Betriebssysteme

Einige Begriffe und Grundlagen

Anhang zu "MiniOS, ein Minibetriebssystem für Mikrocontroller"

Version vom 17. Februar 2006, Hj. Kern, Winterthur

Inhalt

Betriebssysteme *Einige Begriffe und Grundlagen* 1
Inhalt 1
Dieses Dokument 2
Begriffe, Grundlagen 2
Echtzeit (real time), Echtzeitbetriebssystem RTOS 2
Embedded-System 2
Der Kernel 3
Der monolithische Kernel 3
Der Mikro-Kernel (micro kernel) 3
Task, Prozess (process), Thread 4
Verkettete Liste (linked list) 4
Task Control Block TCB, Process Control Block PCB 5
Ereignisse (events), Ereignissteuerung (event-driven multitasking) 5
Task Status (task states) 5
Current Task 6
Warteschlange (waiting queue) 6
Der Task-Scheduler 6
Stack, Stackpointer (SP) 6
Der Heap 6
Task-Kontextwechsel, Kontext-Switching (task context switching) 7
Preemptives Multitasking 7
Kooperatives (non preemptives) Multitasking 7
Prioritäts-Scheduling (priority-based scheduling), Task-Priorität (task prioritiy) 8
Round Robin Scheduling 8
System Idle Mode ("idle" = untätig) 8
Gemeinsame Ressourcen (shared resources) 9
Kritische Bereiche (critical sections or regions) Wiedereintritt (reentrancy, reentrant / non reentrant functions) 9
Die Funktion als "kritischer Bereich" 9
Prioritätsumkehr (priority inversion), Blockaden (deadlocks) 10
ISR, Interrupt Service Routine 11
Multitasking und Interrupts 11
Kommunikation und Synchronisation 13
Ausblick 14
Quellen 14

Dieses Dokument

soll dem Leser einige Begriffe aus der Welt der Betriebssysteme vermitteln. Diese gelten für universale Rechner (wie z.B. MS-Windows-Rechner) genauso wie für Mikrocontroller-Systeme (Embedded-Systeme). An Hand der gelieferten Informationen sollte es möglich sein, das Grundwissen "Betriebssysteme" aufzubauen und zu erweitern.

Begriffe, Grundlagen

Echtzeit (real time), Echtzeitbetriebssystem RTOS

In der Informatik spricht man von Echtzeit (real time), um sie von einer Modellzeit zu unterscheiden. Echtzeit ist die Zeit, die Abläufe in der "realen Welt" verbrauchen. Die Modellzeit hingegen ist die von einer Software selbst verwaltete Laufzeit. Ist die Modellzeit identisch mit der Echtzeit, so spricht man von Echtzeitverhalten.

Ein Echtzeitbetriebssystem (auch RTOS, real time operating system genannt) ist ein Betriebssystem, das für die Einhaltung von Zeitbedingungen und Vorhersagbarkeit des Prozessverhaltens optimiert wurde. Ein RTOS wird oft für Embedded-Systeme eingesetzt, also Systeme und Maschinen, die mittels eines Mikrocontrollers (embedded controller) gesteuert werden.

Zur genaueren Umschreibung was unter Echtzeit zu verstehen ist, wird manchmal unterschieden zwischen "weicher" und "harter" Echtzeit.

Weiche Echtzeit: Anforderungen an das Zeitverhalten des Systems sind statistisch definiert. Beispiel: Ein "weicher" MS-Windows-Rechner spielt eine mp3-Datei ab: Das Fehlen einiger Datenfragmente im Falle von Ueberlastspitzen, macht sich im Sound kaum bemerkbar und führt nicht zu Schäden im System.

Harte Echtzeit: Hier muss das Einhalten von Zeitpunkten in einer Anwendung garantiert werden können. Beispiel: Alle Interrupts eines gewissen Typs *müssen* in einem definierten Zeitfenster beantwortet werden. Harte Echtzeitbedingungen werden an Systeme gestellt, die z.B. Regelkreise und Maschinen steuern.

Embedded-System

In einem ES ist der in ein System "eingebettete" Prozessor für eine ihm fest zugewiesene Aufgabe zuständig. Häufig werden für diesen Zweck Mikrocontroller eingesetzt. Als Embedded Systeme gelten z.B. Geräte oder Maschinen die mittels Mikrocontroller gesteuert und überwacht werden.

Der Kernel

Der Kernel ist der zentrale Bestandteil eines Betriebssystems.
Gängige Anforderungen an einen Kernel sind:

- Parallelverarbeitung verschiedener Aufgaben (z.B. Multitasking)
- Einhaltung zeitkritischer Grenzen
- Offenheit für unterschiedliche Anwendungen und Erweiterungen

Der monolithische Kernel

Dieser besteht aus einem kompakten Block, in dem alle Funktionen integriert und praktisch untrennbar miteinander verbunden sind. Die Abhängigkeiten der Blöcke untereinander sind teilweise schwer nachvollziehbar. Beispiele für monolithische Kernel sind MS-Windows oder der Linux-Kernel. Vor- und Nachteile: In etwa die umgekehrten Vor- und Nachteile des Mikro-Kernels.

Der Mikro-Kernel (micro kernel)

ist ein vor allem bei Echtzeitbetriebssystemen (RTOS) verwendeter Betriebssystemkern, der weniger Funktionen als ein monolithischer Kernel enthält. Hier finden sich in der Regel lediglich Funktionen zur Task-/Prozessverwaltung und Grundfunktionen zur Synchronisation und Kommunikation.

Gerätetreiber und andere Teile eines Betriebssystems mit Micro-Kernel sind als eigenständige Anwender-Tasks implementiert und laufen zusammen mit dem Anwendungsprogramm auf Nutzerebene (User Level), dies im Gegensatz zum monolithischen Kernel, bei dem alle Treiber im privilegierten Modus (System Level) betrieben werden.

Vorteile:
Klares Interface-Design und separierte Komponenten. Der Absturz einer einzelnen Komponente des OS führt nicht zwangsläufig zum Zusammenbruch des gesamten Systems.
Die "Trusted Computing Base", der Kern-Code, dem ein Anwender vertrauen muss, ist im Vergleich zu monolithischen Betriebssystemen relativ klein und somit einfacher zu verifizieren. Die Vision hierbei ist, die Korrektheit des kompletten Betriebssystems zu beweisen.

Nachteile:
Geschwindigkeit, da das auf dem Mikrokern laufende Betriebssystem aus vielen einzelnen Nutzer-Prozessen besteht, sind wesentlich mehr Kontextwechsel nötig als bei monolithischen Betriebssystemen. Dadurch ist ein Mikrokern in der Regel langsamer als ein Monolith.

MiniOS ist als Mikro-Kernel aufgebaut. HW-Treiber, IO-Treiber etc. für die jeweilige Anwendung werden vom User als Tasks implementiert und gehören somit zur Nutzer/Anwender-Ebene.

Task, Prozess (process), Thread

Ein Task oder Prozess (beide Begriffe bezeichnen dasselbe) ist ein eigenständiger Programmteil zur Bearbeitung einer vorgegebenen Aufgabe innerhalb eines Betriebssystems. Eine Applikation kann aus einem oder mehreren Prozessen bestehen. Ein Prozess wiederum kann aus mehreren Threads aufgebaut sein, dies sind kleinere eigenständig ablaufende Programmteile innerhalb von Prozessen. Ein Thread benutzt den gleichen Speicherraum wie sein Eltern-Prozess. Prozesse jedoch sind bezüglich Speichernutzung voneinander abgeschottet, → Stabilität.

Siehe MS-Windows: <CTRL+ALT+DEL> zeigt die laufenden Anwendungen und die zugehörigen Prozesse.

Eine Task in einem Betriebssystem für Mikrocontroller ist normalerweise als Funktion mit einer Endlosschleife implementiert, etwa in der folgenden Art:

Listing - Task-Funktionsrumpf mit Endlosschleife

```
// Task-Funktionsrumpf mit Endlosschleife

void Task1_Function(void)
{
   InitTask();

   while (1)
   {
       TaskJob();
   }                   // no Return
}
```

Verkettete Liste (linked list)

Eine Datenstruktur zur Verwaltung von Task- und Event-Daten, als Datenpuffer, Mailbox und Warteschlange (queue) in Betriebsystemen.

Der Aufbau sieht etwa so aus:
Listenkopf → Listenelement1 → Listenelement2 → ListenelementN → Listenende.

Jedes Listenelement zeigt mittels Pointer (→) zum nächsten Element. Ein Start-Pointer zeigt auf den "Listenkopf" (head) und ist quasi der "Griff" (handle) um auf die Liste zuzugreifen. Am Listenende steht ein Null-Pointer als Abschluss. Eine verkettete Liste erlaubt ein einfaches Einfügen und Löschen von Listenelementen, ohne Daten zu kopieren, da es sich um Pointer-Operationen (Pointer "umbiegen") handelt. Die Listengrösse ist nur begrenzt vom zur Verfügung stehenden Speicher.

Listing - struct-Element einer verketteten Liste: RT-Linux-Task (unvollständig)

```
struct rt_task_struct {
    int   *stack;
    int   *stack_bottom;
    int    state;
    int    priority;
    RTIME  resume_time;
    ...
    struct rt_task_struct *next;  // Zeiger auf nächstes
                                  // Listenelement
};
```

Task Control Block TCB, Process Control Block PCB

Die Prozess- oder Taskdaten (Stack-Infos, Status, Priorität etc.) werden mittels des TCBs verwaltet. Dieser ist häufig als verkettete Liste aufgebaut, um Tasks dynamisch einzufügen (create task) oder zu löschen (delete task).

Ereignisse (events), Ereignissteuerung (event-driven multitasking)

Ereignissteuerung liegt dann vor, wenn die Tasks eines Systems auf Ereignisse warten und erst bei deren Eintreffen aktiviert (= rechenbereit) werden. Grundlage von Ereignissen sind oft HW-Interrupts oder das Ergebnis einer zyklischen Abfrage (polling) der HW oder SW.

Ein OS reagiert auf Events, indem es Messages an die betreffenden Tasks verschickt, z.B. mit einer Funktion PostMessage() oder indem es Semaphore aktiviert, z.B. mit einer Funktion SignalSemaphore().

Die Event-Daten (Event-Typ, Zugehörigkeit etc.) werden im Event Control Block (ECB) als verkettete Liste verwaltet und können dynamisch erstellt (create event) oder gelöscht (delete event) werden.

Tabelle - Mögliche Events und ihre Grundlagen in einem System

Ereignisse in einem System, Beispiele	*Grundlage*
eine Taste wird gedrückt	HW Tastatur-Interrrupt
ein Timer läuft ab	HW Timer-Interrupt, SW-Timer
ein Fehler passiert (exception, error)	SW-Exception, Trap *)
ein HW-Interrupt einer Peripherie-Einheit trifft ein	HW Peripherie-Interrupt
eine Ressource wird frei zur Benutzung	SW-Event *)
ein Resultat einer Berechnung ist bereit	SW-Event *)
ein (Windows-) Mausklick erfolgt	HW Mouse-Interrupt
eine Schnittstelle empfängt oder sendet ein Datenbyte	HW Schnittstellen-Interrupt
neue Daten für das Display sind bereit	SW-Event *)
ein Semaphor ändert den Zustand	SW-Event *)
eine Message trifft in der Mailbox ein	SW-Event *)

*) Grundlage ist oft ein HW-Ereignis

Task Status (task states)

Das Betriebssystem ordnet einer Task verschiedene Zustände zu, die dem Scheduler mitteilen, ob und was mit der Task zu tun ist.

Bild - Einige wichtige Task States

```
TaskInit()   →   READY   ← →   RUNNING   ← →   WAITING
```

Die gerade rechnende Task besitzt den Status RUNNING. Die CPU kann innerhalb eines Zeitabschnitts *eine* RUNNING-Task bearbeiten. RUNNING kann nur eine rechenbereite READY-Task werden. Häufig wartet eine Task auf ein Ereignis und beansprucht keine Prozessorzeit, sie besitzt den Status WAITING. Sobald das erwartete Ereignis eintrifft, weist die Ereignisverarbeitung der wartenden Task den Status READY zu, sie ist jetzt rechenbereit.

Current Task

ist die gerade rechnende, resp. laufende Task, Status: RUNNING.

Warteschlange (waiting queue)

Wartende Tasks erhalten beim Eintreffen des erwarteten Ereignisses den Zustand READY und werden in die READY-Queue (Warteschlange für bereitstehende Tasks) eingefügt. Warteschlangen werden meist als verkettete Listen angelegt, um Tasks einzufügen oder zu entfernen.

Der Task-Scheduler

ist Teil des Kernels und teilt den READY-Tasks CPU-Rechenzeit zu, indem sie Tasks aktiviert oder deaktiviert, in Abhängigkeit von den eintreffenden Ereignissen. Der Scheduler "plant" und führt die Task-Kontextwechsel durch. Je nach Art des Scheduler-Algorithmus, kann dies auf Grund von Task-Prioritäten (priority scheduling) oder indem alle Tasks der Reihe nach aufgerufen werden (round robin scheduling), geschehen.

Stack, Stackpointer (SP)

Der Stack einer Anwendung belegt einen Teil des Systemspeichers und enthält lokale Funktionsvariablen, Rücksprungadressen für Subfunktionen, Funktionsparameter und Rückgabewerte, gesicherte CPU-Register (PUSH und POP) etc.

Der Stackpointer (SP, ein CPU-Register) enthält die Adresse des nächsten freien Stack-Speicherplatzes ("zeigt" auf freien Speicherplatz im Stack). Ein Stack-Abschnitt der alle temporär gesicherten Bytes einer aufgerufenen Funktion umfasst, heisst Stack Frame.

In einem Multitasking-OS erhält jede Task (= Funktion) ihren eigenen Stack-Bereich zugewiesen (task stack space).

Der Heap

("Haufen") ist ein Speicherbereich in dem eine Applikation Speicherblöcke reservieren (allozieren, C: malloc(), C++: new) und wieder freigeben (C: free(), C++: delete) kann. Auf dem Heap werden globale Daten und Objekte verwaltet (dynamische Speicherverwaltung).

Task-Kontextwechsel, Kontext-Switching (task context switching)

Beim Task-Wechsel wird vom Scheduler ein Kontext-Switch durchgeführt. Die laufende Task wird deaktiviert, indem die Prozessorregister auf den Stack gerettet und die Task-Umgebung (task context), d.h. Stackpointer und Program-Counter, gesichert werden. Beim Kontextwechsel wird der Inhalt des SP so modifiziert, dass er auf den neu benötigten Stack-Bereich zeigt (Stackpointer "umbiegen"). Der Kontext der nächsten auszuführenden Task wird geladen und deren Bearbeitung weitergeführt (task resume).

Die Leistungsfähigkeit des Kernels nimmt mit zunehmender Anzahl Task-Wechsel pro Zeitintervall ab, da ein Task-Switching "unprodukive" Zeit beansprucht (system overhead).

Preemptives Multitasking

Der Kernel - genauer der Scheduler - entscheidet auf Grund der eintreffenden Ereignisse, Interrupts, Task-Prioritäten etc., ob ein Taskwechsel vorzunehmen sei.
Jede laufende Task kann jederzeit (asynchron) vom Kernel unterbrochen werden (preemption). Der Kontext-Switch wird innerhalb von Interrupt-Routinen ausgeführt.

Vorteile des preemptiven MT:
- Vorhersagbare kurze Ereignis-Beantwortungszeiten (event response time → Echtzeit-Kriterium).
- Eine einzelne Task kann nicht das ganze System blockieren.

Nachteile:
- Die Interrupt-Latenzzeit ist lang, da die Interrupts im Programmablauf häufig gesperrt werden (interrupts disabled).
- Massnahmen zur Synchronisation beim Zugriff auf gemeinsame Ressourcen sind nötig.
- Der Programmablauf schwierig vorhersagbar (nicht deterministisch), da die Taskwechsel von asynchronen Ereignissen abhängig sind, Fehler sind schwer zu finden.

Kooperatives (non preemptives) Multitasking

Die Task entscheidet, ob und wann die Kontrolle zurück an den Scheduler gehen soll, damit eine andere READY-Task Prozessorzeit zugewiesen bekommt. An geeigneter Stelle im Task-Code sind OS-Funktionen oder Makros wie OS_Yield() oder SCHEDULE() eingefügt, die den Scheduler aufrufen. Der Task-Kontextwechsel erfolgt synchron zum Programmablauf, d.h. immer an fest definierten Stellen innerhalb der Task.

In einfachen kooperativen Systemen, die keine separaten Stack-Bereiche für jede Task aufweisen (siehe MiniOS) kann der Kontextwechsel ohne grossen Speicherbedarf und schnell erfolgen.

Vorteile des kooperativen MT:
- Die Interrupt-Latenzzeit ist kurz, da die Interrupts meistens eingeschaltet bleiben.
- Sperrmechanismen für gemeinsame Ressourcen und kritische Bereiche sind nicht nötig.
- Der Programmablauf ist vorhersagbar (deterministisch), da die Taskwechsel vom Programmierer bestimmt werden, Fehler sind verfolgbar.

Nachteile:
- Systemblockierung durch eine Task ist möglich.
- Die Ereignis-Beantwortungszeit ist lang, da die rechnende Task zuerst beenden muss.
- Einfache Systeme besitzen nur einen gemeinsam genutzten Stack-Bereich: ein Stack-Fehler einer Task kann daher das System lahmlegen.

Prioritäts-Scheduling (priority-based scheduling), Task-Priorität (task prioritiy)

Um ein Prioritäten-Scheduling durchzuführen, wird den Tasks ein Prioritätswert (z.B. 0...255) zugeordnet, um sie entsprechend der Rangordnung und Wichtigkeit behandeln zu können. Statische Prioritäten werden zur Kompilationszeit fixiert. Einige Betriebssysteme können die Prioritäten auch dynamisch (während dem Programmablauf) verändern und neu zuordnen.

Prioritäten-Scheduling, preemptiv: Der Scheduler wacht darüber, dass die gerade rechnende Task diejenige mit der höchsten Priorität ist.

Prioritäten-Scheduling, kooperativ: Nach dem von der aktuellen Task ausgeführten Rücksprung in den Scheduler, wird aus der READY-Warteschlange die Task mit der höchsten Priorität ausgeführt.

Nachteile des Prioritäts-Scheduling: Tasks mit niedriger Priorität bekommen wenig oder gar keine Prozessorzeit vom Scheduler mehr zugeordnet, falls höher priorisierte Tasks voll ausgelastet sind: *die Task "verhungert" (task starvation)*. Dynamische Priorisierung der Tasks, erweiterte Scheduling-Algorithmen und andere Massnahmen verhindern diesen Nebeneffekt.

Round Robin Scheduling

Die READY-Tasks werden nicht entsprechend der Reihenfolge ihrer Prioritäten, sondern in fester Reihenfolge hintereinander ausgeführt. Beim preemptiven RR-Scheduling wird den Tasks häufig ein Zeitintervall (*Quantum* oder *Time Slice* genannt) zugeodnet, nach dessen Ablauf die Task unterbrochen und zur nächsten READY-Task gewechselt wird.

System Idle Mode ("idle" = untätig)

Wenn keine Task rechenbereit ist (alle warten oder sind gestoppt), ruft der Scheduler die Idle-Task (Leerlauf- oder Ruheprozess) auf und wartet auf das nächste Ereignis (z.B. einen Interrupt). Die Idle-Task wird oft dazu benutzt, um die Systemauslastung zu erfassen und Power-Management-Funktionen auszuführen (z.B. den Akku im Notebook-PC schonen, indem die CPU-Clock-Frequenz gesenkt wird).

Gemeinsame Ressourcen (shared resources)

Gemeinsame Ressourcen sind System- oder Programmkomponenten, die von verschiedenen Tasks (und ISR) gemeinsam genutzt werden und nur einmal vorhanden sind, wie zum Beispiel:

- gemeinsame Speicherbereiche (shared memory)
- globale Variablen und Datenstrukturen
- Schnittstellen
- IO-Ports
- (LC)Display
- div. Ein-, Ausgabegeräte

Um eine exklusive Nutzung eines Betriebsmittels zu erreichen, müssen gemeinsame Ressourcen vor gleichzeitigen Mehrfachzugriffen geschützt werden. Dies sollte "Kollisionen" und Datenverluste verhindern. Synchronisierungs- und Schutzmechanismen die *den Zugriff auf die Ressource steuern*, decken sich mit denjenigen, die nachfolgend zum Thema "kritische Bereiche" erwähnt werden.

In kooperativen Systemen stellt die Nutzung von gemeinsamen Ressourcen systembedingt kein Problem dar (ausser beim Zugriff durch ISRs), da der aktuelle Task die Ressource erst nach Beenden des Zugriffs freigibt.

Kritische Bereiche (critical sections or regions)
Wiedereintritt (reentrancy, reentrant / non reentrant functions)

"Kritischer Bereich" nennt man den Bereich des Codes, der auf eine gemeinsame Ressource zugreift (z.B. mittels einer Funktion). Dies führt zu Mehrfachaufrufen oder Wiedereintritten in den Funktionscode (reentrancy) in preemptiven Systemen oder ISRs.

Hier ein Beispiel einer Anwendung, die - oft unbemerkt - zu Datenfehlern führt:
Ein Hauptprogramm ruft gerade eine Funktion FnA() auf, welche zusammengehörige Messwerte in ein (globales) Array schreibt. *Während dem Schreibvorgang* wird sie von einer ISR (Interrupt) unterbrochen, welche ebenfalls FnA() aufruft und vollständig ausführt. Nach der Rückkehr ins Hauptprogramm wird der Schreibvorgang der *ersten Wertereihe* zu Ende geführt. Weil FnA() nicht wiedereintrittsfähig ist, liegen im Array jetzt Messwerte von zwei verschiedenen Messreihen ohne Zusammenhang!

Die Funktion als "kritischer Bereich"

Nicht wiedereintrittsfähig (non-reentrant) ist eine Funktion, welche auf Ressourcen zugreift, die gleichzeitig von andern Tasks oder ISR genutzt werden könnten. Dies gilt auch, wenn die Funktion static-Variablen oder sonstige globale Variablen und Datenstrukturen (siehe auch Arrays, Struct oder Float!) verwendet. Ein Mehrfachaufruf führt dazu, dass sich Schreibzugriffe auf globale Daten überlagern und zu unbestimmten Ergebnissen (Datenverlust) führen.

Eine *Funktion ist wiedereintrittsfähig*, wenn sie mehrfach aufgerufen werden kann. Wiedereintrittsfähige Funktionen dürfen nur Variablen enthalten, die bei jedem Aufruf im Speicher neu angelegt und beim Verlassen wieder freigegeben werden (→ Kompiler, Stack, lokale Variablen). Eine wiedereintrittsfähige Funktion kann auch rekursiv aufgerufen werden (die Funktion kann sich selbst aufrufen).

Zugriffe einer Funktion auf Ressourcen, die systembedingt nur einmal vorhanden sind, müssen synchronisiert werden (reentrancy protection).

Ein temporäres Stoppen des Schedulers oder der Interrupts und Schutzmechanismen nach dem Prinzip des *gegenseitigen Ausschlusses* (mutual exclusion, Mutexe, Semaphore, u.a.) verhindern, dass die gerade den kritischen Bereich durchlaufende Task dabei unterbrochen wird. Als gefürchtete Nebeneffekte solcher Schutzmassnahmen können Deadlocks und Prioritätsinversionen (beides führt zu Systemblockaden) auftreten, siehe unten.

In kooperativen Systemen stellen Critical Sections systembedingt kein Problem dar (ausser beim Zugriff durch ISRs), da der aktuelle Task den kritischen Bereich ohne Unterbruch beenden kann.

Prioritätsumkehr (priority inversion), Blockaden (deadlocks)

sind Konflikte, die bei preemptiven Systemen auftreten, wenn Tasks aufeinander warten oder sich gegenseitig blockieren. Schutzmassnahmen für kritische Bereiche und gemeinsam genutzte Ressourcen können zu solchen unerwünschten Nebeneffekten führen.

Gegenmassnahmen sind u.a.: Timeout-Ueberwachung, dynamische Priorisierung der Tasks, richtige Reihenfolge bei der Zuteilung der Ressourcen etc.

Beispiel einer Prioritätsumkehr:
Eine hoch priorisierte Task TA ist aktiv und benötigt eine Ressource, die gerade von einer deaktivierten Task TB mit niedriger Priorität belegt wird. Falls der Scheduler eisern die Regel "höchste Priorität zuerst" einhält, ist TA blockiert und muss auf TB warten, welche die Ressource zuerst freigeben muss. Dies kommt einer Prioritätsumkehr gleich, die der Scheduler nur beheben kann, indem er temporär die Prioritäten der Tasks tauscht, bis die Ressource frei wird.

Beispiel eines Deadlocks:

- Task T1 belegt die serielle Schnittstelle und wartet auf den Zugang zum Datenpuffer um die Schnittstelle zu bedienen.
- Task T2 belegt den Datenpuffer und wartet auf den Zugang zur seriellen Schnittstelle welche von T1 besetzt ist...

ISR, Interrupt Service Routine

ISR: Einige Regeln um Aerger mit Interrupts zu vermeiden

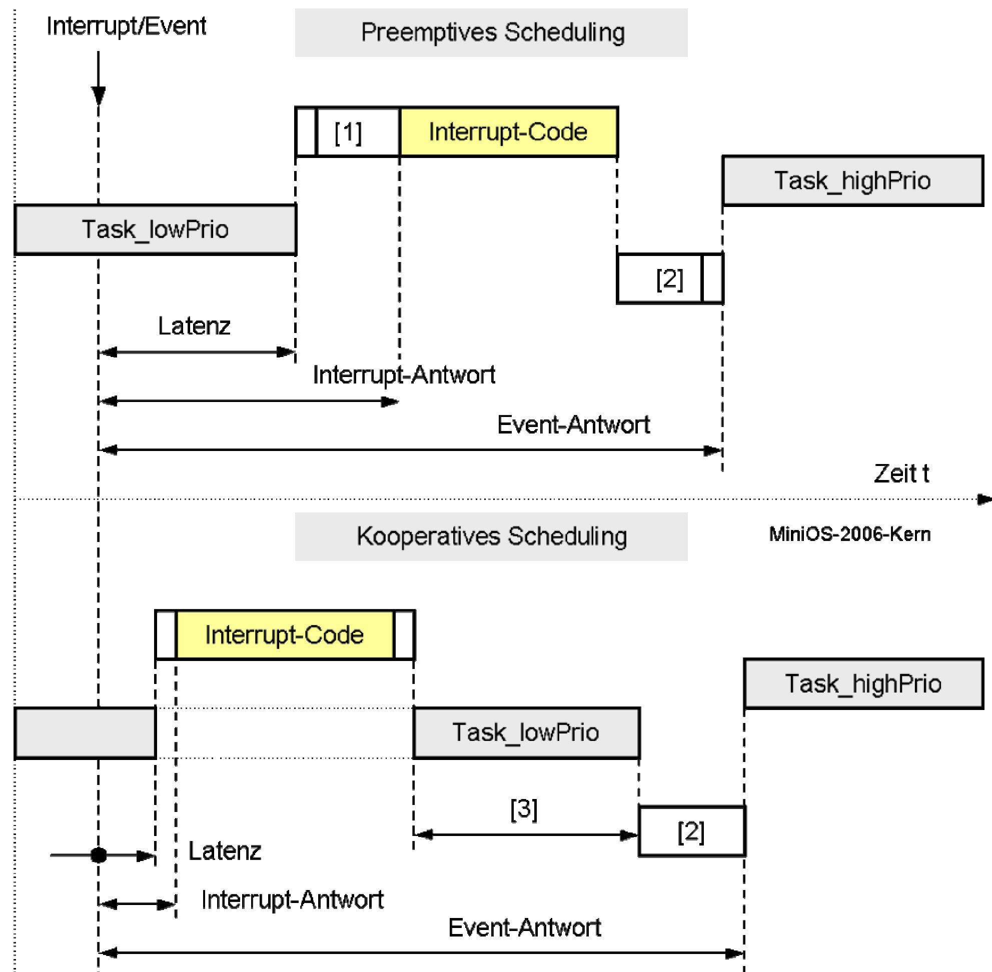
- Die ISR ist eine von den Tasks (*) losgelöste Funktion, deren Ausführung jederzeit und überall im Programmablauf (asynchron) eintreten kann. Die ISR hat weder Uebergabe- noch Rückgabeparameter.
- Eine ISR kann nicht auf Ereignisse warten.
- Die ISR so kurz wie möglich halten, damit keine anderen Interrupts "ausgebremst" werden.
- Eine ISR kann durch andere Interrupts oder Tasks blockiert werden (interrupt disabled) und darum möglicherweise zeitliche Anforderungen nicht erfüllen.
- Die ISR so entwerfen, dass keine kritischen Bereiche entstehen, Code "reentrant" halten.
- Keine System-Ressourcen nutzen, auf die auch innerhalb von Tasks zugegriffen wird (shared resources), es sei denn, die Mittel sind dafür vorgesehen (z.B. Semaphore, Event-Flags).
- Ein Semaphor als Event-Flag in der ISR signalisiert der darauf wartenden Task das Eintreffen des Interrupts.

(*) bedeutet immer auch main()-Programm!

Multitasking und Interrupts

Beim Umgang mit Interrupts gelten die Ueberlegungen zum Thema Wiedereintritt und die Massnahmen zum Schutz von kritischen Bereichen und gemeinsamen Ressourcen (siehe da).

Bild - Scheduling mit Interrupts: preemptives und kooperatives System



Latenz, Interrupt-Latenzzeit (interrupt latency, latent = verborgen):
ist die Zeitdifferenz vom Eintreffen des Interrupt-Ereignisses (interrupt request) bis zur Bearbeitung der ersten Instruktion der ISR. Sie setzt sich im Wesentlichen zusammen aus der längsten im System vorkommenden Interrupt-Sperrzeit (interrupt disable time) plus der Bearbeitungszeit der längsten vorkommenden CPU-Instruktion.

Interrupt-Antwortzeit (interrupt response time):
ist die Latenzzeit plus die Zeit für die Sicherung des CPU-Kontexts (Registersicherung auf den Stack), plus allenfalls die Zeit für die Bearbeitung der Eintrittsfunktion in den Kernel bei preemptiven Systemen [1].

Event-Antwortzeit (event response time):
ist die Zeitdifferenz zwischen dem Eintreffen des Ereignisses und dem Beginn der Bearbeitung der zugehörigen Task. Zu den erwähnten Zeiten addieren sich die maximale Zeit für die Bearbeitung der ISR, plus die Zeit für den Task-Kontextwechsel im Scheduler [2]. Beim kooperativen System wird diese Zeit zusätzlich vom längsten vorkommenden Zeitintervall zwischen zwei Kontextwechseln einer Task [3] bestimmt.

Kommunikation und Synchronisation

Tasks und ISR brauchen Mittel um miteinander zu kommunizieren oder sich zu synchronisieren und um Zugriffe auf Ressourcen zu steuern. Ein Betriebssystem sollte einige dieser Mittel zur Verfügung stellen.

Synchrone Kommunikation (blockierend):
die Teilnehmer warten aufeinander und übergeben sich die Daten. Zur Steuerung der Uebergabe dienen Synchronisationseinrichtungen, z.B. Semaphore.

Asynchrone Kommunikation (nicht blockierend):
ein Teilnehmer deponiert die Daten an einem vereinbarten Ort, wo sie der andere irgendwann abholen kann. Als "Uebergabeorte" dienen z.B. Mailboxen, FIFO-Puffer und andere Datenstrukturen.

Einige Mittel für die Kommunikation und Synchronisierung:

- **Semaphore** sind OS-Variablen, die u.a. eingesetzt werden für:
  - Signalisierung von Ereignissen
  - Steuerung des Zugriffs auf Ressourcen

- **Message-Puffer oder Mailboxen** sind vom OS unterstützte Datenpuffer für die *Inter-Task-Kommunikation (message passing)*.
  Eine Task kann Messages an eine andere Task senden oder von ihr Nachrichten empfangen (direkte Punkt-zu-Punkt Kommunikation). Werden die Messages über Mailboxen ausgetauscht, spricht man von indirekter Kommunikation.
  Globale Mailboxen stehen allen Tasks zur Verfügung.

- **Shared Memory**, gemeinsamer *globaler Speicherbereich* für Event-Flags, Variablen, Pointer auf Funktionen und Datenstrukturen etc. Shared-memory-Zugriffe müssen synchronisiert werden um Datenfehler ("Kollisionen") zu verhindern.

Kommunikationsmittel, welche nicht vom OS unterstützt werden (z.B. globale Variablen), besitzen den Nachteil, dass sie von der betreffenden Task zyklisch abgefragt werden müssen (software polling) und damit unnötig CPU-Zeit beanspruchen.

Kommunikationsmittel können auch kombiniert eingesetzt werden:
Beispiel: Eine Task kommuniziert das Ende einer Berechnung mittels einer Message in der globalen Mailbox. Die auf diese Nachricht wartende zweite Task ruft daraufhin eine Funktion auf, um die Ergebnisse zu übernehmen und weiter zu bearbeiten.

Ausblick

Auch für "sehr kleine" 8-Bit-Mikrocontrolleranwendungen, basierend auf Kontroller-Typen wie PIC von Arizona-Microchip, AVR von Atmel etc. werden zunehmend häufig "schmale" ressourcen-schonende Multitasking-OS eingesetzt. Dies lohnt sich in punkto Entwicklungskosten und "Time-to-Market": möglichst schnell an den Markt mit einem neuen Produkt, da die Konkurrenz bekanntlich nicht schläft...

Erweiterte Betriebssysteme enthalten Module für die dynamische Speicherverwaltung (memory management unit, MMU), den Multiuser- und Multiprozessor-Betrieb. Ethernet-Anbindung, zunehmende Vernetzung, "verteilte Systeme" und Multicore-Design: dies scheinen die Tendenzen für die nächsten Jahre zu sein, auch im Bereich der Embedded-Systeme!

Quellen

Grundlage der Dokumentation sind Texte und Artikel von verschiedenen Autoren, u.a.

J. Labrosse: *Embedded Systems Building Blocks*, CMP Books, 2002