Grundlegende Übungen
Aufgabenblatt 4 - Fadensynchronisation

Peter Ulbrich, Martin Hoffmann

13. Dezember 2010

Ausgangspunkt für diese Aufgabenstellung ist ein ereignisgesteuerter Ablaufplaner, der Fadensynchronisation mittels eines *Stack-Based Priority Ceiling Protocols* unterstützt. Der Ablaufplaner hat dabei die Eigenschaften, wie sie in den Tafelübungen zu Aufgabe 4 besprochen wurden.

Aufgabe 1 Gegeben seien die vier Aufgaben in der folgenden Tabelle:

Aufgabe	p/a	Periode	WCET	Blockierungszeit
T1	p	10 ms	2 ms	1 ms
T2	p	5 ms	1 ms	NA
T3	p	40 ms	7 ms	4.5 ms
T4	p	15 ms	6 ms	NA

Zwischen den Aufgaben T1 und T3 besteht ein *Produzenten-Konsumenten-Verhältnis*. Aufgabe T1 stellt ein Zwischenergebnis bereit, das von T3 weiterverarbeitet wird. Weil T1 diese Zwischenergebnisse schneller erzeugt, als T3 sie abarbeitet, müssen sie in einem Ringpuffer zwischengespeichert werden. Bei diesem Puffer handelt es sich um eine gemeinsame Datenstruktur, der Zugriff hierauf muss also synchronisiert werden.

a) Implementieren Sie die Aufgaben T1 bis T4. Zur periodischen Aktivierung der Aufgaben verwenden sie einen Zeitgeber. Vergeben Sie die Ausgangsprioritäten der einzelnen Aufgaben gemäß des *Rate Monotonic Algorithm. Verwenden Sie zur Synchronisation von T1 und T3 einen Mutex.*

b) Nehmen Sie an, der Mutex wäre als einfacher Semaphor (wird von EZStubs nicht unterstützt!) implementiert. Welches Problem kann bei der Ausführen mit den o.g. Parametern auftreten? Welche Möglichkeiten kennen Sie, diese Auswirkungen zu vermeiden? **Hinweis:** T4 beeinflusst das Verhältnis von T1 und T3.

c) Nehmen Sie an, das System würde mittels *Verdrängungssteuerung* (NPCS, siehe Kapitel 7 "Zugriffskontrolle", Folie 7-13) synchronisiert. Welche Folgen hätte dies für die Echtzeiteigenschaften des Systems? **Annahme:** $Periode = Termin$. Implementieren Sie die Aufgaben und zeigen Sie dies mit einer Messung. **Hinweis:** Verdrängungssteuerung lässt sich mit Hilfe der Funktionen `guard.enter()` und `guard.leave()` implementieren.

Aufgabe 2 Gegeben seien die drei Aufgaben in der folgenden Tabelle:

Aufgabe	p/a	Periode	WCET	Blockierungszeit
T1	p	10 ms	1 ms	0.5 ms
T2	p	10 ms	1 ms	0.5 ms
T3	p	20 ms	2 ms	0.5 ms

Auch hier besteht zwischen den Aufgaben (T1;T2) und T3 ein Produzenten-Konsumenten-Verhältnis. Die Aufgaben T1 und T2 stellen ihre Zwischenergebnisse in einem Ringpuffer bereit der von T3 ausgelesen wird.

a) Implementieren Sie die Aufgaben T1, T2 und T3 wie in Teilaufgabe 1. Vergeben Sie auch hier die Prioritäten der einzelnen Aufgaben gemäß des *Rate Monotonic Algorithm*. Synchronisieren Sie den Ringpuffer mittels einer Mutex.

b) Nehmen Sie an, die Aufgabe T2 gibt innerhalb des kritischen Abschnitts aktiv die Kontrolle über die CPU ab (mittels `yield()`). Erläutern Sie, welchen Einfluss diese Entscheidung auf den weiteren Ablauf hat.

Allgemeine Hinweise

- Entpacken Sie die Vorgabe für Aufgabe 4 im Unterverzeichnis `aufgabe4` Erstellen sie im Unterverzeichnis `tests` für jede der Aufgaben eine Datei `aufgabeX.cc` X=1,2,3, in der Sie die einzelnen Aufgaben implementieren.
- Beantworten Sie die Zusatzfragen innerhalb der Dateien `aufgabeX.cpp`. Geben Sie jeweils deutlich an, auf welche Zusatzaufgabe sich Ihre Erläuterungen beziehen.
- Für etwaige Zeitmessungen verwenden Sie die Klasse `Timer` (siehe EZ-Stubs Einführung, Folien 31ff).
- In manchen Fällen sind Zufallszahlen hilfreich - in `debug/random.h` findet ihr die Schnittstelle eines Generators für Pseudozufallszahlen (Mersenne-Twister). **Wichtig**: Vor der Verwendung mit `random_init()` initialisieren.