

Aufgabe 2: Aperiodic Task Support

Echtzeitsysteme - Übungen zur Vorlesung

Fabian Scheler, Peter Ulbrich, Niko Böhm

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

16. November 2009

Erweiterung des Quadrokopter-Kontrollsystems

- ▶ Steuerung mit einer **Fernsteuerung**
 - ▶ Entgegennehmen von Steuerbefehlen
 - ▶ Umsetzen der Steuerbefehle
- ▶ Aufnahme von **Bildern** und deren Versand
 - ▶ benutzergesteuert
 - ▶ Pufferung notwendig

Unterschied der Funktionen: **zeitlichen Anforderungen**

- ▶ Steuerkommandos
 - ▶ Umsetzung innerhalb einer bestimmten Zeit unerlässlich
 - ▶ andernfalls ist keine Kontrolle des Quadrokopters möglich

 **harte Termine**

- ▶ Aufnahme und Versand von Bildern
 - ▶ Übertragungszeit i.d.R. relativ egal
 - ▶ lediglich der **QoS** (*Quality of Service*) wird beeinflusst

 **weiche Termine**

Zeitliche Eigenschaften dieser Funktionen

- ▶ **Wie häufig** werden diese Funktionen aktiviert?
 - ▶ hängt vom Verhalten des Benutzers ab
 - ▶ zahlreiche Versuche Benutzerverhalten in Modellen abzubilden
 - 👉 **Wie hilfreich sind diese Modelle?**
 - 👉 **Notfalls:** Beschränkung des Nutzers!
- ▶ **Wie schnell** muss man auf diese Aktivierungen reagieren?
 - ▶ Aufnahme von **Bilder** $\leadsto$ egal
 - ▶ **Steuerkommandos** $\leadsto$ Quadrokopter muss kontrollierbar sein
 - ▶ Rückmeldung auf Eingaben in Echtzeit $\leadsto$ **Benutzer**
 - ▶ Geschwindigkeit des Quadrokopters $\leadsto$ **Dynamik**
 - 👉 **Notfalls:** Beschränkung der Dynamik!
- ▶ Welche Möglichkeiten haben wir bisher damit umzugehen?
 - ▶ Abfragebetrieb (engl. *Polling*) :-)

Lösungsmöglichkeiten

- ▶ Abfragebetrieb (engl. *Polling*)
- ▶ Hintergrundbetrieb (engl. *Background Execution*)
- ▶ Slack-Stealing

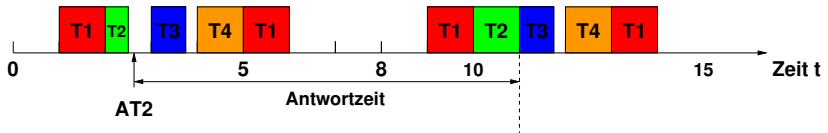
Darstellung an folgendem Beispiel:

Aufgabe	p/a	Periode	Phase	Zw'ankunftszeit	WCET
T1	p	8 ms	1 ms	NA	1 ms
T2	a	NA	NA	x ms	1 ms
T3	p	8 ms	3 ms	NA	0,75 ms
T4	p	8 ms	4 ms	NA	1 ms
T1	p	8 ms	5 ms	NA	1 ms

- ▶ T2 bearbeitet das aperiodische Ereignis AT2

Abfragebetrieb

- ▶ **Periodisches Abfragen:** Muss **T2** bearbeitet werden?
 - ▶ schlimmstenfalls muss eine Periode gewartet werden
- ▶ resultierender Ablauf:



Vorteile

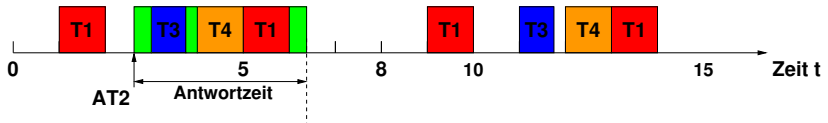
- ▶ periodische Aufgaben
 - ▶ keine Beeinflussung

Nachteile

- ▶ lange Antwortzeit
- ▶ Overhead
 - ▶ periodisches Abfragen
 - ▶ Abtasttheorem (Nyquist-Shannon)

Hintergrundbetrieb

- ▶ **direkte Reaktion** auf nicht-periodische Aufgaben ist möglich
- ▶ Behandlung in der **Ruhezeit** der Ablauftabelle
- ▶ resultierender Ablauf:



Vorteile

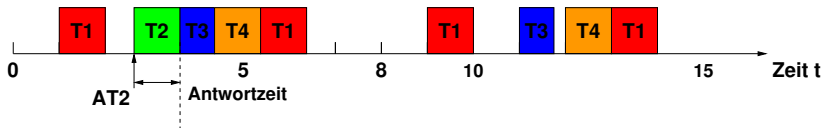
- ▶ bessere Antwortzeiten
- ▶ kein Abfragen notwendig
 ~> kein Overhead

Nachteile

- ▶ **komplexe** Implementierung
- ▶ Betriebsmittel
 - ▶ nebenläufiger Zugriff
 - ▶ problematisch

Slack-Stealing

- ▶ manche periodische Aufgaben haben *Luft nach Hinten*
 - ▶ Fertigstellung deutlich vor der Deadline macht keinen Sinn
- ☞ schiebe die periodische Aufgabe nach hinten
 - ▶ man stiehlt also ihre *Schlupfzeit* (engl. *Slack-Time*)
- ▶ resultierender Ablauf:



Vorteile

- ▶ sehr gute Antwortzeiten
- ▶ periodische Aufgaben
 - ▶ tolerierbare Beeinflussung

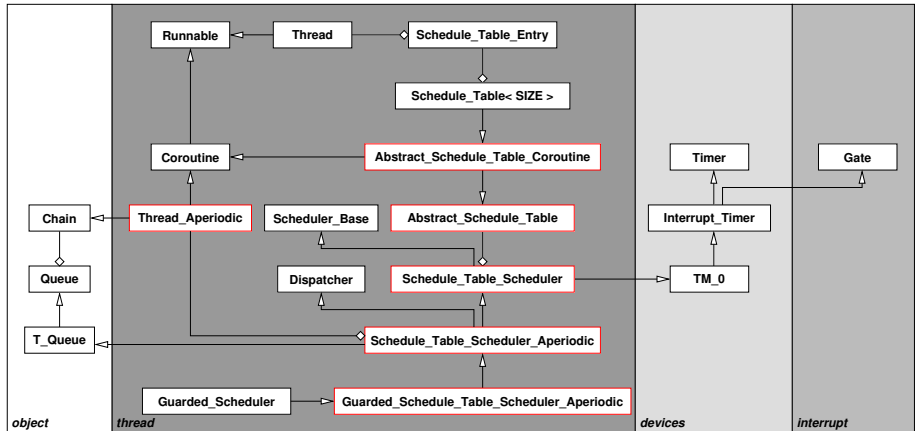
Nachteile

- ▶ **komplexe** Implementierung

Aufgabenstellung

- ▶ Unterstützung für **aperiodische Aufgaben**
 - ▶ **keine** sporadischen Aufgaben
- ▶ Hintergrundbetrieb
 - ▶ **kein** Slack-Stealing
 - ▶ **kein** Polling
- ▶ **aperiodische Aufgaben**
 - ▶ werden in **FIFO-Reihenfolge** abgearbeitet
 - ▶ sind **ein- oder mehrfach aktivierbar**
 - ▶ **Aktivierungsreihenfolge irrelevant**
 - ▶ **aperiodische Aufgabe beendet** $\leadsto$ **erneute Aktivierung**
 - ▶ lassen periodischen Aufgaben den Vortritt

Klassenhierarchie



► **rot eingerahmte Klassen** $\rightsquigarrow$ Implementierung bzw. Ergänzung

Überblick

- ▶ **Grundproblem:** aperiodische Aufgaben müssen **verdrängbar** sein
 - ▶ Zustandssicherung wird notwendig

~> **Grundidee:** verwende Koroutinen

- ▶ für aperiodische Aufgaben ~> **offensichtlich**
- ▶ aber auch auf für periodische Aufgaben ~> **???**
 - ▶ Ablauftabelle führt periodische Aufgaben aus ~> `action()`
 - ▶ Start einer periodischen Aufgabe ~> Einlastung der Ablauftabelle

- ▶ `Schedule_Table` **erbt von** `Abstract_..._Coroutine`

👉 **Vorgaben kopieren**

- ▶ System lässt sich nicht mehr übersetzen
- ▶ Vererbungshierarchie anpassen

Klasse Thread_Aperiodic

- ▶ ist eine Coroutine
 - ☞ kann von periodischen Aufgaben **verdrängt** werden
- ▶ ist ein Kettenglied (Chain)
 - ☞ Abarbeitungsreihenfolge: **FIFO**
- ▶ Vorsicht bei der Typumwandlung
 - ▶ **Mehrfachvererbung**
 - ☞ Verwendung einer Template-Klasse T_Queue
 - ▶ verwendet `static_cast< typename T >()`

Klasse `Abstract_Schedule_Table_Coroutine`

- ▶ ist eine `Coroutine`
 - 👉 Wechsel zwischen aperiodischen Aufgaben und Ablauftabelle
- ▶ Ausführung einer periodischen Aufgabe
 - ~> Ablauftabelle initialisieren (`Coroutine::init()`)
 - ~> Ablauftabelle einlasten
- ▶ Methode `action()`
 - ▶ nächste periodische **Aufgabe holen** ~> `next_thread()`
 - ▶ **Aufgaben ausführen** ~> Methode `action()` aufrufen
 - ▶ periodische Aufgabe ruft zum Schluss `scheduler.exit()` auf!

Klasse `Schedule_Table_Scheduler_Aperiodic`

- ▶ Ausführung **periodischer** und **aperiodischer** Aufgaben
- ▶ Ausführung **periodischer Aufgaben**
 - ▶ immer in `Abstract_Schedule_..._Coroutine::action()`
 - ▶ **Koroutine** wird immer neu gestartet
 - ▶ **Ablaufabelle** $\leadsto$ Aufgabe 1
- ▶ Ausführung **aperiodischer Aufgaben**
 - ▶ können verdrängt werden
 - $\leadsto$ Zustandssicherung und Wechsel zur Ablaufabelle
 - $\leadsto$ anschließend: Fortsetzung der aperiodischen Aufgabe

Benutzerschnittstelle

- ▶ Aufgabe beenden:   `...::exit()`

- ▶ **periodische Aufgabe** beenden
 - ▶ gibt es eine ausführungsbereite aperiodische Aufgabe?
 -  aperiodische Aufgabe ausführen
 -  **Achtung:** wird dieselbe aperiodische Aufgabe erneut ausgeführt
 - ▶ sonst: kritischen Abschnitt verlassen und warten

- ▶ **aperiodischen Aufgabe** beenden
 - ▶ Aufgabe aus dem FIFO entfernen
 -  ggf. Aufgabe neu in den FIFO aufnehmen
 -  Koroutine erneut initialisieren
 - ~> weiter wie beim Beenden einer periodischen Aufgabe

Benutzerschnittstelle

- ▶ aperiodische Aufgabe aktivieren: **IS** `...::ready()`
 - ▶ Faden in den FIFO eintragen
 - ▶  Koroutine initialisieren
 - ▶ ggf. **Aktivierung** des Schedulers
 - ▶ Wie? $\leadsto$ Aufruf von `set_need_reschedule()`
 - ▶ Wann? $\leadsto$ in Ruhephasen
- ▶ Ablaufabelle setzen bzw. wechseln
 - ▶ siehe Aufgabe 1
 - ▶ **Achtung:** **IS** `...::switch_schedule_table()`
 - ▶  Synchronisation notwendig

Systemschnittstelle

- ▶ Scheduler starten: ❶ `...::start()`
 - ▶ Ablaufabelle initialisieren (wie in Aufgabe 1)
 - ▶ Timer soll Unterbrechungen erzeugen
 - ▶ warten
- ▶ Aufgabe *wechseln*: ❷ `...::reschedule()`
 - ▶ nächste periodischen oder aperiodische Aufgabe starten
 - 👉 Sicherung des Zustands einer aperiodischen Aufgabe
 - ~> Klasse `Dispatcher`
 - ~> Wie werden periodische Aufgaben gestartet?
 - ▶ folgende Übergänge sind denkbar
 - ▶ periodisch ~> aperiodisch
 - ▶ aperiodisch ~> aperiodisch
 - ▶ aperiodisch ~> periodisch
 - ▶ (periodisch ~> periodisch)
- 👉 Scheduler muss den Übergang kennen

Synchronisation

- ▶ Implementierung durch **Hüllfunktionen** (engl. *Wrapper*)
 - ▶ in der Klasse `Guarded_Schedule_..._Aperiodic`
 - ▶ **Achtung:** `protected-Vererbung`
 - ↪ Aufrufe müssen explizit durchgereicht werden

```
void exit() {
    Secure secure;
    Schedule_Table_Scheduler_Aperiodic::exit();
}
```

- ▶ **Balance** von `guard.enter()` und `guard.leave()`
- 👉 `main()` **betritt** vor `user_main()` einen kritischen Abschnitt
 - ↪ Aufruf von `guard.enter()`
- 👉 jede Koroutine startet in der Funktion `kickoff()`
 - ▶ `kickoff()` **verlässt** einen kritischen Abschnitt
 - ↪ Aufruf von `guard.leave()`

Vorgabe

- ▶ **Unterbrechungen erzeugen** $\leadsto$ Klasse `Test_Interrupt_0`
 - ▶ Header: `debug/test_interrupt_0.h`
 - ▶ von dieser Klasse ableiten
 - ▶ `isr()` und `dsr()` implementieren
 - ▶ Objekt erzeugen und die Methode `trigger()` aufrufen
 - $\leadsto$ erzeugt eine Unterbrechung, Behandlung im `isr()` und `dsr()`
- ▶ **Koroutinenimplementierung** $\leadsto$ Klasse `Coroutine`
 - ▶ Header: `thread/coroutine.h`
 - ▶ vor der Ausführung initialisieren $\leadsto$ `init()`
- ▶ **Dispatcher** $\leadsto$ Klasse `Dispatcher`
 - ▶ Header: `thread/dispatch.h`
 - ▶ Koroutine **starten**: `Dispatcher::go()`
 - ▶ Laden des Zustands, keine Zustandssicherung
 - ▶ Koroutine **wechseln**: `Dispatcher::dispatch()`
 - ▶ Zustand der laufenden Koroutine sichern, der übergebenen laden
 - ▶ **Achtung**: übergebene Koroutine $\neq$ laufende Koroutine!

Weiter gedacht

- ▶ Was bräuchte man noch für Slack-Stealing?
 - 👉 Wissen über **Ausführungszeiten** und **Deadlines** zur Laufzeit
- ▶ Was bräuchte man noch für sporadische Ereignisse?
 - 👉 **Übernahmeprüfung**, **Ausführungszeiten**, **Deadlines**
- ▶ entstandene Problematik: **Unterbrechungen**
- ▶ **schwierig**: gemeinsame Betriebsmittel
 - ▶ geteilt von periodische **und** aperiodische Aufgaben
 - 👉 Warum?