
R CS

RCS	Lehrstuhl für Realzeit-Computersysteme

- Was sind RTOS und was macht sie aus?
- Warum verwendet man sie anstatt herkömmlicher Betriebssysteme oder dedizierter Hardware?
- Verschiedene Arten von RTOS und ihre Vor-/Nachteile (Wofür eignen sich bestimmte Systeme besonders gut, grober Überblick über vorhandenes)
- Beschreiben der Zielhardware/Randbedingungen
- Auskristallisieren, warum in der Arbeit gerade Linux RT Patch und FreeRTOS verwendet werden (evtl noch andere, z.B. MicroCOS, Xenomai)

Vergleichskriterien für Betriebssysteme

- Performanz
- Sicherheit
- Ressourcen-/Speicherverbrauch
- Speicherverwaltung

1. Latenzzeiten/Jitter
    a) Interrupt durch Taster → Aufblinken von LED (die Zeit, die dasAufblinken benötigt, kann gemessen und abgezogen werden, so dass nur die Zeit vom Drücken des Tasters bis zum Ausführen der ISR bleibt)
    b) Andere Interruptquellen? (→ z.B CAN, Ethernet, SPI, ...)
    c) Verschiedene Taktzeiten von FreeRTOS
2. Durchsatz an Daten
    a) Ethernet
    b) CAN
    c) SPI
3. Bootzeit
    a) Was hat Einwirkungen auf die Bootzeit?
    b) Indikatoren → Wann ist das System hochgefahren?
    c) Bestimmtes Programm wird gestartet → z.B. Aufleuchten von LED
    d) Bestimmte Programme können den Bootvorgang aufzeichen (Bootchart)
4. Scheduling/Context-Switching
    a) Hardware-Timer → genauer als Software-Timer

b) In FreeRTOS: Axi-Timer, der im PL instanziert wird

c) In Linux: clock_gettime() mit CLOCK_MONOTONIC (braucht keinen Treiber für Axi-Timer und Zugriffszeit auf Hardware fällt weg)

d) Beispiel: Start Task1 and Timer → Sleep Task1 → WakeUp Task2 → Timer stop

e) SPI

Für die Latenzzeit von Interrupts soll ein Interrupt von einer externen Quelle ausgelöst werden und dann wird gemessen, wann die Interruptserviceroutine betreten wird. Konkret wird periodisch ein GPIO-Interrupt durch einen Signalgenerator in Hardware ausgelöst. Die GPIO wird über das EMIO-Interface angebunden. In der dazugehörigen ISR wird eine LED angeschaltet. Es wird die Zeit zwischen dem Setzen des Interruptsignals und aufblinken der LED gemessen. Von dieser Zeit muss abgezogen werden, wie lange das Anschalten der LED und die Zeitmessung an sich dauert. Grundsätzlich wird jede Messung 1024 Mal durchgeführt. Für die Zusatzmessungen sind die Durchschnittszeiten interessant. Für die Hauptmessung ist zusätzlich der Worst-Case-Fall zu beachten.

Bei FreeRTOS werden die Interrupts unabhängig vom Betriebssystem verwaltet. Der EMIO-GPIO-Interrupt hat nach dem System-Timer die höchste Priorität.

Ergebnis:

Anz. Messungen	19100
Durchschnittswert	756,87ns
Standardabweichung	8,5977ns
Minimalwert	738ns
Maximalwert	768ns
Durchschnittswert LED-Anschalten	740ns
Durchschnittswert Timer-Overhead	520ns
LED-Overhead bei ISR-Messung	220ns
Durchschnitt - Overhead	536,87ns
Durchschnitt - Overhead (Taktzyklen)	806 Zyklen

Ergebnis:

Anz. Messungen	19160
Durchschnittswert	13,569us
Standardabweichung	1,4412us
Minimalwert	8us
Maximalwert	28,8us
Durchschnittswert LED-Anschalten	
Durchschnittswert Timer-Overhead	
LED-Overhead bei ISR-Messung	
Durchschnitt - Overhead	s
Durchschnitt - Overhead (Taktzyklen)	

1. Welche Unterschiede/Gemeinsamkeiten gibt es zwischen FreeRTOS und Linux?
2. Prioritäten
3. Semaphore
   a) Task1 setzt Timer und nimmt Semaphor → Schlafen → Task2 lässt Semaphor wieder los → Timer stop
   b) Task 1 setzt Timer und nimmt Semaphor → Task1 lässt Semaphor wieder los und Timer wird beendet
   c) Messen der Zeit, die benötigt wird, wenn ein Semaphor belegt ist und ein Task versucht, darauf zuzugreifen?
4. Message Passing (s. Semaphore)
5. Queues (s. Semaphore)
6. Flags (s. Semaphore)
7. Posix-Features in Linux

1. Speicherplatzverbrauch des gesamten Systems
2. MPU-Unterstützung
3. In welchem Rahmen sind dynamische Speicherzugriffe möglich?
4. Ggf. Zeitverbrauch bei Speicherallokation/-fragmentierung
   a) Allokation von z.B. 1000 Paketen und Messen der Zeit
   b) Vergleich von Context Switch mit Speicher Allokation und ohne (?)

c) Vergleich von verschiedenen Methoden der Speicherallokation → Was ist der Worst Case, der passieren kann?

Es gibt immer eine Mindestfrakturgröße. Außerdem sind die Funktionen Thread-Save, d.h. können durch keinen anderen Task unterbrochen werden. Ausnahmen davon bildet je nach Implementierung Fall 3.

Blöcke werden allokiert, wenn genug Speicher da ist und nie wieder freigegeben.

Zugriffszeit	Konstant
Worst Case	Nicht mehr genügend Speicher vorhanden
Schlussfolgerung	Schnell, aber vorher überlegen, ob der Speicher für die Lebensdauer der Anwendung reicht.
Testfall	Einfaches Allozieren, da kein Rechenaufwand durch Freigaben notwendig.

Es gibt eine minimale Blockgröße. Es gibt eine Liste, in der die Blöcke nach Größe sortiert sind. Es wird immer der nächst größte Block alloziert →Iteration durch Liste. Kein Verschmelzen von Blocks bei Freigabe. Zu große Blocks werden aufgeteilt. Der neu entstandene Block wird wieder in die Liste einsortiert. Nur sinnvoll, wenn der allozierte Speicher immer in etwa die gleiche Größe hat.

Maskierte malloc und free Aufrufe des jeweiligen Compilers.

Liste mit Blockzeigern und Blockgröße. Liste wird durchsucht, bis ein passendes Element gefunden wird. Bei Freigabe werden nebeneinander liegende Blöcke wieder zusammengeführt.

Zugriffszeit	Am Anfang konstant, weil nur ein Block. Sobald die Liste mehrere Elemente besitzt, ist die Zugriffszeit linear abhängig von der Länge der Liste.
Worst Case	Nicht mehr genügend Speicher vorhanden oder es ist Speicher vorhanden, aber nicht mehr an einem Stück oder es gibt sehr viele kleine Segmente in der Liste und nur ein größeres ganz hinten
Schlussfolgerung	Durch Freigaben langsamer als in Fall eins. Nicht sinnvoll, wenn allozierte Blockgröße variiert.
Testfall	Allozieren von möglichst vielen minimal großen Blöcken und einem, der die doppelte Größe hat. Alle wieder freigeben → Lange Liste mit vielen Einträgen → Nochmal den größeren Block allozieren. Die Zeit für die Längste Freigabe kann auch gemessen werden.

Zugriffszeit	
Worst Case	
Schlussfolgerung	
Testfall	

- Unter welchen Voraussetzungen ist eine Verifizierung möglich?
- Verifizierung bei einem ganz bestimmten Szenario

Zugriffszeit	Wie in Fall zwei, aber insgesamt schneller, da Blöcke bei der Freigabe wieder zusammengeführt werden und insgesamt tendenziell weniger Blöcke durchiteriert werden müssen.
Worst Case	Wie in Fall zwei
Schlussfolgerung	Flexibelste Alternative, Freigabe ist geringfügig langsamer als in Fall zwei, weil Blöcke noch zusammengeführt werden.
Testfall	Allozieren wie in Fall zwei. Freigabe von jedem zweiten Block, sodass Speicher segmentiert bleibt. Dann nochmal den hintersten Block allozieren.

- Verschiedene Auslastung (IO, Tasks, ohne Belastung, Speicherzugriffe) mit unterschiedlichen Ausführungsperioden
- Wie wirkt sich RT-Patch aus im Vergleich zum normalen Linux?
- Ggf. vorhandene Messinstrumente von Linux nutzen (?)
- Wie kann man die Performanz v.a. von Linux verbessern? → Abspecken des Systems

- Alternative Betriebssysteme: microCOS, microLinux, Xenomai
- Multiprozessoren
- Verbesserungen fur RT-Linux?
- Allgemeine, anerkannte Benchmarking-Methoden (Algorithmen): Rhealstone ist für Realzeit-Betriebssysteme geeignet (Whetstone und Dhrystone eher für verschiedene Hardwarearchitekturen)
- Messen von Bootzeiten mit bootchart
- Linux schneller machen durch Laden der Treiber zur Laufzeit?