

Aufgabe 3: Multi-Level-Queue-Scheduler

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

1. Dezember 2009

Grundlegende Problematik zeitgesteuerter Systeme

- ▶ **vollständiges a-priori Wissen** notwendig
 - ▶ Lastparameter aller Ereignisse
 - ▶ Periode, Phase, Jitter, WCET, Termine, ...
- ~ ist nicht immer verfügbar :-(

☞ teilweise hat das mit *Faulheit* zu tun ...

- ▶ **unzureichende Analyse** der physikalischen Umgebung
- ▶ zeitliches Verhalten physikalischer Objekte ~ **hohe Komplexität**

☞ teilweise hat man aber einfach keine Chance ...

- ▶ **physikalisches Objekt** ist nicht exakt quantifizierbar
- ▶ Abhängigkeit von **Benutzereingaben**
 - ▶ beeinflusst Auslösezeitpunkte, WCETs, ...
- ▶ sich **ändernde** oder **unbekannte Anforderungen**
 - ▶ z.B. in lang andauernden Projekten: Space Shuttle [?]

Folgerung

~ zeitgesteuerte Systeme und **unsicheres Wissen** ...

- ▶ ... **vertragen sich nicht!**
- ▶ ... gehen teilweise gar nicht oder ...
- ▶ ... führen zu teilweise ineffizienten Implementierungen
 - ▶ siehe Abfragebetrieb und Abtasttheorem nach Nyquist-Shannon

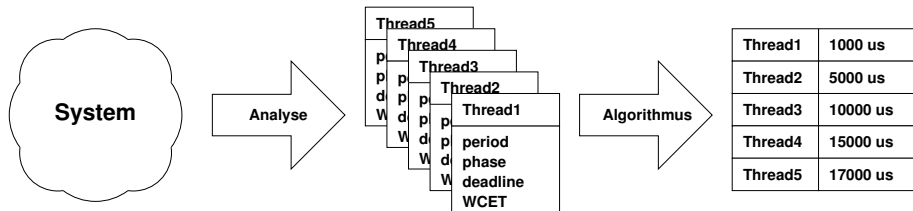
☞ **ereignisgesteuerte Systeme**

- ▶ können auch unsicheres Wissen verwerten
 - ~ zur **Laufzeit** wird daraus oftmals **sicheres Wissen**
 - ~ **Einplanung**, **Koordinierung** finden zur **Laufzeit** statt

☞ u.U. wird aus unsicherem Wissen sicheres Wissen ...

- ▶ schon während der Entwicklung
- ~ Rückkehr zu einem zeitgesteuerten System? ;-)
- ~ Geht das überhaupt???
- ~ Welche Probleme handelt man sich ein???

Zeitsteuerung



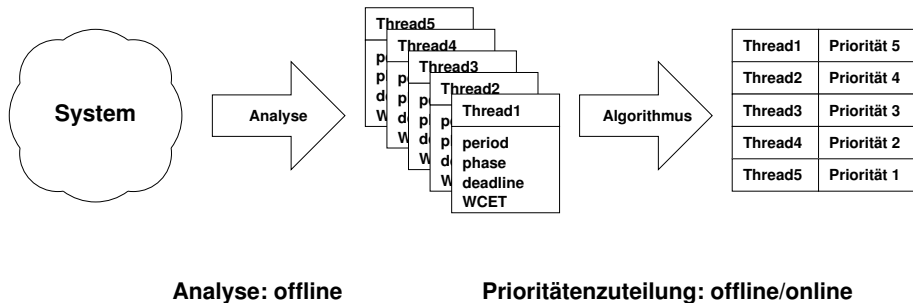
Analyse: offline

Ablaufplanung: offline

- ▶ Abbildungen während der **Entwicklung**
 - ▶ Ereignisse $\mapsto$ Aufgaben
 - ▶ Aufgaben $\mapsto$ Startzeitpunkte
 - ▶ aufgeschrieben in einer **Ablauftabelle**

Ereignissteuerung

Abbildungen während der Entwicklungszeit

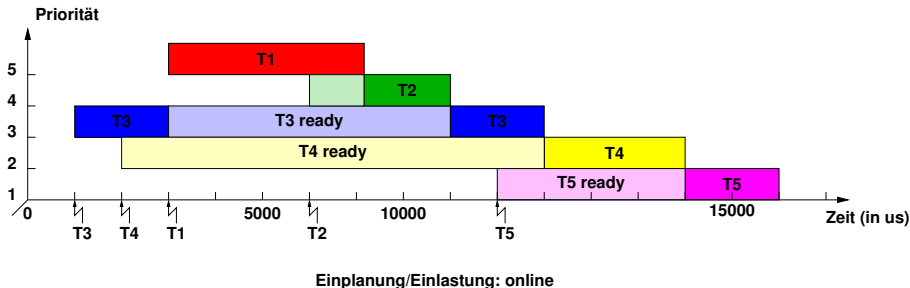


► Abbildungen während der **Entwicklung**

- Ereignisse $\mapsto$ Aufgaben
- Ereignisse $\mapsto$ Prioritäten
 - **statische Prioritäten** $\leadsto$ RMA, DMA

Ereignissteuerung

Abbildungen zur Laufzeit



► Abbildungen während der Laufzeit

- Ereignisse $\mapsto$ Prioritäten
 - dynamische Prioritäten für **Aufgaben** $\leadsto$ EDF, LRT
 - **Wichtig:** Prioritätsgefüge ändert sich nicht
 - dynamische Prioritäten für **Arbeitsaufträge** $\leadsto$ LST
 - **Wichtig:** Prioritätsgefüge kann sich ändern
- Aufgaben $\mapsto$ Startzeitpunkt

Eigenschaften ereignisgesteuerter Systeme

- ▶ Unterstützung für **periodische** und **nicht-periodische Ereignisse**
- ▶ **Verdrängbarkeit:**
 - ▶ voll-präemptiv
 - ▶ präemptiv
 - ▶ nicht-präemptiv
 - ▶ gemischt-präemptiv
- ▶ Synchronisation **explizit** notwendig
- ▶ **Ablaufplan wird zur Laufzeit berechnet**

Vorteile

- ▶ (nicht-)periodische Aufgaben
- ▶ weniger a-priori Wissen
- ▶ kein Polling
- ▶ Flexibilität

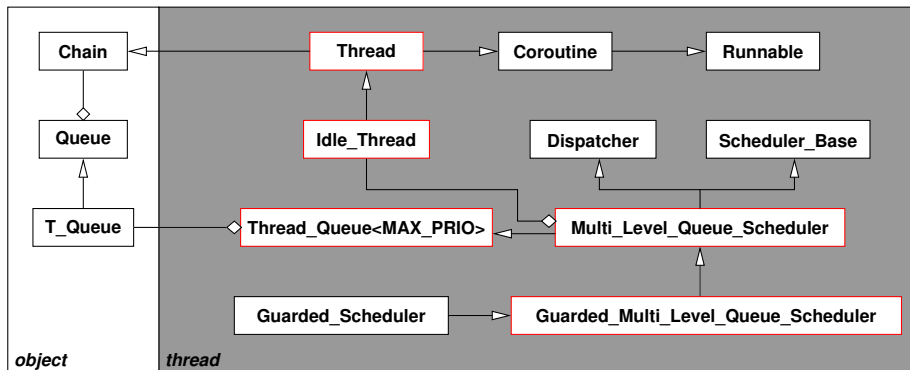
Nachteile

- ▶ Laufzeitsystem
- ▶ Synchronisation
- ▶ Flusskontrolle

Aufgabenstellung

- ▶ ereignisgesteuerter Online-Scheduler
- ▶ basierend auf statischen Prioritäten
 - ▶ 0 - niedrigste Priorität, n - höchste Priorität
- ▶ Fäden ...
 - ▶ sind ein- oder mehrfach aktivierbar
 - ▶ Aktivierungsreihenfolge irrelevant
 - ▶ Faden beendet $\leadsto$ erneute Aktivierung
 - ▶ sind durch Unterbrechungsbehandlungen aktivierbar
 - ▶ können andere Fäden aktivieren und beenden
 - ▶ können ...
 - ▶ sich selbst beenden
 - ▶ die Kontrolle über den Prozessor abgeben
- ▶ Ruhephasen: Ausführung eines `Idle_Thread`

Klassenhierarchie



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

Klassen Thread und Idle_Thread

► Klasse Thread

- ist eine Koroutine $\leadsto$ erbt von der Klasse Coroutine
- ist ein Kettenglied $\leadsto$ erbt von der Klasse Chain
- hat eine statische Priorität
- weiß die Anzahl der Aktivierungen $\leadsto$ Zähler

► Klasse Idle_Thread

- ist ein Thread
- gibt immer die Kontrolle ab
- kann immer verdrängt werden

Klasse Thread_Queue< int MAX_PRIO >

Implementierungen verschiedener Mächtigkeit

Template $\leadsto$ statisch konfigurierbare Anzahl von Prioritätsebenen

Variante 1: ein Faden/Priorität, Faden ist **einfach aktivierbar**

☞ Array bzw. Bitmap

Variante 2: mehrere Fäden/Priorität, Faden ist **einfach aktivierbar**

☞ Array aus FIFO-Listen

Problematik: Termin $>$ min. Zw'nankunftszeit $\leadsto$ **Mehrfachaktivierungen**

Variante 3: mehrere Faden/Priorität, Faden ist **mehrfach aktivierbar**

☞ Variante 2 + **Aktivierungszähler**

Problematik: Aktivierungsreihenfolge (notwendig für AUTOSAR OS)

Variante 4: mehrere Fäden/Priorität, Faden ist **mehrfach aktivierbar**,
Reihenfolge

☞ **mehrfachverkettete Liste**, **mehrdimensionales Array** ...

Klasse Thread_Queue< int MAX_PRIO >

Schnittstelle

- ▶ Faden am Anfang der Liste: `...::peek()`
 - ▶ welcher Faden steht am Anfang der Liste
 - ▶ dieser Faden wird i.d.R. auch gerade ausgeführt
- ▶ Faden am Anfang der Liste entfernen: `...::dequeue()`
 - ▶ evtl. macht es Sinn den Faden auch zurückzugeben
- ▶ beliebigen Faden aus der Liste entfernen: `...::remove()`
- ▶ Faden in die Liste einfügen: `...::enqueue()`

Klasse Multi_Level_Queue_Scheduler

Benutzerschnittstelle

Aufgabe: den höchst-prioren Faden auswählen und ausführen

- ▶ liefert den aktuell ausgeführter Faden: **1** `...::current()`
- ▶ einen Faden aktivieren: **1S** `...::add()`
 - ▶ Faden in die **Bereitliste** einfügen, **mehrfache Aktivierung** beachten
 - ▶ **Scheduler aktivieren?** $\leadsto$ `set_need_reschedule()`
- ▶ die Kontrolle über den Prozessor abgeben: **1S** `...::yield()`
 - ▶ laufenden Faden **ans Ende der Bereitliste** stellen
 - ▶ **Scheduler aktivieren?** $\leadsto$ `set_need_reschedule()`
 - ▶ **Achtung:** kein dispatch auf sich selbst!
- ▶ den laufenden Faden beenden: **1S** `...::exit()`
 - ▶ aus der **Bereitliste entfernen**, **mehrfache Aktivierung** beachten
 - ▶ **Scheduler aktivieren!** $\leadsto$ `set_need_reschedule()`
- ▶ anderen Faden beenden: **1S** `...::kill()`
 - ▶ aus der **Bereitliste entfernen**, **mehrfache Aktivierung** beachten
 - ▶ laufenden Faden beenden $\leadsto$ `...::exit()`
 - ▶ **Scheduler aktivieren?** $\leadsto$ `set_need_reschedule()`

Klasse Multi_Level_Queue_Scheduler

Systemschnittstelle

- ▶ Scheduler starten: ❶ `...::start()`
 - ▶ höchst-prioren, lafbereiten Faden auswählen
 - ▶ diesen Faden ausführen
- ▶ Neueinplanung durchführen: ❶ `...::reschedule()`
 - ▶ siehe `...::start()`
- ▶ Idle_Thread setzen: ❶ `...::set_idle_thread()`
 - ▶ den Idle_Thread setzen
 - ▶ **Achtung:** muss vor `...::start()` aufgerufen werden!

Hinweise

► Prioritätszahl

- **anwendungsspezifisch** $\leadsto$ muss vom Benutzer bestimmt werden
 - Benutzer legt Scheduler-Objekt selbst an
 - Konfigurationswerkzeug
 - **Hier:** hart kodiert - 8 Prioritätsebenen

► Wo initialisiert man die Koroutine

- beim **Einfügen** in die Bereitliste
- $\leadsto$ alle Fäden in der Bereitliste sind initialisiert

Weiter gedacht

Laufzeitkomplexität

- ▶ Wo hat der Scheduler konstantes Laufzeitverhalten, wo nicht?
- ▶ **konstant**
 - ▶ `enqueue()` und damit `add()`
 - ▶ `dequeue()` und damit `yield()`
 - ▶ Zugriff auf das Array der FIFO-Listen, d.h. `peek()`
- ▶ **linear**
 - ▶ Bestimmung der maximalen Priorität und damit `exit()`
 - ▶ `remove()` und damit `kill()`
- ▶ **Fazit:**
 - ▶ Verdrängung, Aktivierung etc. ist **konstant**
 - ▶ maximale Priorität muss nicht extra bestimmt werden
 - ▶ Beenden ist **linear**
 - ▶ maximale Priorität muss gesondert bestimmt werden

Literatur