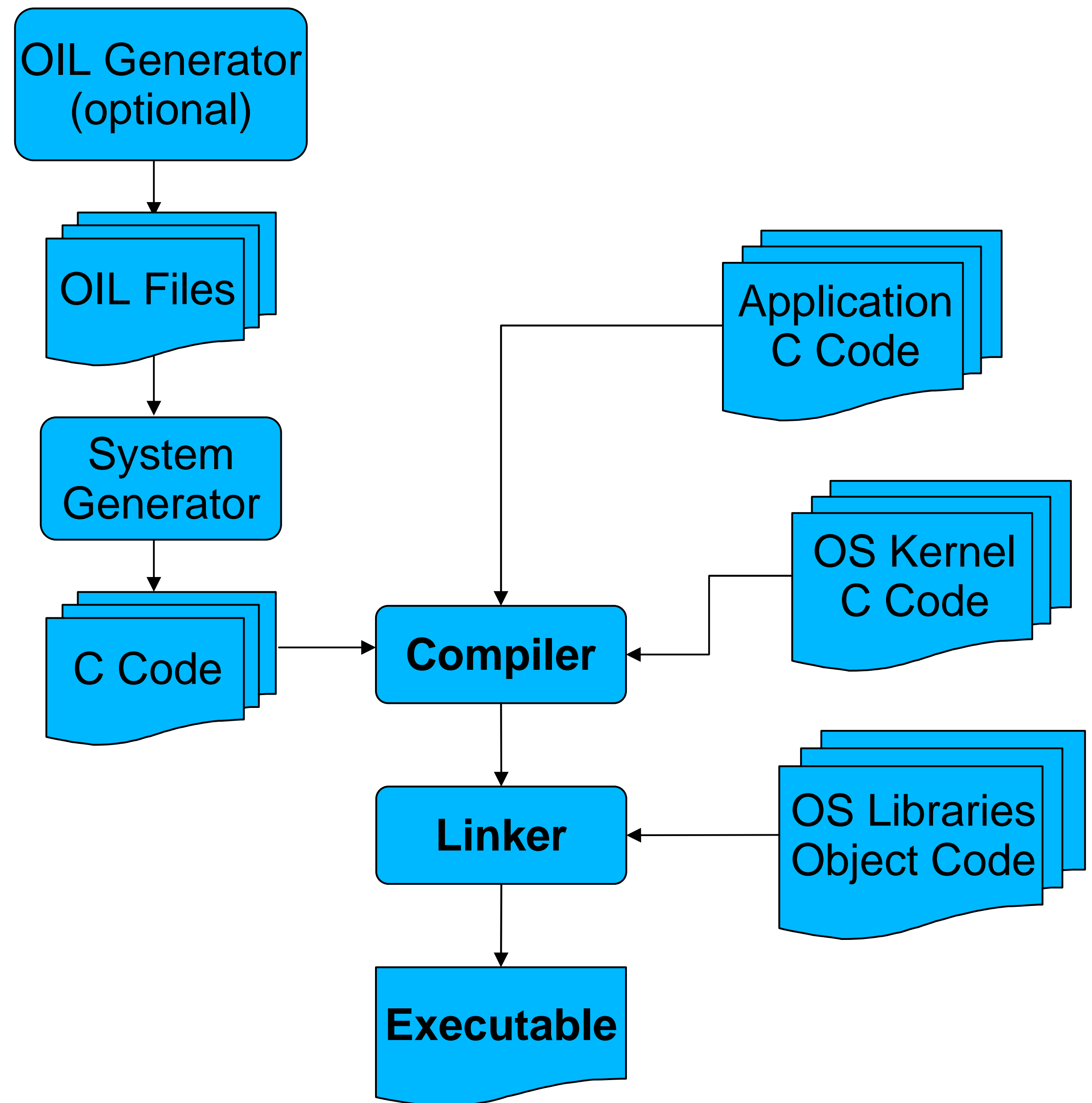
Außerdem eine Konfigurationssprache:

- **OIL** = OSEK Implementation Language: für die Beschreibung der Betriebssystem- und Prozesskonfigurationen; das Betriebssystem wird daraus generiert

5. RTOS Standards

Beispiel OIL File (Auszug):

```
CPU ExampleCPU {
     OS MyOs {
     ...
     };

     TASK MyTask1 {
          PRIORITY = 17;
          ...
     };

     TASK MyTask1 {
          StackSize = 64;
          ...
     };

     ALARM MyAlarm1 {
          ACTION = ACTIVATETASK {
               TASK = MyTask1;
          };
          START = TRUE;
          ...
     };

     MESSAGE MyMsg1 {
          ITEMTYPE = "SensorData";
          ...
     };

     ISR MyIsr1 {
          RCV_MESSAGES = MyMsg1;
          RCV_MESSAGES = MyMsg2;
          ...
     };
}; // End CPU ExampleCPU
```



5. RTOS Standards

Im OIL-File kann definiert werden, ob eine Task unterbrechbar ist oder nicht. OSEK unterstützt nur statische Prioritäten.

OSEK unterscheidet 2 Klassen von Tasks:
- Basic Tasks: können sich nicht selbst blockieren
- Extended Tasks: können blockieren (mittels `wait()` Anweisung)

Der OSEK Standard ist frei verfügbar (www.osek-vdx.org), Implementierungen sind aber kommerziell.

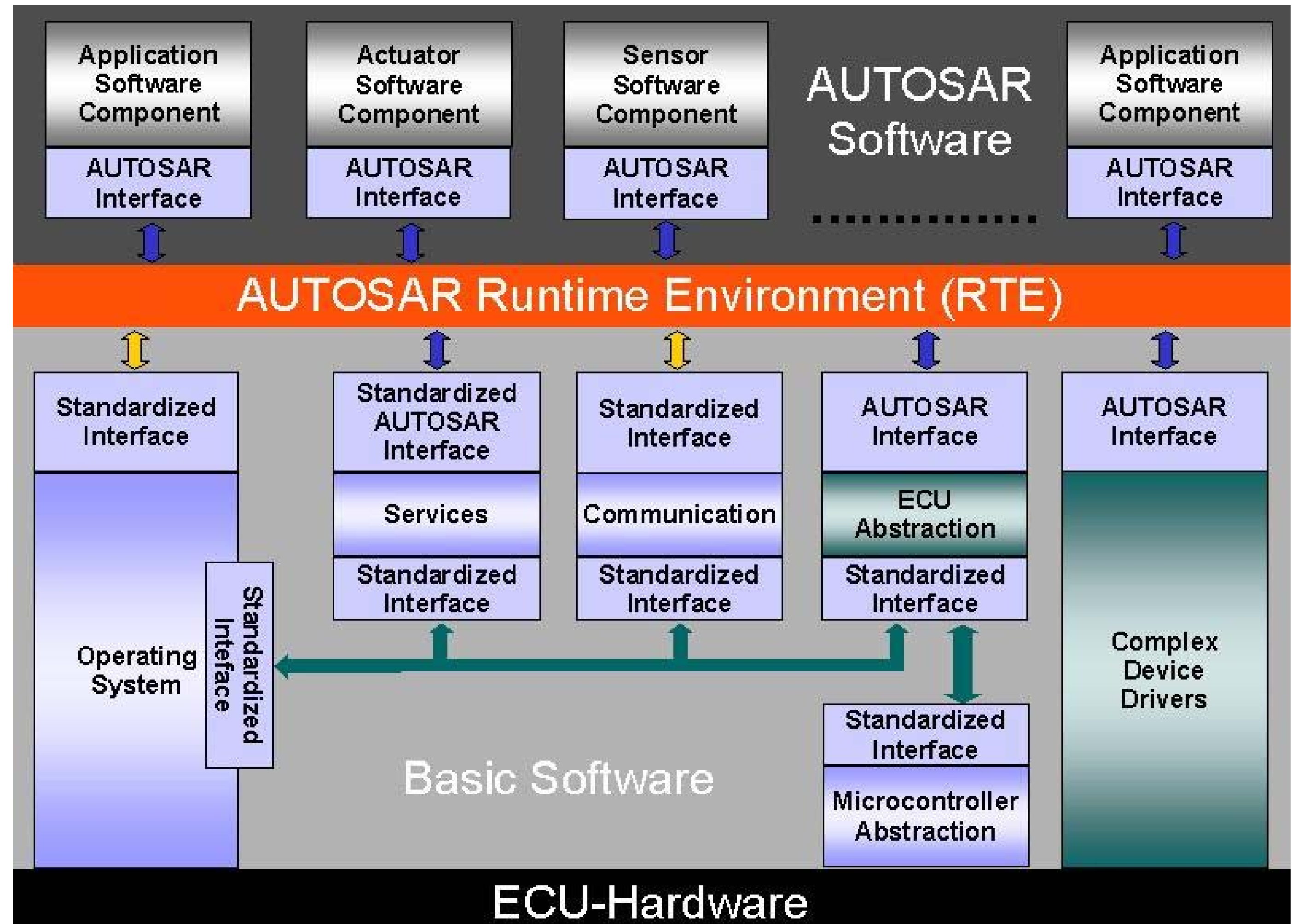
Es gibt auch einige Initiativen für eine freie Implementierung des Standards:
- www.openosek.org
- trampoline.rts-software.org
- opensek.sourceforge.net (FreeeOSEK)

Es existieren ebenso Ansätze für ein zeitgesteuertes Betriebssystem (OSEK-Time), sowie eine fehlertolerante Kommunikationsschicht.

5. RTOS Standards

AUTOSAR = AUTomotive Open System ARchitecture ist eine Initiative von inzwischen über 100 Firmen der Automobilindustrie zur Standardisierung der Softwarearchitektur für Steuergeräte.

AUTOSAR definiert – unter anderem – ein Standard-API für ein Betriebssystem: **AUTOSAR-OS**.



AUTOSAR-OS basiert auf OSEK-OS. Applikationen, die für ein OSEK-OS entwickelt wurden, laufen auch mit einem AUTOSAR-OS.

Darüberhinaus definiert AUTOSAR-OS einige Erweiterungen, wie z.B.:

- **Schedule Tables**: statisches Abarbeiten von Tasks
- **OS-Applications**: eine Menge von OS-Objekten (Tasks, Schedule Tables, ISRs, ...), die eine zusammenhängende funktionale Einheit bilden, werden zu einer OS-Application zusammengefasst. OS-Applications werden unterteilt in:
    - **trusted OS-Applications**: uneingeschränkter Zugriff auf Speicher und Betriebssystemdienste
    - **non-trusted OS-Applications**: nur eingeschränkter Zugriff auf Speicher und Betriebssystemdienste. Zeitverhalten wird überwacht.
- **Memory Protection**: Das OS überwacht den Zugriff auf Daten, Stack und Code einer OS-Application. Versucht eine non-trusted OS-Application auf Daten einer anderen OS-Application zuzugreifen, wird eine Fehlerbehandlungsroutine ausgelöst (`E_OS_PROTECTION_MEMORY`).

5. RTOS Standards

- **Timing Protection**: Überwachung des Zeitverhaltens von non-trusted Applications. Bei Verletzung einer der u.g. Zeitschranken wird eine Fehlerbehandlungsroutine ausgelöst (`E_OS_PROTECTION_TIME`):
  - **execution time protection**: stellt sicher, dass eine Task oder Kategorie 2 ISR ihre konfigurierte maximale Ausführungszeit ("Execution Budget") nicht überschreitet.
  - **locking time protection**: definiert die maximale Zeit ("Lock Budget"), für die eine Task geblockt bzw. gesperrt werden darf (z.B. Warten auf Zugriff auf Ressourcen). Unterbrechungen durch höher-priore Tasks, aufgrund des "normalen" Scheduling, sind nicht im Lock Budget enthalten.
  - **inter-arrival time protection**: überwacht die minimale Zeitspanne ("Time Frame") zwischen 2 Aktivierungen einer Task. Das schützt u.A. vor sog. "babbling idiots"; z.B. Sensoren, die aufgrund einer Fehlfunktion dauernd Interrupts auslösen.
  - Bemerkung: Mit den o.g. protection Mechanismen ist sichergestellt, dass Deadline-Verletzungen bei Tasks erkannt werden. Eine "deadline protection" ist somit nicht nötig.

5. RTOS Standards

Ähnlich wie bei OSEK-OS gibt es für AUTOSAR-OS vier verschiedene **Scalability Classes** mit aufsteigender Komplexität. Protection-Mechanismen sind z.B. nur in den höchsten Klassen verfügbar.

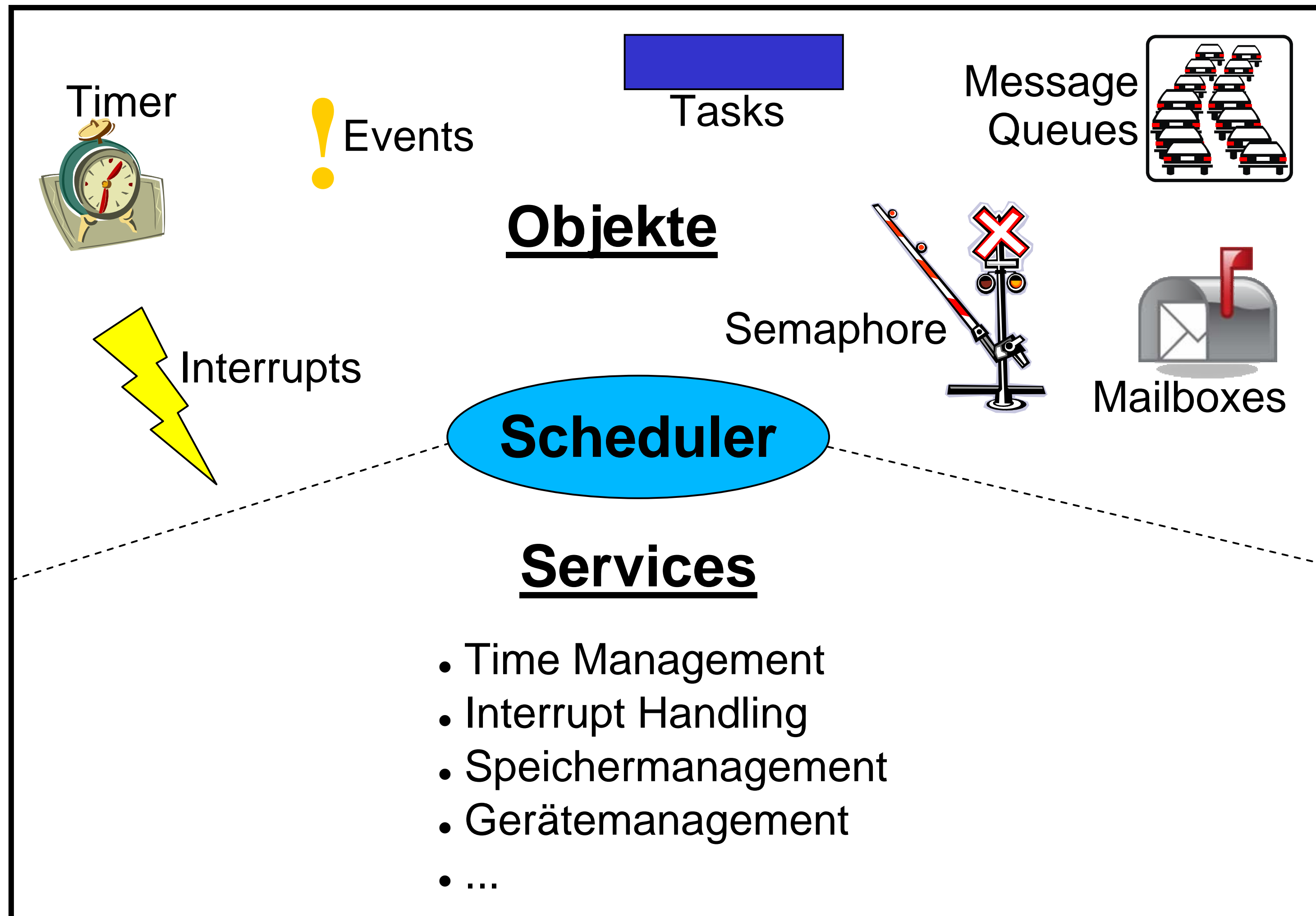
Die Sprache OIL wurde in AUTOSAR durch ein xml-Austauschformat abgelöst.

Der Nachfolger von OSEK-COM für die Kommunikation ist AUTOSAR-COM.

Der AUTOSAR Standard ist frei verfügbar (www.autosar.org). Wer ihn kommerziell nutzen will, muss Mitglied im AUTOSAR Konsortium sein und einen entsprechenden Beitrag zahlen.

Die ersten Serienfahrzeuge mit – teilweise – AUTOSAR Software gingen bereits 2009 in Produktion.

6. Elemente eines RTOS



6. Elemente eines RTOS

Der **Scheduler** ist verantwortlich für die Zuteilung von Rechenzeit an die aktiven Tasks, gemäß eines Scheduling-Algorithmus.

POSIX unterstützt prioritätenbasiertes Scheduling, gemischt mit FIFO, Round-Robin, OTHER (d.h. selbst implementiert). Prioritäten können sich dynamisch ändern.

OSEK und AUTOSAR unterstützen statische Prioritäten. AUTOSAR stellt zusätzlich Schedule Tables zur Verfügung. Diese können mit etwas mehr Aufwand aber auch in einem OSEK-OS implementiert werden.

6. Elemente eines RTOS

Timer

Für ein Echt**zeit**system ist es naturgemäß essentiell, Zugriff auf die "echte" Zeit zu haben, z.B. um:

- verstrichene Zeit zu messen,
- eine Task für einen bestimmten Zeitraum zu verzögern
- eine Task zu einem bestimmten Zeitpunkt zu aktivieren
- nach einem bestimmten Zeitraum zu reagieren, wenn ein erwartetes Ereignis nicht eintritt
- in bestimmten Zeiträumen ein Lebenszeichen zu senden

Jedes RTOS stellt dazu **Uhren** und **Timer Services** mit unterschiedlicher Genauigkeit und Komplexität zur Verfügung. Sie benutzen die Uhren, die i.d.R. vom Prozessor zur Verfügung gestellt werden. Die Zeit wird dabei in **Ticks**, d.h. Prozessortakte, gemessen und entsprechend der Taktfrequenz umgerechnet. Uhren unterscheiden sich in:

- globale Uhren für ein gesamtes Netzwerk; z.B. per GPS
- lokale Uhren. In einem Netzwerk entsteht hierbei das Problem der Uhrensynchronisation

6. Elemente eines RTOS

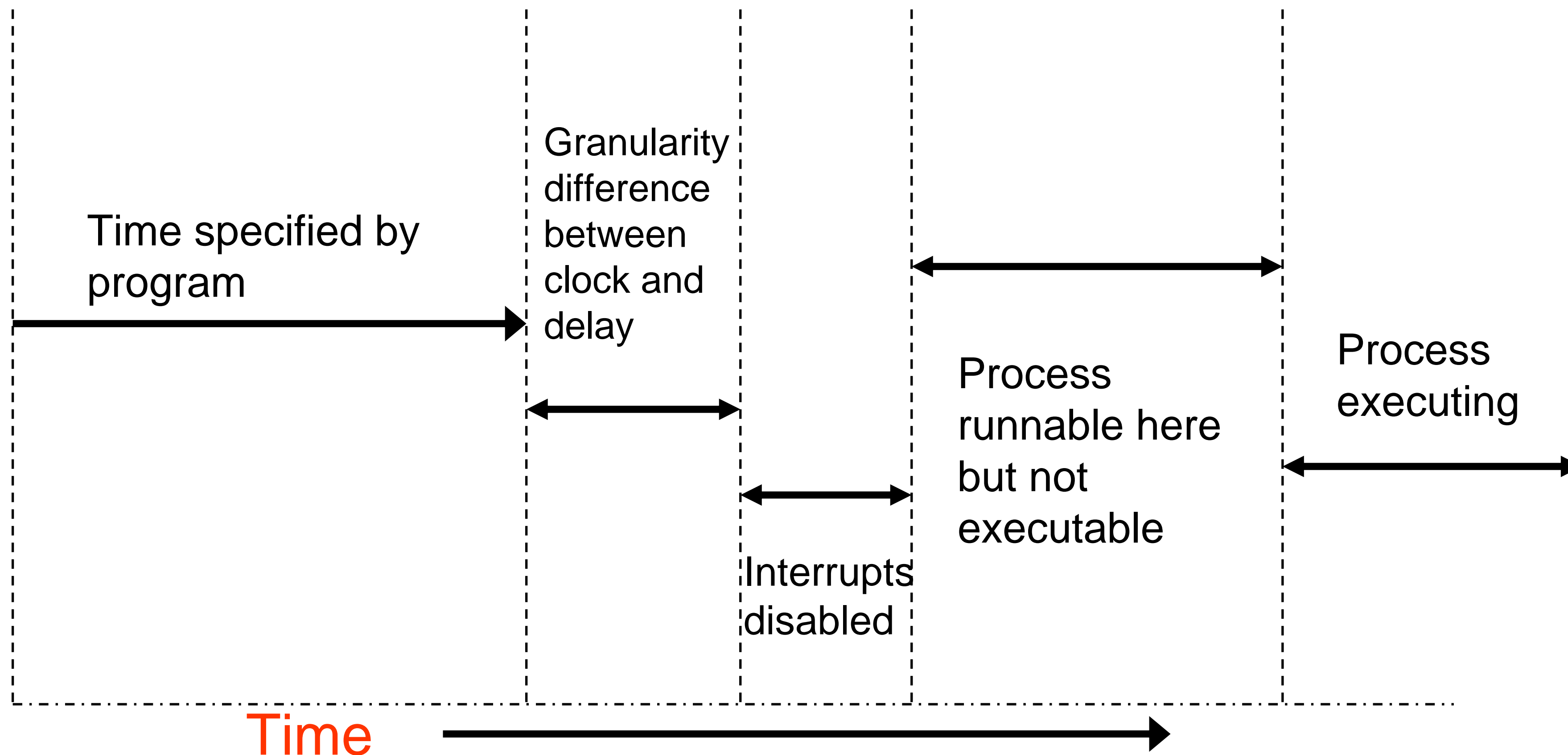
Beispiel: eine bestimmte Zeitspanne warten

```
start = clock();      //get current time
while (clock() - start < 31.4)
   ; // wait
```

Dieses "busy waiting" sollte allerdings vermieden werden. Meist stellt das RTOS dazu eigene Services zur Verfügung. In POSIX z.B. `sleep()` und `nanosleep()`. Dabei kann die entsprechende Task für den gegebenen Zeitraum suspendiert werden.

Zusätzlich ist zu beachten, dass die tatsächlich verstrichene Zeit i.d.R. nicht exakt gleich der für `sleep()` angegebenen ist, wie die folgende Graphik zeigt:

6. Elemente eines RTOS



Diesen Overhead nennt man **local drift**. Es lässt sich zumindest vermeiden, dass dieser Drift sich im Laufe der Zeit aufsummiert, indem man absolute Delays verwendet:

6. Elemente eines RTOS

Ressourcen (Betriebsmittel)

Eine **Ressource** ist ein

- (angeschlossenes) Gerät oder
- gemeinsam genutzte Daten.

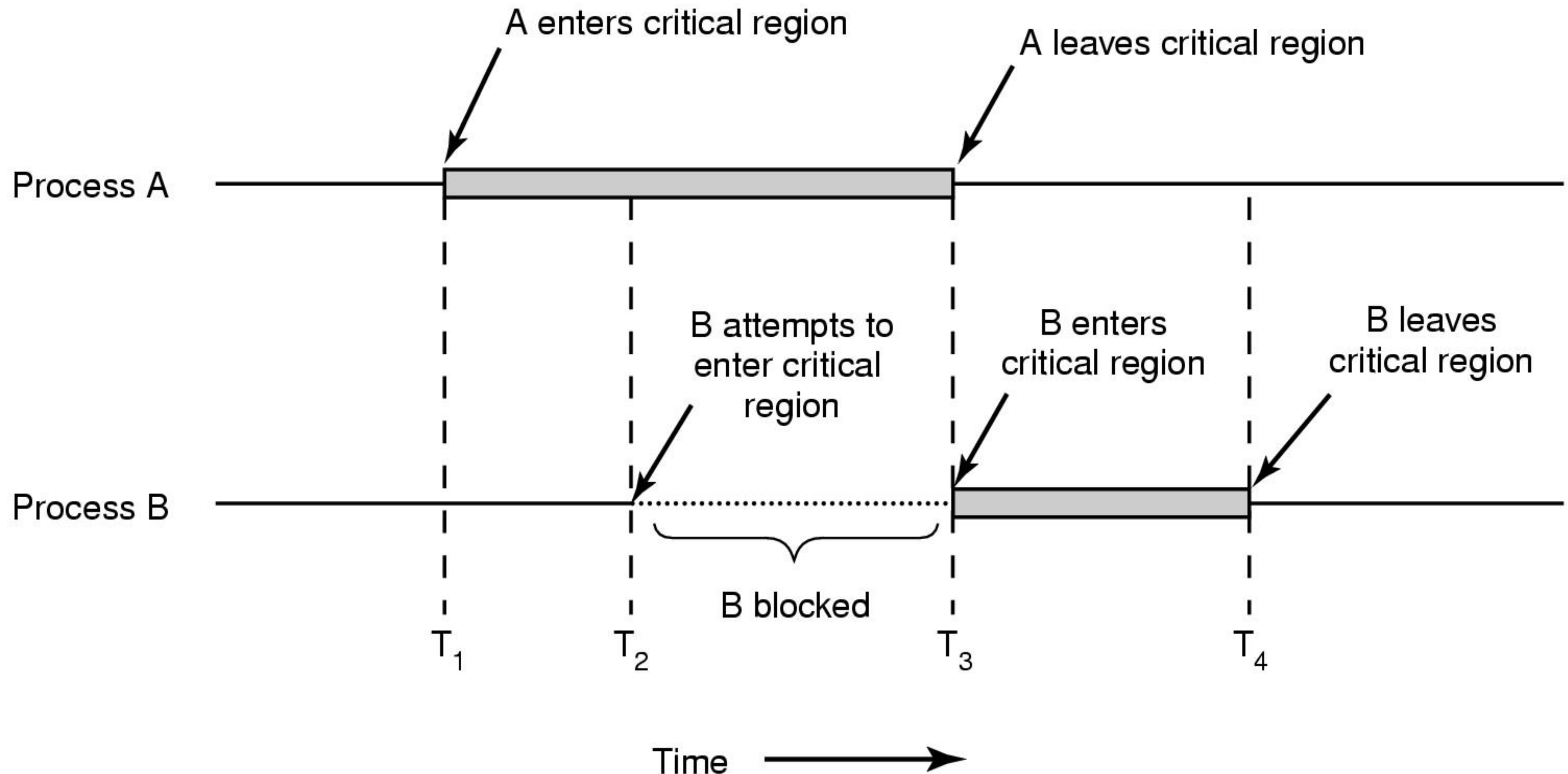
Unterbrechbare Ressourcen können einer Task ohne unerfreuliche Nebenerscheinungen entzogen werden (z.B. CPU). **Ununterbrechbare Ressourcen** können ihrem Besitzer nicht entzogen werden, ohne dass dessen Ausführung fehlschlägt (z.B. CD-Brenner). Allerdings gibt es in manchen Fällen recht einfache aber sehr spezielle Möglichkeiten den Zugriff zu regeln (z.B. Drucker-Spooling).

In allen anderen Fällen tritt ein Problem auf, wenn mehrere Tasks auf gemeinsam genutzte Ressourcen zugreifen, so dass es die Aufgabe des Betriebssystems ist, dafür Schutzmechanismen zur Verfügung zu stellen.

Zur abstrakten Formulierung des Problems für den allgemeinen Fall, werden die Teile eines Computer-Programms, in denen auf gemeinsam genutzte Daten zugegriffen wird, **kritischer Abschnitt** (= Ressource) genannt.

6. Elemente eines RTOS

Das Betriebssystem stellt **primitive Operationen** zur Verfügung, um **wechselseitigen Ausschluss** zu erzielen. Diese sind vor dem Betreten und nach dem Verlassen eines kritischen Abschnitts zu verwenden.



6. Elemente eines RTOS

Semaphoren

Ein Mittel (neben anderen) zur **Interprozess-Kommunikation (IPC)** sind Semaphoren [Dijkstra, 1965]. Ein (zählendes) **Semaphor** ist eine Integer-Variable, die eine bestimmte Anzahl an Weckrufen (Kapazität) verwaltet, und

- deren Wert Null anzeigt, dass alle Weckrufe verbraucht sind,
- ein Wert größer Null bedeutet, dass noch Kapazität vorhanden ist.

Operationen auf Semaphoren sind:

- **`DOWN()`** (**`sem_wait()`** in POSIX) prüft, ob der Wert des Semaphors größer als Null ist: dann wird der Wert vermindert. Ist der Wert Null, **blockiert** die Task und kann später durch einen anderen Thread per **`UP()`** wieder aktiviert werden.
- **`UP()`** (**`sem_post()`** in POSIX) inkrementiert das Semaphor und weckt irgendeine wartende Task.

Beides sind **atomare Operationen**, d.h. es ist sichergestellt, dass während der Ausführung keine andere Task auf das Semaphor zugreifen kann. Wir beschränken uns auf eine maximale Kapazität von ***1*** (auch **binären Semaphoren** bzw. Mutexe).

6. Elemente eines RTOS

Beispiel Semaphoren in POSIX für Konto-Buchungen:

```
#include <semaphore.h>
...
sem_t sem;    /* DEFINE semaphore */
int konten[100];

void buchen (int KNr, int betrag){
    /* Down */
    sem_wait(&sem);

    int alt = konten[KNr];
    int neu = alt + betrag;
    konten[KNr] = neu;

    /* Up */
    sem_post(&sem);
}
...
main(...

    sem_init(/* name */ &sem, /* shared flag */ 0, /* init value */ 1);

...
```

6. Elemente eines RTOS

Interrupts

(Unterbrechungen) werden von einem Gerät ausgelöst und führen dazu, dass eine entsprechende Interrupt-Routine (Behandlungsroutine) durch das Betriebssystem ausgeführt wird:

- Dazu wird die CPU (falls möglich) in den Kern-Modus umgeschaltet und während dieser Zeit werden keine weiteren Interrupts entgegen genommen.
- Die Zeit, die ein System braucht um Interrupts zu verarbeiten (d.h. die Zeitspanne bis zum Aufruf der Behandlungsroutine), wird **Interrupt Latency** genannt.
- Interrupt-Routinen haben üblicherweise eine höhere Priorität als andere Tasks.
- In Interrupt-Routinen wird oft ein gemeinsam genutzter Puffer benutzt, um Daten mit Tasks auszutauschen, welcher dadurch geschützt wird, dass
  - Interrupts unterbunden werden (**disable**), bevor auf die Daten zugegriffen wird und
  - Interrupts wieder erlaubt werden (**enable**), wenn man damit fertig ist.
- Dies ist wiederum nicht anderes, als ein **binäres Semaphor** zu benutzen!

6. Elemente eines RTOS

Bemerkung: In der Tat implementiert ein Betriebssystem ein Semaphor dadurch, während der Semaphor-Operationen die Interrupts zu unterbinden.

Schlussfolgerungen:

- **Interrupt-Routinen** lassen sich als hoch priorisierte Task mit einer bestimmten Periodenlänge modellieren und so in die Analyse einbringen
- Der **Disable/Enable**-Interupt-Mechanismus lässt sich in Blockiert-Zeiten umsetzen.

6. Elemente eines RTOS

In OSEK gibt es 2 Arten von Interrupt Service Routinen:

- ISR Kategorie 1: Die Routine benutzt keine Betriebssystemdienste (kann also insbesondere keine neuen Tasks aktivieren). Die unterbrochene Task wird im Anschluss an die ISR fortgesetzt. Diese Kategorie wird für extrem hochpriore Interrupts verwendet, die sehr schnell bearbeitet werden müssen.

- ISR Kategorie 2: Aufrufe von Betriebssystemdiensten sind erlaubt. Im Anschluss findet ein Rescheduling statt, d.h. der Scheduler wählt die nächste auszuführende Task.

Synchronisation via messages

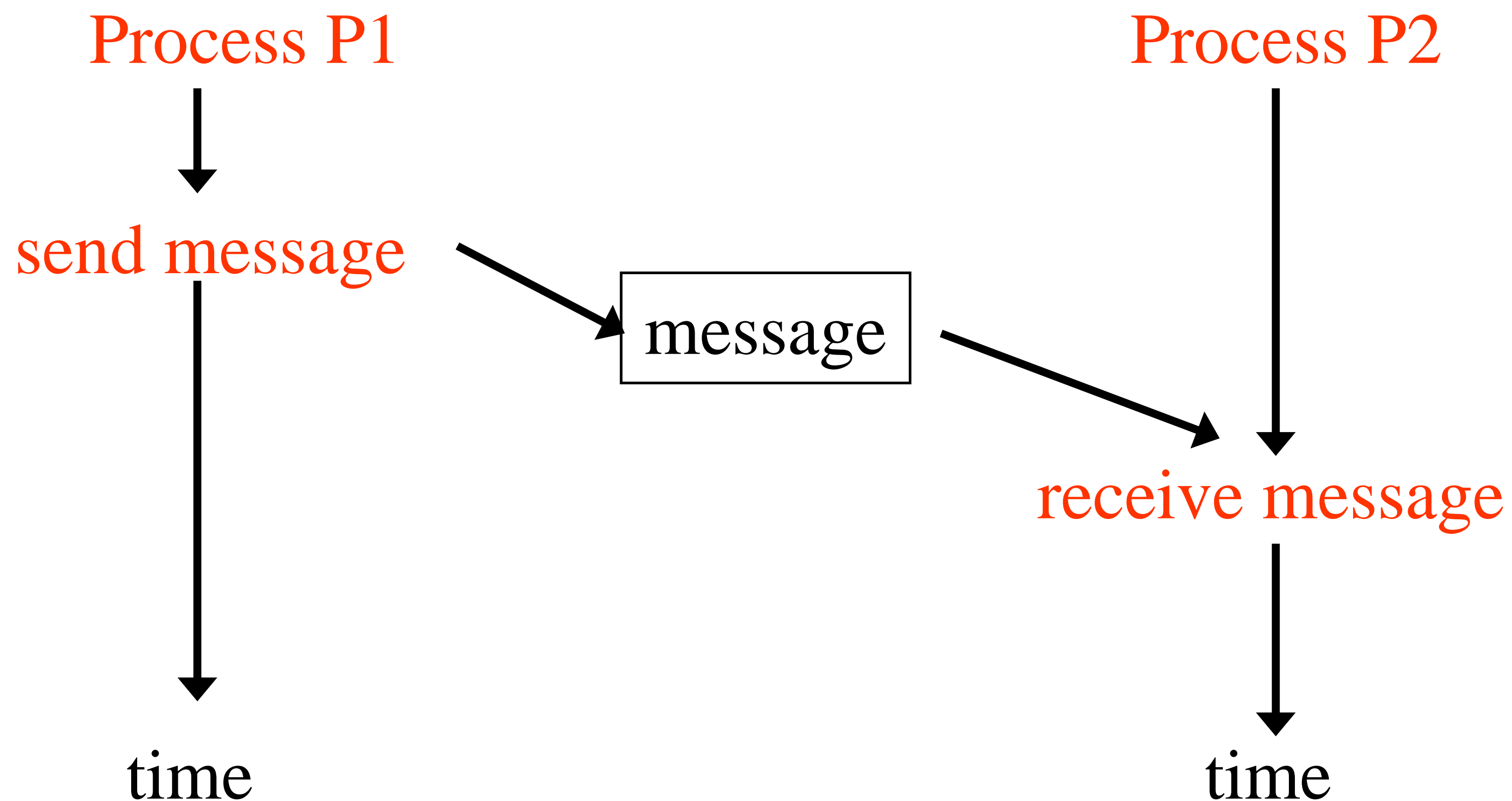
Messages ermöglichen sowohl Kommunikation als auch Synchronisation zwischen Tasks. Kommunikation kann

- asynchron oder
- synchron

erfolgen.

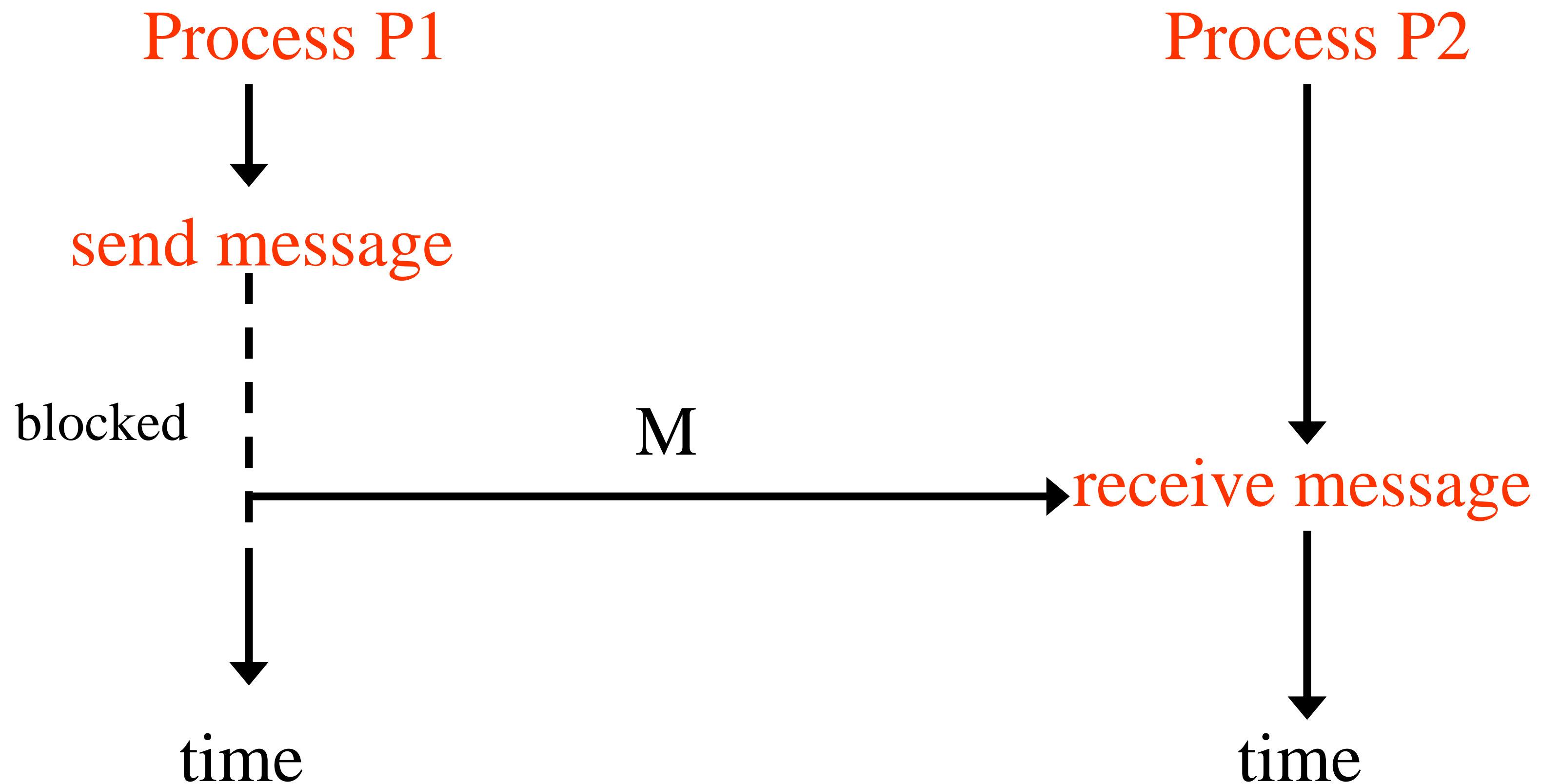
Asynchrone Kommunikation:

benutzt einen gemeinsamen Puffer: Mailbox oder Message Queue. Der Zugriff auf diesen Puffer muss natürlich (z.B. durch ein Semaphor) geschützt sein.



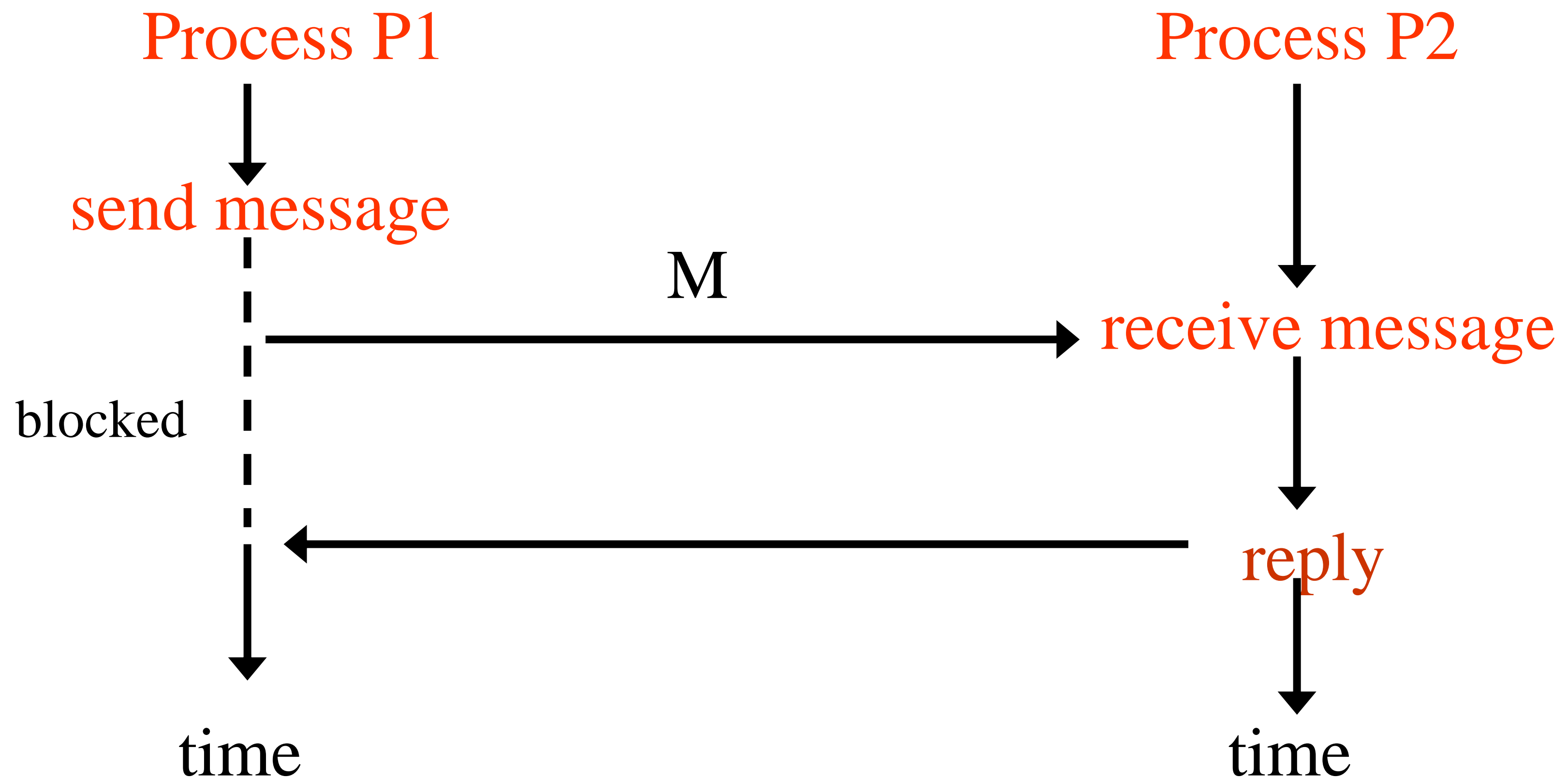
Synchrone Kommunikation:

oder auch Rendezvous; benötigt keinen Puffer.

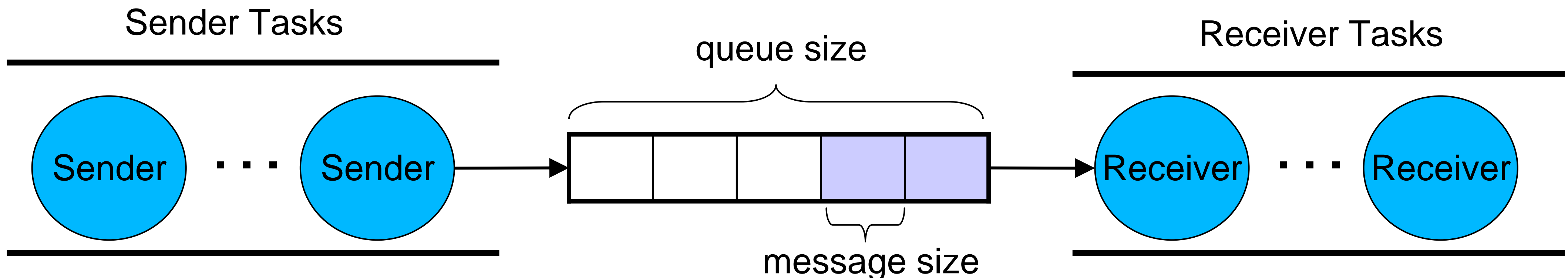


Remote Invocation:

oder auch Client/Server Kommunikation. Die Message M enthält die Parameter des "Funktionsaufrufs". Die Antwort liefert den Rückgabewert der Funktion.



Message Queue:



Eine Message Queue ist charakterisiert durch:

- ihre Länge, d.h. die maximale Anzahl Nachrichten
- die maximale Größe (bzw. Datentyp) einer Nachricht
- das Verhalten, wenn die Queue voll bzw. leer ist
  - voll: der Sender wird blockiert
  - leer: der Receiver wird blockiert
  - alternativ: bei voller Queue wird der letzte Wert überschrieben ("last-is-best Semantik"). Beim Versuch eine leere Queue zu lesen erhält der Receiver nur einen Fehlerstatus, wird aber nicht blockiert.

6. Elemente eines RTOS

Prinzipiell kann eine Message aus beliebigen Datentypen bestehen.
Wenn Sender und Receiver aber auf verschiedenen Prozessoren liegen, muss der Datentyp ggf. angepasst werden.
Die Message wird vom Sender-Prozessor als Folge von Bytes verschickt (z.B. in Form einer CAN-Message). Diese Bytefolge wird beim Empfänger wieder entsprechend interpretiert und in den Datentyp des Empfänger-Prozessors umgewandelt.

Message Queues in POSIX:

- POSIX unterstützt asynchrone message queues (`mqd_t`)
- Eine message queue kann mehrere sender und receiver haben
- Für Messages können Prioritäten definiert werden
- POSIX message queues haben verschiedene Attribute:
  - maximale Größe (Anzahl Messages)
  - Größe einer Message
  - Anzahl Messages in der Queue
  - verschiedene Flags, die das Verhalten steuern
- Zum Setzen und Lesen dieser Eigenschaften gibt es Attribut-Objekte (`struct mq_attr`).
- Ähnlich wie Files oder Streams wird eine Queue durch `mq_open`, `mq_close`, `mq_unlink` erzeugt bzw. gelöscht.
- Daten werden von/auf character buffer gelesen/geschrieben.
- Es ist nicht definiert, welcher der wartenden Sender/Receiver geweckt wird, wenn eine Queue nicht mehr voll/leer ist.
- Ein Receiver kann sich durch ein Signal (vgl. OS Event) informieren lassen, wenn in eine leere Queue geschrieben wurde.

6. Elemente eines RTOS

```
/* Establish connection between a process and a message queue NAME and
   return message queue descriptor or (mqd_t) -1 on error. */
extern mqd_t mq_open (const char *__name, int __oflag, ...);

/* Removes the association between message queue descriptor MQDES and its message
queue.  */
extern int mq_close (mqd_t __mqdes);

/* Query status and attributes of message queue MQDES.  */
extern int mq_getattr (mqd_t __mqdes, struct mq_attr *__mqstat);

/* Set attributes associated with message queue MQDES */
extern int mq_setattr (mqd_t __mqdes,
                 const struct mq_attr *__restrict __mqstat,
                 struct mq_attr *__restrict __omqstat);

/* Remove message queue named NAME.  */
extern int mq_unlink (const char *__name);

/*Register notification issued upon message arrival to an empty message queue MQDES.*/
extern int mq_notify (mqd_t __mqdes, const struct sigevent *__notification);

/* Receive the oldest from highest priority messages in message queue MQDES.  */
extern ssize_t mq_receive (mqd_t __mqdes, char *__msg_ptr, size_t __msg_len,
                          unsigned int *__msg_prio);

/* Add message pointed by MSG_PTR to message queue MQDES.  */
extern int mq_send (mqd_t __mqdes, const char *__msg_ptr, size_t __msg_len,
                   unsigned int __msg_prio);
```

Send / Receive mit Timeouts:

```
/* Receive the oldest from highest priority messages in message queue
   MQDES, stop waiting if ABS_TIMEOUT expires.  */
extern ssize_t mq_timedreceive (mqd_t __mqdes, char *__restrict __msg_ptr,
                               size_t __msg_len,
                               unsigned int *__restrict __msg_prio,
                             const struct timespec *__restrict __abs_timeout);

/* Add message pointed by MSG_PTR to message queue MQDES, stop blocking
   on full message queue if ABS_TIMEOUT expires.  */
extern int mq_timedsend (mqd_t __mqdes, const char *__msg_ptr,
                         size_t __msg_len, unsigned int __msg_prio,
                         const struct timespec *__abs_timeout);
```

7. System Overhead

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

- Queuing von Tasks
- Kontextwechsel
- Update interner Datenstrukturen
- Intertask Kommunikation: Senden und Empfangen von Nachrichten
- Bedienung von Interrupts

Normalerweise ist die Prozessorzeit für das OS gegenüber den anderen Tasks vernachlässigbar.
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 Schedulability Analyse erreicht werden.

7. System Overhead

Einer der wichtigsten Beiträge zum Systemoverhead ist die **Context Switching** Zeit, z.B. für das Sichern/Wiederherstellen der Prozessor-Register, des Stacks, etc.:

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

Ein weiterer wichtiger Beitrag ist die Prozessorzeit zur Behandlung der **Timer- und anderer Interrupts.**

Diese Zeiten müssen bei der Schedulability Analyse zusätzlich berücksichtigt werden.

8. Zusammenfassung

Wichtige Eigenschaften von RTOS

- Skalierbarkeit, minimaler Ressourcenverbrauch
- Vorhersagbarkeit
- Stabilität

RTOS Standards

- POSIX
- OSEK
- AUTOSAR
- proprietäre Lösungen

Elemente eines RTOS

- Tasks, Scheduler
- Timer
- Interrupts
- Events
- Message Queues