Aufgabe 1: Schedule Table Scheduler

Echtzeitsysteme - Übungen zur Vorlesung

Peter Ulbrich, Martin Hoffmann

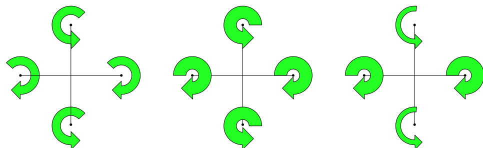
Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
www4.informatik.uni-erlangen.de

25. Oktober 2010

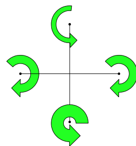
Aufgabenstellung: Quadrokopter-Kontrollsystem

Anforderungen an das Kontrollsystem

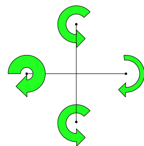
- Schwebeflug in konstanter Höhe.
- Steigflug oder Sinkflug.
- Drehung um die eigene Achse nach links oder nach rechst.
- Gleiten bei konstanter Höhe nach links, nach rechts, nach vorne oder nach hinten.



Schwebeflug Steigflug Drehung



Gleiten: vorwärts



Gleiten: seitlich

Wie geht das?

- Was muss man tun?
- Wer löst das algorithmische Problem?
- Was läuft dabei im Steuerungssystem ab?
- Wie oft muss man im Steuerungssystem etwas tun?
- Muss man das regelmäßig tun?

Ablauftabellen

Was ist ein Ablauftabelle? $\leadsto$ im wesentlichen ein Stundenplan

$$f : \mathcal{T} \mapsto \mathbb{Z} \times \mathbb{Z}$$

$\mathcal{T} = \{T_1, T_2, \ldots, T_x\}$	eine Menge von Aufgaben
$(t_1, t_2) \in \mathbb{Z} \times \mathbb{Z}$	eine Menge von Zeitintervallen

Wie kann man Ablauftabellen darstellen? $\leadsto$ Liste von Tupeln

- Aufgabe
- Startzeitpunkt
- eine Aufgabe kann auch mehrmals auftreten

$Aufgabe_1$	$Zeitpunkt_1$
$Aufgabe_2$	$Zeitpunkt_2$
...	...
$Aufgabe_n$	$Zeitpunkt_n$

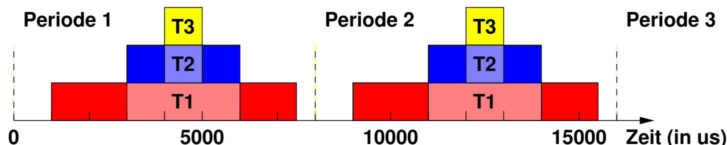
Ausführung von Ablauftabellen

Es existieren zwei Ausführungsmodelle für Ablauftabellen

- präemptive Ausführung
  - verschiedene Aufgaben können sich gegenseitig verdrängen
- nicht-präemptive Ausführung
  - verschiedene Aufgaben werden *ungestört* ausgeführt

Präemptive Ausführung von Ablauftabellen

Aufgaben, die sich in Ausführung befinden, können von später gestarteten Aufgaben verdrängt werden:



Konsequenz

- Erhaltung des Zustands
- zeitliche Beeinflussung
- nebenläufiger Datenzugriff
- strikte Stapelbildung (eher Prozeduraufruf als Faden)

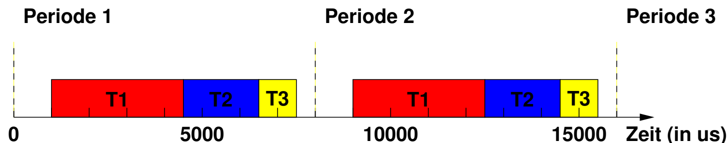
Terminüberschreitung

⇝ Abbruch

- Prozeduraufruf entfernen
- der muss nicht oben auf dem Stapel liegen

Nicht-präemptive Ausführung von Ablauftabellen

- sequentielle Aneinanderreihung von Aufgaben
- keine Unterbrechung oder Verdrängung



Konsequenz

- keine Zustandsicherung
- keine zeitliche Beeinflussung
- kein nebenläufiger Datenzugriff

Ablauftabelle

- muss berechnet werden
- ⇝ ist **NP-vollständig**
- evtl. Aufteilung von Aufgaben notwendig

Aufgabe des Schedulers

- Abarbeiten der Ablauftabelle
  - Aufgaben an den vereinbarten Zeitpunkten einlasten

- Einlastung erfolgt nicht-präemptiv
  - laufende Aufgaben werden ggf. abgebrochen
  - einzulastende Aufgabe starten
  - keine weitere Fehlerbehandlung

- Woher kennt der Scheduler den nächsten Einlastungszeitpunkt?

☞ Zeitgeber (engl. *timer*) stellt diese Information zur Verfügung

Wie kann man den Zeitgeber ansprechen?

- Abfragebetrieb vs. Unterbrecherbetrieb
- Möglichkeiten im Unterbrecherbetrieb:

festes Zeitraster $\rightsquigarrow$ Unterbrechungen in regelmäßigen Abständen

+ einfacher zu implementieren, funktioniert immer
+ einmalige Initialisierung des Zeitgebers
+ relativ flexibel
– Overhead durch unnötige Unterbrechungen

variables Zeitraster $\rightsquigarrow$ Unterbrechung beim nächsten Eintrag

+ kein Overhead durch unnötige Unterbrechungen
– funktioniert bei sehr großen Zeitspannen nicht
– wiederholte Initialisierung des Zeitgebers

☞ Unterbrecherbetrieb mit festem Zeitraster

☞ Mit welcher Frequenz erzeugt man Unterbrechungen $\rightsquigarrow$ GGT

Aufgabenstellung

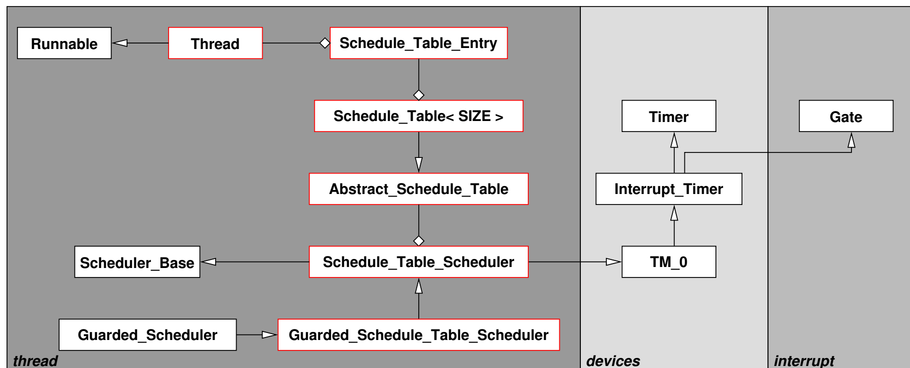
Grundlegende Übungen

- Aufgabenstellung $\rightsquigarrow$ Eigenständig, siehe Webseite
- Vorgabe $\rightsquigarrow$ Funktionsfähiges ezStubs
- Testfälle und Problembeschreibung

Erweiterte Übungen

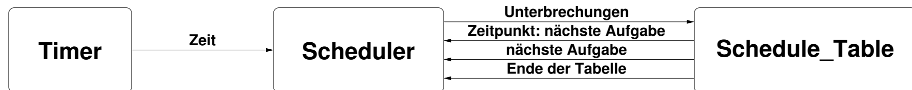
- Aufgabenstellung $\rightsquigarrow$ nachfolgende Folien
- Vorgabe $\rightsquigarrow$ Schnittstellen und Teile von ezStubs
- Schedule Table Scheduler und Testfälle

Klassenhierarchie



- Erweiterte Übung: Schedule Table Scheduler
- rot eingerahmte Klassen $\rightsquigarrow$ Implementierung bzw. Ergänzung

Überblick



- der Scheduler *kennt* die Unterbrechungen des Zeitgebers
  - der Scheduler implementiert die Unterbrechungsbehandlung

- die Ablauftabelle
  - weiß wann ihr Ende erreicht wurde
  - weiß wann die nächste Aufgabe dran ist
  - weiß welche Aufgabe als nächstes dran ist

Systemschnittstelle

- wird von EZStubs benutzt um den Scheduler zu *aktivieren*
- Systemschnittstelle:
  - `void reschedule()`
  - `void start()` **(darf nicht zurückkehren!!!)**
  - Klasse `Scheduler_Base`

Klasse `Scheduler_Base`

```
class Scheduler_Base {
public:
  // Flag setzen:  Thread-Wechsel
  set_need_reschedule();
  // Flag zuruecksetzen: Thread-Wechsel
  clear_need_reschedule();
  // Flag abfragen: Thread-Wechsel
  get_need_reschedule();
};
```

Aktivierung

- beim Verlassen des Kerns in `Guard::leave()`

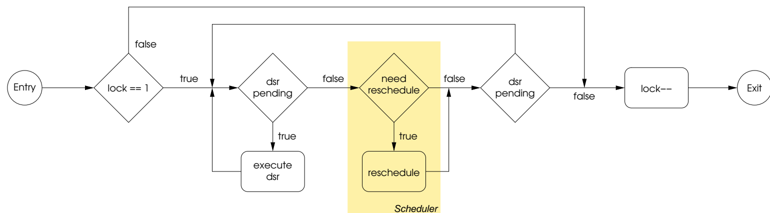


Abbildung: Guard::leave()

- nach der Benutzerdefinierten Startfunktion `user_main()`: Aufruf von `start()`
- wird in EZStubs nicht verwendet: explizit in blockierenden Systemaufrufen

Stack

- ARM-Prozessoren haben mehr als einen Stack Pointer
  - hängt vom jeweiligen Prozessor-Modus ab
  - Threads laufen auf dem System-Stack
  - Interrupts laufen auf dem IRQ- (ISRs) und Supervisor-Stack (DSRs)
- Fadenkontext kann nicht auf dem IRQ-Stack abgelegt werden
- im Interrupt-Handler sind Umschaltungen notwendig

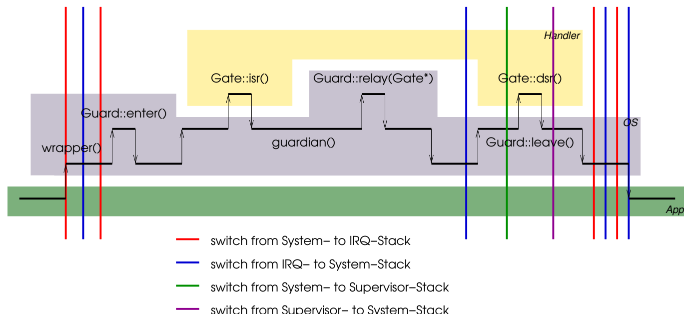


Abbildung: Umschaltungen des Stacks in der Unterbrechungsbehandlung

Stack - `Guard::leave()`

- nur DSRs werden auf dem IRQ-Stack abgearbeitet

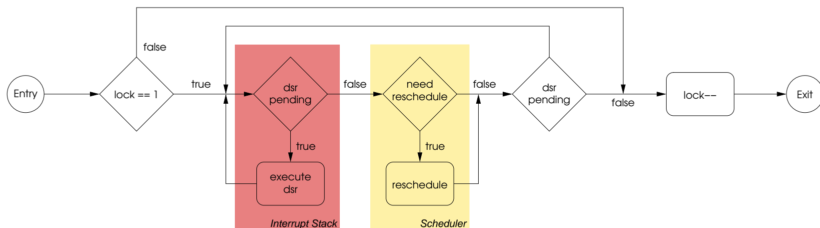


Abbildung: Guard::leave()

Klasse `Thread`

- keine `Coroutine`
  - keine Verdrängung
  - kein Blockieren
  - ☞ nicht notwendig
- nur `Runnable`
  - muss nur die Methode `Runnable::action()` implementieren
  - Methode wird in der Anwendung implementiert

Klassen `Schedule_Table< int L >` und `Abstract_Schedule_Table`

- Paar aus abstrakter Basisklasse und Template-Klasse
  - statische Speicherallokation
    - ☞ Array der Größe L in der Template-Klasse
  - Ablauftabellen verschiedener Größe
    - ☞ verschiedene Instanzen der Templateklasse
    - die verschiedenen Template-Instanzen sind verschiedene Typen
  - ⇝ gemeinsame Schnittstelle durch die abstrakte Basisklasse
- einzelnen Elemente der Tabelle
  - Typ: `Schedule_Table_Entry` ⇝ C-Struktur
  - Definition in der Klasse `Abstract_Schedule_Table`
- Was kommt in die abstrakte Klasse, was in die Template-Klasse?
  - direkter Zugriff auf die Datenstruktur ⇝ Template-Klasse
  - darauf aufbauende Operationen ⇝ abstrakte Klasse

Schnittstelle

- Konstruktor: ❶ `Schedule_Table()`
  - Frequenz der Ablauftabelle ⇝ Wiederholungsperiode
  - Stapelspeicher für die Abarbeitung dieser Ablauftabelle verwendet
- Tupel hinzufügen: ❶ `Schedule_Table< L >::add()`
  - Eintragen der Aufgabe und des zugehörigen Startzeitpunkts
- Tupel lesen: `Schedule_Table< L >::get_entry()`
  - Liefert den entsprechenden Tupel für einen gegebene Index
- Tabelle initialisieren: `Schedule_Table< L >::init()`
  - Zeitraster berechnen ⇝ GGT
  - Startzeitpunkte auf Zeitgeberunterbrechungen abbilden
- nächste Aufgabe: ❶ `Abstract_..._Table::next_thread()`
  - liefert die nächste Aufgabe, die ausgeführt werden soll

Klasse `Schedule_Table_Scheduler`

- alle Aufgaben verwenden denselben dem Stapelspeicher
  - es gibt keine überlappte Ausführung

- erbt von der Klasse `TM_0`
  - implementiert die Unterbrechungsbehandlung dieses Zeitgebers

☞ Methoden ❶ `isr()` und ❶ `dsr()` müssen implementiert werden

Unterbrechungsbehandlung

- ISR: ❶ `Schedule_Table_Scheduler::isr()`
  - Unterbrechungen mitzählen
  - falls nötig, `set_need_reschedule()` aufrufen
- DSR: ❶ `Schedule_Table_Scheduler::dsr()`
  - nichts

Benutzerschnittstelle

- Ablauftabelle setzen: ❶ `...::set_schedule_table()`
  - setze initiale Ablauftabelle
  - Ablauftabelle initialisieren
  - Zeitgeber initialisieren
  - **vor** `Schedule_Table_Scheduler::start()`
- Ablauftabelle wechseln: ❶ `...::switch_schedule_table()`
  - merke neue Ablauftabelle
  - führe die alte Ablauftabelle zu Ende aus
  - starte neue Ablauftabelle am Ende der Runde
- Aufgabe beenden: ❶ `...::exit()`
  - auf Unterbrechungen warten

Systemschnittstelle

- Scheduler starten: ❶ `...::start()`
  - Zeitgeber soll periodisch Unterbrechungen erzeugen
  - auf Unterbrechungen warten
    - Achtung: `user_main()` ist synchronisiert
    - ⇝ den kritischen Abschnitt mit `guard.leave()` verlassen!
  - darf nicht zurückkehren

- Aufgabe *wechseln*: ❶ `...::reschedule()`
  - nächsten Faden/nächste Aufgabe bestimmen
  - `need_reschedule` zurücksetzen
  - Kontext initialisieren ⇝ `init_context()`
  - Aufgabe starten ⇝ Kontext laden ⇝ `load_context()`

Wie implementiert man in EZStubs einen Scheduler?

- siehe EZStubs-Einführung
  - Benutzerschnittstelle
  - Systemschnittstelle
  - Methoden der Klasse `Scheduler_Base`:
    - `Scheduler_Base::set_need_reschedule()`
    - `Scheduler_Base::clear_need_reschedule()`
    - `Scheduler_Base::get_need_reschedule()`

Kontextverwaltung

- Routinen in `thread/context.h`
  - `init_context`
    - den Kontext auf dem Stapelspeicher passend initialisieren
    - Achtung: modifiziert den übergebenen Stackpointer!
  - `load_context`
    - den Kontext vom Stapelspeicher laden, die Aufgabe starten
- alle Aufgaben verwenden selben Stapelspeicher
- ☞ Kontextinitialisierung kann Schaden anrichten
  - man überschreibt die Rücksprungadresse von `init_context`
- ☞ mögliche Lösungen
  - `init_context` einbetten $\rightsquigarrow$ kein Rücksprung
  - Stackpointer manipulieren
- in EZStubs kann dieser Fehler nicht auftreten :-)
  - `isr()` und `dsr()` verwenden einem anderen Stapelspeicher