

Aufgabe 5: Deferrable Server

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

18. Januar 2010

Bekannte Lösungsansätze

Hintergrundbetrieb (engl. *Background Execution*)

- ▶ siehe Aufgabe 2
- ▶ Abarbeitung während der Untätigkeitsintervalle

Vorteile

- ▶ keine Beeinflussung

Nachteile

- ▶ schlechte Antwortzeiten

Unterbrecherbetrieb (engl. *Interrupt-Driven Execution*)

- ▶ sofortige Behandlung nicht-periodischer Ereignisse

Vorteile

- ▶ sehr gute Antwortzeiten

Nachteile

- ▶ starke Beeinflussung

Nicht-periodische Ereignisse

Behandlung auch in ereignisgesteuerten Systemen problematisch:

- ▶ der **Zeitpunkt** des Auftretens ist **unbekannt**
- ▶ der **Einfluss** auf das restliche System muss kontrollierbar sein

Abstraktion für die Behandlung nicht-periodischer Ereignisse

- ▶ gute mittlere **Antwortzeiten**
- ▶ geringe, kontrollierbare **Beeinflussung** periodischer Aufgaben
- ▶ effiziente **Akzeptanztests**

Bekannte Lösungsansätze

Abfragender Zusteller (engl. *Polling Server*)

- ▶ siehe Aufgabe 1 und 2
- ▶ Charakteristische Größen
 - ▶ Periode p und Ausführungsbudget e

Vorteile

- ▶ keine Beeinflussung

Nachteile

- ▶ schlechte Antwortzeiten

Slack-Stealing

- ▶ Kombination von Hintergrund- und Unterbrecherbetrieb

Vorteile

- ▶ gute Antwortzeiten
- ▶ tolerierbare Beeinflussung

Nachteile

- ▶ komplexe Implementierung
- ▶ für zeitgesteuerte Systeme

Bandbreite bewahrende Zusteller

engl. *Bandwidth Preserving Servers*

- ▶ verwandt mit dem **Polling Server**, charakterisiert durch
 - ▶ eine **Wiederherstellungsperiode** p
 - ▶ ein **Ausführungsbudget** e
- ▶ **Unterschied:** Restbudget verfällt nicht
- ▶ zusätzlich:
 - ▶ **Konsumregeln** (engl. *consumption rules*)
 - ▶ **Wiederherstellungsregeln** (engl. *replenishment rules*)
- ▶ verschiedene Varianten Bandbreite bewahrender Zusteller
 - ▶ Deferrable Server
 - ▶ Simple Sporadic Server
 - ▶ SpSL Sporadic Server
 - ▶ Constant Utilization Server
 - ▶ Fair Queueing Server
- ▶ zusätzlich z.B. mit Hintergrundbetrieb kombinierbar

Deferrable Server

Einfachste Implementierung eines Bandbreite bewahrenden Zustellers

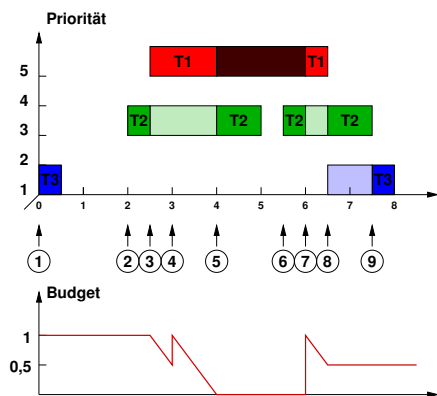
Konsumregel Für jede Zeiteinheit der Ausführung wird eine Zeiteinheit des Budgets verbraucht.

Wiederherstellungsregel Zu den Zeitpunkten kp , $k = 0, 1, 2, 3, \dots$ wird das Budget komplett wiederhergestellt.

Beispiel: Gegeben sei ein System mit folgenden Parametern

Bezeichnung	Phase	Period	WCET	Priorität
T1 (Def. Server)		3	1	5
T2	2	3,5	1,5	3
T3	0	6,5	0,5	1

Beispiel: Ablauf

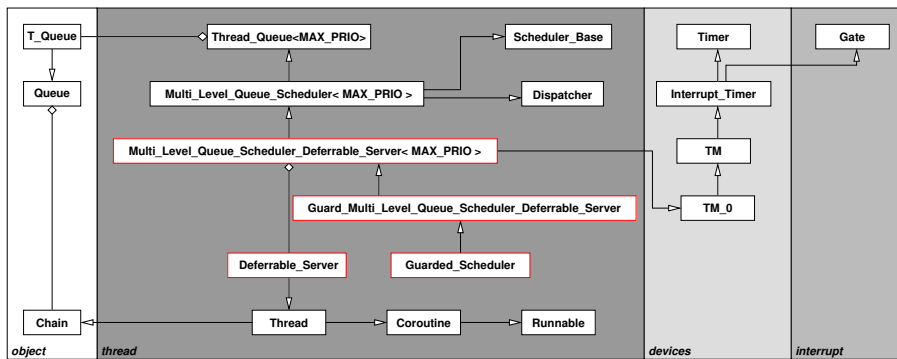


1. Auslösung & Ausführung T3
2. Auslösung & Ausführung T2
3. aperiodisches Ereignis
~ Deferrable Server T1
4. Budget T1: Wiederherstellung
5. Budget T1: aufgebraucht
~ Fortsetzung T2
6. Auslösung & Ausführung T2
7. Budget T1: Wiederherstellung
~ Fortsetzung T1
8. T1 fertig, Auslösung T3
9. T2 fertig
~ Ausführung T3

Verhalten der Schnittstelle

- ▶ **Aktivierung:** `add()`
 - ▶ der Server benötigt Budget um ausgeführt zu werden
- ▶ **Beenden:** `exit()`, `kill()`
 - ▶ der Server behält das restliche Budget
- ▶ **Kontrolle abgeben:** `yield()`
 - ▶ der Server verliert sein Budget
- ▶ **Registrieren:** `enlist()`
 - ▶ den Server beim Scheduler anmelden
- ▶ Server wird als **eigenständiger Faden** implementiert
- ▶ maximal 1 Server pro Priorität
- ▶ Server hat **Vorrang gegenüber Threads** derselben Priorität
- ▶ Server dürfen **keinen Mutex** verwenden

Klassenhierarchie



- ▶ **rot eingrahmte Klassen** ~ Implementierung bzw. Ergänzung

Klassen Deferrable_Server

- ▶ Spezialisierung eines Thread
- ▶ Verfügt über Budget und Wiederherstellungsperiode
- ▶ **1** Konstruktor
 - ▶ Budget und Wiederherstellungsperiode initialisieren
- ▶ Budget wiederherstellen: `....::replenish()`
 - ▶ stellt das Budget des Deferrable Servers wieder her
- ▶ Budget teilweise verbrauchen: `....::consume(us_type)`
 - ▶ verbraucht einen angegebenen Teil des Budgets
- ▶ Budget komplett verbrauchen: `....::consume()`
 - ▶ braucht das Budget restlos auf
 - ▶ Warum könnte man das brauchen?
- ▶ Budget: `us_type ....::get_current_budget()`
- ▶ Periode: `us_type ....::get_replenishment_period()`

Klasse MLQ_Scheduler_Defferrable_Server

Grundsätzliche Aufgaben

- ▶ Verwaltung von Threads und Deferrable Servern
- ▶ maximal 1 Deferrable Server pro Priorität
- ▶ maximal 1 Aktivierung pro Deferrable Server
- ▶ **Überwachung** der Ausführungsbudgets
- ▶ **Wiederherstellung** der Ausführungsbudgets

Klasse MLQ_Scheduler_Defferrable_Server

Benutzerschnittstelle

- ▶ Server auslösen: **1S** `....::add(Deferrable_Server*)`
 - ▶ aktiviert den Deferrable Server
 - ▶ analog zur Aktivierung von Threads
- ▶ Server beenden: **1S** `....::exit()`
 - ▶ der Server beendet seine Ausführung
 - ▶ das restliche Budget bleibt erhalten
- ▶ Server beenden: **1S** `....::kill(Deferrable_Server*)`
 - ▶ der Server beendet seine Ausführung
 - ▶ das restliche Budget bleibt erhalten
- ▶ Kontrolle abgeben: **1S** `....::yield()`
 - ▶ der Server verzichtet auf den Prozessor
 - ▶ das restliche Budget geht verloren
- ▶ Server registrieren: `....::enlist(Deferrable_Server*)`
 - ▶ Registriert den Server beim Scheduler ~ Wofür?

Klasse MLQ_Scheduler_Deferrable_Server

Systemschnittstelle

- ▶ Scheduler starten: `...::start()`
 - ▶ prinzipiell wie in [Aufgabe 3](#)
 - ▶ zusätzlich:
 - ▶ relative Wiederherstellungszeitpunkte der Server bestimmen
 - ▶ Timer: IRQ beim nächsten relativen Wiederherstellungszeitpunkt
 - ▶ Zeitmessung starten ~ [Warum?](#)
- ▶ nächsten Faden bestimmen: `...::schedule()`
 - ▶ prinzipiell wie in [Aufgabe 3](#) und zusätzlich:
 - ▶ Zeitmessung stoppen und Ergebnis auslesen ~ [Warum?](#)
 - ▶ nächsten relativen Wiederherstellungszeitpunkt aktualisieren
 - ▶ falls notwendig: Budget verbrauchen ~ [Wann?](#)
 - ▶ als nächstes kommt ein Deferrable Server
 - ▶ Timer neu programmieren ~ Budget des Servers überwachen
- ▶ Einplanung durchführen: `...::reschedule()`
 - ▶ wie in [Aufgabe 3](#)
 - ▶ Einplanungsentscheidung umsetzen: nächsten Faden einlasten

Klasse MLQ_Scheduler_Deferrable_Server

Unterbrechungsbehandlung

- ▶ Unterbrechungsbehandlung: `...::isr()`
 - ▶ Unterbrechung bestätigen
- ▶ nachgelagerte Unterbrechungsbehandlung: `...::dsr()`
 - ▶ Zeitmessung stoppen und Ergebnis auslesen
 - ▶ **Möglichkeit 1:** Budgets auffrischen
 - ▶ falls notwendig: Budget verbrauchen ~ [Wann?](#)
 - ▶ nächsten Wiederherstellungszeitpunkt bestimmen
 - ▶ Budgets auffüllen
 - ▶ **Möglichkeit 2:** Budget ist verbraucht
 - ▶ nächsten Wiederherstellungszeitpunkt bestimmen
 - ▶ Budget vollständig aufbrauchen
 - ▶ `set_need_reschedule()` ~ [Warum?](#)
 - ▶ Timer neu programmieren ~ Wiederherstellen/Budget überwachen
 - ▶ Zeitmessung starten

Zeitmessung

- ▶ Klasse `Timer`
 - ▶ Zeitmessung starten: `...::start_time()`
 - ▶ Zeitmessung stoppen: `...::stop_time()`
 - ▶ Ergebnis auslesen: `...::get_time()`
- ▶ Zeitmessung kann auch über Unterbrechungen hinweg erfolgen
- ▶ Solange darf Timer nicht neu programmiert werden