INSTITUTE FOR REAL-TIME COMPUTER SYSTEMS
TECHNISCHE UNIVERSITÄT MÜNCHEN
PROFESSOR SAMARJIT CHAKRABORTY

Das ist ein etwas längerer Titel dieses leeren Diplomarbeitsgerüsts

Nadja Peters

Master's Thesis

Das ist ein etwas längerer Titel dieses leeren Diplomarbeitsgerüsts

Master's Thesis

Supervised by the Institute for Real-Time Computer Systems
Technische Universität München
Prof. Dr. sc. Samarjit Chakraborty

Executed bei Siemens

Advisor: Dipl.–Ing. Philipp Kindt

Author: Nadja Peters
Arcisstraße 21
80333 München

Submitted in June 2013

Acknowledgements

Vielen Dank . . .

München, im Monat Jahr

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

RCS Realzeit-Computer-Systeme

RTOS Realtime Operation System (Realzeitbetriebssystem)

POSIX Portable Operating System Interface

ISR Interrupt Service Routine

IRQ Interrupt Service Request

SMP Symmetric Multiprocessor

API Application Programming Interface

I/O Input/Output

PID Process Identifier

FIFO First-In-First-Out

OS Operation System

QOS Quality of Service

CFS Completely fair scheduler

Zusammenfassung

Die Kurzfassung ...

«««< HEAD ======= »»»> a4fadd5671de61751ce4fe354d86c6eca5cdf7cd

1 Einleitung

1.1 Realzeitbetriebssysteme

Was sind RTOS und was macht sie aus?

Im Allgemeinen ist ein Betriebssystem dafür verantwortlich, die Hardwareressourcen von einem System zu verwalten sowie für die Verwaltung von Benutzer- und Systemprozessen. Es gibt verschiedene Arten von Betriebssystemen. Im Alltag werden so genannte General Purpose OS verwendet, zum Beispiel Microsoft Windows oder MAC OS. Diese versuchen, die Ressourcen möglichst fair einzusetzen und die Antwortzeiten gegenüber dem Benutzer so gering wie möglich zu halten. In einem Realzeitbetriebssystem liegt der Fokus auf dem Garantieren von zeitlicher Determinierung von kritischen Anwendungen. Bestimmte Prozesse müssen in einer maximal festgelegten Zeit abgeschlossen sein, da das System sonst instabil werden könnte. Das ganze kann man an einem Beispiel greifbar machen:
Bei einem General Purpose OS kann es mehrere Sekunden dauern, bis sich der Internet Browser geöffnet hat. Würde es beispielsweise in einem Auto mehrere Sekunden dauern, bis sich der Airbag geöffnet hat, kann ein Mensch bereits verletzt worden sein. Dieses ist eine zeitkritische Anwendungen, die eine deterministische Antwortzeit garantieren muss. Bei Echtzeit unterscheidet man zwischen harter und weicher Echtzeit. Wenn bei harter Echtzeit die Deadlines verletzt werden, dann führt das zu einem Versagen des Systems und hat Folgen wie das Verletzten von Menschenleben. Bei weicher Echtzeit haben Verletzungen der Deadlines gegebenfalls einen Einfluss auf die QOS, führen aber üblicherweise nicht zum Absturz des Systems oder bergen Gefahr für beteiligte Personen.

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

Warum FreeRTOS und Linux

Auskristallisieren, warum in der Arbeit gerade Linux RT Patch und FreeRTOS verwendet werden (evtl noch andere, z.B. MicroCOS, Xenomai)

1.2 Welche Eigenschaften werden verglichen?

1.3 Relevante Betriebssysteme

1.3.1 Linux

Prozesse

Prozesse in Linux sind als doppelt verkettet Liste implementiert. Die Prozesse im State *Running* haben eine Liste pro Priorität. Prozesse in States wie *Task_Stopped*, *Exit_Zombie* oder *Exit_Dead* werden nicht in speziellen Listen gespeichert, weil der Zugriff meistens über PID oder Kind Prozesse stattfindet. Tasks, die interruptable oder nicht interruptable sind, müssen feiner gruppiert werden, um den Anforderungen der entsprechenden Funktionalität zu genügen. Zum Beispiel werden wartende (schlafende) Prozesse in einer Waiting Queue eingereiht. Je nach dem, ob ein Ereignis eintrifft, oder eine Ressource frei wird, werden entsprechende Prozesse wieder aufgeweckt. Context Switches werden nur im Kernel Modus vollzogen. Der Switch wird über Software gesteuert, nicht über Hardware.

Scheduling Policy

The scheduler stores the records about the planned tasks in a red-black tree, using the spent processor time as a key. This allows it to pick efficiently the process that has used the least amount of time (it is stored in the leftmost node of the tree). The entry of the picked process is then removed from the tree, the spent execution time is updated and the entry is then returned to the tree where it normally takes some other location. The new leftmost node is then picked from the tree, repeating the iteration.

If the task spends a lot of its time sleeping, then its spent time value is low and it automatically gets the priority boost when it finally needs it. Hence such tasks do not get less processor time than the tasks that are constantly running.

CFS is an implementation of a well-studied, classic scheduling algorithm called weighted fair queuing. Summing up, CFS works like this: it runs a task a bit, and when the task schedules (or a scheduler tick happens) the task's CPU usage is äccounted for": the

(small) time it just spent using the physical CPU is added to p->se.vruntime. Once p->se.vruntime gets high enough so that another task becomes the "leftmost tasköf the time-ordered rbtree it maintains (plus a small amount of "granularity"distance relative to the leftmost task so that we do not over-schedule tasks and trash the cache), then the new leftmost task is picked and the current task is preempted.

CFS doesn't use priorities directly but instead uses them as a decay factor for the time a task is permitted to execute. Lower-priority tasks have higher factors of decay, where higher-priority tasks have lower factors of delay. This means that the time a task is permitted to execute dissipates more quickly for a lower-priority task than for a higher-priority task. That's an elegant solution to avoid maintaining run queues per priority.

Interrupts

Regardless of the kind of circuit that caused the interrupt, all I/O interrupt handlers perform the same four basic actions:

1. Save the IRQ value and the register's contents on the Kernel Mode stack.
2. Send an acknowledgment to the PIC that is servicing the IRQ line, thus allowing it to issue further interrupts.
3. Execute the interrupt service routines (ISRs) associated with all the devices that share the IRQ.
4. Terminate by jumping to the *ret_from_intr* address.

1.3.2 RT Linux

Um Linux RT fähiger zu machen, werden IRQs anders behandelt als im normalen Linux. Im normalen Linux unterbrechen sie einfach den laufenden Betrieb des Betriebssystems. Mit dem RT-Patch werden die ISRs in eigene Prozesse mit Prioritäten umgewandelt, sodass RT-Prozesse eine höhere Priorität haben können als IRQs und damit weniger anfällig für Unterbrechungen sind. IRQs werden im RT-Patch mit der FIFO_SCHED Policy geschedulet und haben eine Priorität von 50.

In PREEMPT_RT, normal spinlocks (spinlock_t and rwlock_t) are preemptible, as are RCU read-side critical sections. Es gibt aber eine zusätzliche Art von Spin Locks, die das traditionelle Verhalten erlaubt.

1.3.3 FreeRTOS

- Tasks können die gleiche Priorität haben

- Tick rate bestimmt Zeitauflösung → je öfter es aufgerufen wird, desto mehr Zeit wird für das Betriebssystem aufgewendet (Task switches werden dann ausgeführt)
- Normalerweise sollten ISR so kurz wie möglich sein. Deswegen wird ein hoch priorisierter Task aus der ISR aufgerufen, der eine Priorität größer oder gleich dem System-Interrupt hat und wird dadurch nicht durch das System unterbrochen
- A mutex, recursive mutex, binary semaphore and a counting semaphore is using the existing queue mechanism.
- Mutexes include priority inheritance mechanism, binary semaphores do not. This makes binary semaphores the better choice for implementing synchronisation (between tasks or between tasks and an interrupt), and mutexes the better choice for implementing simple mutual exclusion.
- Rekursiver Mutex: Ein Mutex kann mehrfach gelockt werden, muss aber auch mehrfach wieder entlockt werden.

2 Benchmarking

Vergleichskriterien für Betriebssysteme

- Performanz
- Sicherheit
- Ressourcen-/Speicherverbrauch
- Speicherverwaltung

2.1 Performanz

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)

2.1.1 Latenzzeiten von Interrupts

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.

FreeRTOS

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

Axi-Timer: 50 MHz

Ergebnis:

Anz. Messungen	19100
Durchschnittswert	754,8ns
Standardabweichung	8,5977ns (19.31 ns)
Minimalwert	740ns
Maximalwert	780ns
Durchschnittswert LED-Anschalten	700-740ns (130 ns)
Durchschnittswert Timer-Overhead	520ns (9ns)
LED-Overhead bei ISR-Messung	220ns (121 ns)
Durchschnitt - Overhead	633,8ns
Durchschnitt - Overhead (Taktzyklen)	422 Zyklen

mit Task:

LinuxRT

Messung mit wmb(): Ergebnis:

Messung ohne wmb(): Ergebnis:

Messung For-Loops in Zyklen:

Messung While-Loops in Zyklen: Overhead Timer-Messung: $6 Zyklen$

Anz. Messungen	19100
Durchschnittswert	754,8ns
Standardabweichung	8,5977ns (19.31 ns)
Minimalwert	719ns
Maximalwert	840ns
Durchschnittswert LED-Anschalten	700-740ns (130 ns)
Durchschnittswert Timer-Overhead	520ns (9ns)
LED-Overhead bei ISR-Messung	220ns (121 ns)
Durchschnitt - Overhead	633,8ns
Durchschnitt - Overhead (Taktzyklen)	422 Zyklen

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)	

Overhead Registerzuweisung: $19 - 6 = 13 Zyklen = 19 ns$
Messung der Registerzuweisung mit wmb()[1)]

[1)] wmb(These functions insert hardware memory barriers in the compiled instruction flow; their actual instantiation is platform dependent. An rmb (read memory barrier) guarantees that any reads appearing before the barrier are completed prior to the execution of any subsequent read. wmb guarantees ordering in write operations, and the mbinstruction guarantees both. Each of these functions is a superset of barrier). Das bedeutet, dass die Schreiboperationen auf die Hardware bis zu dieser Barriere abgeschlossen sein müssen und man davon ausgehen kann, dass der Hardwarezugriff bereits erfolgt ist. Somit kann man messen, wie lange ein Hardwarezugriff zum Beschreiben einer LED dauert und diese Zeit von der Gesamtzeit abziehen. Um ein sinnvolles Ergebnis zu erhalten, sollte auch die Messung

Anz. Messungen	19130
Durchschnittswert	13,406us
Standardabweichung	1,3419us
Minimalwert	8,6us
Maximalwert	32us
Durchschnittswert LED-Anschalten	
Durchschnittswert Timer-Overhead	
LED-Overhead bei ISR-Messung	
Durchschnitt - Overhead	s
Durchschnitt - Overhead (Taktzyklen)	

For-Loop 0	Runden 16	180
For-Loop 1	Runden 32	304
For-Loop 2	Runden 64	533
For-Loop 3	Runden 128	1045
For-Loop 4	Runden 264	2133
For-Loop 5	Runden 512	4117
For-Loop 6	Runden 1024	8213
For-Loop 7	Runden 2048	16405
For-Loop 8	Runden 4096	32789
For-Loop 9	Runden 8192	65557

Loop 0	Runden 16	171
Loop 1	Runden 32	301
Loop 2	Runden 64	535
Loop 3	Runden 128	1047
Loop 4	Runden 264	2135
Loop 5	Runden 512	4119
Loop 6	Runden 1024	8215
Loop 7	Runden 2048	16407
Loop 8	Runden 4096	32791
Loop 9	Runden 8192	65559

2.1.2 Unterbrechung von Task durch ISR

Ein Task läuft und speichert in einer While-Schleife immer die aktuelle Zeit. Er wird durch eine ISR unterbrochen. Sobald der Task wieder anläuft, wird wieder die Zeit gemessen. Die gesuchte Zeit ist die Different aus Start- und Endzeit.

Die übliche Methode für Interrupt Service Routinen in FreeRTOS ist, einen hochpriorisierten Task laufen zu lassen, der an einem binären Semaphor blockiert. Wird die ISR aufgerufen, wird dieser Task wieder deblockiert.

2.2 RT-Features

1. Welche Unterschiede/Gemeinsamkeiten gibt es zwischen FreeRTOS und Linux?
2. Prioritäten
3. Flags (s. Semaphore)
4. Posix-Features in Linux

mit wmb() arbeiten.

2.2.1 RT-Features von Linux/Posix

Für Linux gibt es mehrere IPC Möglichkeiten: Die POSIX IPC und die System V IPC. Nach Möglichkeit wird die POSIX IPC benutzt, aber da nicht alle POSIX features auf dem System unterstützt werden, muss dafür auch auf die System V IPC zurückgegriffen werden.

Threads

Threads in Unix sind Teile von Prozessen. Allerdings hat ein Thread einen eigenen Stack Pointer, eigene Register, Scheduling Properties, Signale und andere Daten. Ein Thread existiert, solange der Elternprozess existiert. Ein Prozess kann mehrere Threads haben. Es kann Datenaustausch von Threads im Rahmen eines Prozesses geben. Ein Thread verbraucht deutlich weniger Ressourcen als ein Prozess. Inter-Thread-Kommunikation ist deutlich schneller, weil alles in einem Adressraum stattfindet. Bei Inter-Prozess-Kommunikation ist mindestens ein Kopiervorgang von Prozess zu Prozess erforderlich. Um pthreads benutzen zu können, muss die pthread-Bibliothek eingebunden werden (über -lpthread) und um zum Beispiel Prioritätsvererbung für Mutexe zu verwenden, muss zusätzlich mit dem Compilerflag *-D_GNU_SOURCE=1* kompiliert werden.

Mutexes

An Mutexen kann geblockt werden, aber es kann auch ausprobiert werden, ob sie bereits gelockt sind, und dann kann was anderes gemacht werden.

Conditions

Conditions werden im Zusammenhang mit Mutexen benutzt und dienen zur Synchronisation von mehreren Threads, die Datenabhängig sind. Zur Benutzung: An einer Condition kann gewartet werden. Zuvor muss ein bestimmter Mutex genommen worden sein. Wenn man den Befehl *pthread_cond_wait* ausführt, dann wird damit gewartet und der Mutex automatisch losgelassen. Ein anderer Thread kann sich denselben Mutex holen und dann eine bestimmte Datenverarbeitung an einer Variable durchführen. Wenn dadurch die Bedingung erfüllt wird, ruft dieser Thread die Funktion *pthread_cond_signal*, die den anderen Thread aufweckt, sobald der Mutex losgelassen wurde.

Join

Threads können *gejoint* werden. Dieses ist ein Synchronisationsmechanismus von PThreads. Wird *pthread_join* aufgerufen, blockiert der aktuelle Thread, bis der zu synchronisierende Thread beendet ist.

Message queues

mqd_t. Eine Queue muss erst erstellt werden. *mq_send* und *mq_receive* können blockend und nicht blockend aufgerufen werden, indem das Flag *O_NONBLOCK* gesetzt wird. Tasks können außerdem darüber informiert werden, dass eine Message in der Queue abgelegt wurde. Dieses funktioniert über *mq_notify*. Darüber kann ein Handler installiert werden, der ausgeführt wird, wenn eine neue Nachricht empfangen wird. Wenn ein anderer Task mit *mq_receive* an der Queue blockiert, dann wird kein Signal verschickt und der Handler nicht ausgelöst. The library *rt* muss beim Kompilieren gelinkt werden.

Scheduling

schedPxLib sched_getScheduler, gibt entweder SCHED_FIFO oder SCHED_RR zurück.

- *sched_get_priority_max*
- *sched_set_priority_max*
- *sched_rr_get_interval*
- *sched_yield*
- *sched_rr_get_interval*
- *pthread_setschedprio*
- *pthread_kill*

Semaphores

semPxLib. Posix Semaphores sind zählende Semaphore. Die unterstützen Funktionen sind Prioritätsvererbung, rekurive Semaphore, Timeouts, ... Posix Mutexes und Condition variables wurden implementiert, indem standardmäßige Semaphore verwendet wurden.

2.2.2 RT-Features von FreeRTOS

Relevante Features:

Tasks

Tasks unter FreeRTOS können mit verschiedenen Prioritäten erstellt werden. Der idleTask hat immer die niedrigste Priorität. Tasks haben verschiedene Zustände:

- Running: Task wird ausgeführt. Es kann zur Zeit nur einen einzigen Task geben, der gerade ausgeführt wird.

- Ready: Wartet darauf, ausgeführt zu werden, da ein anderer Task gerade vom Scheduler gescheduled wurde
- Blocked: Task wartet auf ein Ereignis und wird nicht ausgeführt. Grund kann ein Delay oder das Warten an einer Queue oder einem Semaphor sein.
- Suspended: Ein Task wurde von einem anderen Task oder sich selber suspendiert. Dieser Status kann nur durch einen Aufruf der Funktion *xTaskResume* fortgesetzt werden.

Scheduling Policy

Die Policy wird *Fixed Priority Preemptive Scheduling* genannt. Jedem Task wird eine eigene Priorität zugewiesen, wobei Tasks auch die gleiche Priorität haben können. Für jede Priorität existiert eine eigene Liste.

Queues

Queues werden benutzt, um Nachrichten zwischen Tasks auszutauschen. Von einer Queue kann zerstörend oder nicht zerstörend gelesen werden und drauf geschrieben werden. Wenn ein Task an einer Queue wartet, kann er für eine bestimmte Zeit blockiert werde. Eine Queue kann mit unterschiedlichen Größen erzeugt werden. Es gibt auch sogenannte Queue-Sets, die es ermöglichen an mehreren Queue oder auch Semaphoren zu warten.

Semaphore

Es gibt drei verschiedene Arten von Semaphoren (s. **??**), Mutexe, binäre Semaphore und Counting Semaphors. Semaphore werden als Queue implementiert. Werden Semaphore als Mutexe verwendet, ist die Prioritätsvererbung ebenfalls verfügbar. Ein Task kann immer nur an einem Mutex warten, da die Implementierung der Mutexe relativ simpel ist. Wenn ein Mutex von einem niederprioren Task losgelassen wird, kriegt er automatisch seine ursprüngliche Priorität zurück. Die zur Prioritätsvererbung gehörigen Code-Teile werden über Präprozessormacros eingebunden. Sollten also keine Mutexe verwendet werden, sollte das Macro auf jeden Fall auf undefiniert bleiben.

2.2.3 Task Switching

Unter Task Switching versteht man die Zeit, die der Scheduler braucht, um von einem Task zu einem anderen zu wechseln. Dieser Wechsel wird nach der Scheduling-Strategie des Schedulers vollzogen, d.h. der Task wird nicht etwa durch einen Interrupt oder durch einen höher prioren Task unterbrochen.

FreeRTOS

Variante 1 Es werden zwei Tasks *Task1* und *Task2* erzeugt. Diese Tasks haben einen Workload, der darin besteht, in einer For-Schleife eine Variable hochzuzählen. Nach jedem Inkrementieren der Variable wird ein Context-Switch erzwungen (mit taskYIELD()). Wenn die Variable eine bestimmte Höhre erreicht hat, wird das Experiment beendet. Es wird dabei die Zeit gemessen, die zwischen dem Betreten des ersten Tasks und dem Verlassen des letzten Tasks vergeht. Ein Task-Switch trifft also zwei Mal so häufig auf, wie die Schleife durchgelaufen wird.

Von der gemessenen Zeit muss noch der eigentliche Workload abgezogen werden. Dafür werden vor dem Starten der Tasks zwei For-Schleifen durchlaufen mit der gleichen Anzahl an Durchgängen wie in den Tasks.

Variante 2 Es werden zwei Tasks erzeugt, in denen eine For-Schleife mit der Anzahl der Testdurchläufe ausgeführt wird. In der Schleife befindet sich in der Reihenfolge:

- Starte Messung
- Erzwinge Task-Switch mit taskYield()
- Stoppe Messung

Der Vorteil an dieser Methode ist, dass es keinen Overhead gibt. Diese Messung ist also genauer. Von dem Ergebnis muss noch die Messzeit von sechs Zyklen abgezogen werden. Zu beachten ist, dass beide Tasks damit beginnen, die Startzeit der Messung zu speichern. Das führt dazu, dass die erste Startzeit überschrieben wird. Die letzte Startzeit ist dafür ungültig und dard nicht in dem Endergebnis berücksichtigt werden.

Linux

Auch hier ist das Vorgehen wie in Variante 2 von FreeRTOS. Für das taskYield wird die in Linux äquivalente Funktion *sched_yield* benutzt. Die Randbedingungen für die Linux-Experimente (also nicht nur für dieses hier sondern auch für die folgenden Experimente):

Erzeugen der Tasks In Linux muss man beim Erzeugen der Task in folgender Hinsicht aufpassen:
Da Realzeit-Messungen gemacht werden sollen, wird für diese Thread-Erzeugung das Realzeit-Scheduling von Linux gebraucht. Die Prioritäten-Range liegt über den Prioritäten der normalen Prozesse und geht von 1 bis 99, wobei 99 die höchste Priorität ist. Diese Prioritäten können nur für eine Realzeit-Scheduling-Policy vergeben werden. Hier unterstützt Linux Round-Robin- und FIFO-Scheduling. Um eine faire CPU-Verteilung zu gewährleisten, wird in dem Experiment RR benutzt. Beim Starten des Programms wird

deshalb das Scheduling für den Elternprozess verändert und die Priorität auf die maximal mögliche Echtzeit-Priorität gesetzt. Dann werden die eigentlichen Tasks erzeugt, die das Experiment ausführen. Diese haben eine niedrigere Priorität, als der Eltern-Prozess, um ihn nicht vorzeitig zu verdrängen. Wenn beide Tasks erzeugt sind, wird die Priorität es Eltern-Prozesses unter die der Kinderprozesse gesetzt. Diese sind damit am höchsten priorisiert und werden nicht länger durch den Eltern-Prozess blockiert.

2.2.4 Preemption-Zeit

Die Preemption-Zeit ist die Zeit, die benötigt wird, um einen Task-Switch zu vollziehen, wenn ein niederpriorer Task durch einen Interrupt oder durch einen höher priorisierten Task oder einen Interrupt unterbrochen wird. Das bedeutet, der Scheduler wird außerhalb des regulären Tick-Interrupts aufgerufen.

FreeRTOS

Ein niederpriorer Task verrichtet Arbeit. Dieser Task wird nach einer bestimmten Zeit von einem höher priorisierten Task unterbrochen. Es gibt zwei Funktionen in FreeRTOS, um einen Delay herbeizuführen: *vTaskDelay* und *vTaskDelayUntil*. *vTaskDelay* wacht nach einer bestimmten Anzahl von Ticks auf. Um die Zeit zu messen, kann die Startzeit in einer Endlosschleife im arbeitenden Task dauerhaft ausgelesen werden. Wenn der Task unterbrochen wird, wurde die aktuellste Zeit vorher gespeichert. Sobald der höher priorisierte Task aufgewacht ist, wird wieder die Zeit gemessen. Das *vTaskDelayUntil* wird nur jeden Tick ausgeführt.

Eine andere Möglichkeit, die Preemption Zeit zu messen, ist, dass ein hochpriorer Task sich selbst verabschiedet und hinterher von einem niederprioren Task aufgeweckt wird. Dieses führt direkt zu einem Context-Switch.

Noch eine Möglichkeit ist es, einen hochpriorisierten Task 1 zu suspendieren und dann einen niederpriorisieren Task 2 laufen zu lassen. Während der Task 2 läuft, wird ein Interrupt ausgelöst, der Task 1 fortsetzt und somit einen Context-Switch erzeugt.

Linux

2.2.5 Semaphor Shuffle Time

Die Semaphor Shuffle Time ist die Zeit, die ein Task braucht, um an einem von einem anderen Task genommenem Semaphor aufzuwachen, wenn dieser wieder losgelassen wird.

FreeRTOS

FreeRTOS hat mehrere Semaphorarten:

- Mutex mit Priority inheritance
- Binäre Semaphore ohne Priority inheritance
- Semaphore, die hochgezählt werden

Mutexe und Counting Semaphores lösen beim freigeben des Semaphores einen *portYield* aus, wodurch es zu einer Verzögerung kommt.

Mutex Ein Mutex wird verwendet, um Mehrfachzugriffe auf Resourcen zu vermeiden. Mutexe können mit Priority Inheritance verwendet werden, dieser Versuch wird aber mit zwei Tasks gleicher Prioritäten durchgeführt. Wenn ein Task einen Mutex genommen hat, muss dieser ihn auch wieder freigeben. Wenn ein anderer Task an diesem Mutex wartet, wird dieser durch die Freigabe wieder aktiviert.

Der Versuch kann durchgeführt werden, indem ein Task einen Semaphor nimmt und danach ein Context Switch durchgeführt wird. Ein zweiter Task versucht ebenfalls den Semaphor zu nehmen, wird aber blockiert. Als Folge erfolgt wieder ein Task Switch zurück zum ersten Task. Dieser startet eine Zeitmessung, lässt den Semaphor wieder los und veranlasst einen Context Switch. Nun wacht Task 2 auf, da der Semaphor losgelassen wurde. Die Zeitmessung wird beendet. Der Task lässt den Semaphor wieder los und mit einem Context Switch beginnt ein weiterer Durchlauf des Versuchs.

Im Endergebnis muss berücksichtigt werden, dass ein Mal *xSemaphoreTake*, ein Mal *xSemaphoreGive* und ein Context Switch von der Gesamtzeit abgezogen werden.

Binäre Semaphore Binäre Semaphore in FreeRTOS werden ähnlich wie Signale verwendet und diesen eher der Synchronisation als dem Vermeiden von Mehrfachzugriffen.

In diesem Versuch wartet der erste Task an dem Semaphor. Ein zweiter Task wird gestartet, initiiert eine Zeitmessung, gibt den Semaphor frei und macht einen Context Switch. Durch das Freigeben ist nun der ander Task wieder aufgewacht und beendet die Zeitmessung.

Von dem Endergebnis muss ein Mal *xSemaphoreGive* abgezogen werden.

Counting Semaphores Counting Semaphores sind Semaphore, die ein bestimmtes Kontingent haben und die hochgezählt werden, wenn ein Semaphor freigelassen wird (also eine Ressource verfügbar ist) und wieder runtergezählt werden, wenn ein Semaphor genommen

wird (also eine Ressource belegt ist). Ist keine Ressource verfügbar, wird an dem Semaphor gewartet. Ein Task wird wieder aktiv, sobald eine Ressource verfügbar wird.

Diese Semaphore können ähnlich wie Mutexe vermessen werden. Da der Mutex zuerst genommen wird, muss der Semaphor mit seiner maximalen Anzahl (im Versuch 1) initialisiert werden. Der Rest des Versuchs ist analog zum Mutex-Versuch. Am Ende müssen auch die gleichen Werte abgezogen werden.

Linux

2.2.6 Deadlock breaking time

Diese Zeit ist die Zeit, die benötigt wird, um einen Deadlock durch Prioritätsinversion wieder aufzulösen. Gäbe es diese nicht, wäre folgende Situation ein Deadlock:

Task 1 hat die niedrigste Priorität und nimmt sich Mutex M. Bevor M freigegeben wird, wird Task 1 durch Task 2 mit einer höheren Priorität unterbrochen. Task 2 wiederum wird durch Task 3 unterbrochen, was die höchste Priorität hat. Task 3 greift nach Mutex M und wird blockiert. Ohne Prioritätsinversion würde Task 2 weiterlaufen und Task 1 für immer unterbrechen, sodass die für Task 3 benötigte Ressource nie freigegeben wird. Prioritätsinversion sorgt dafür, dass Task 1 vorübergehend die Priorität von Task 3 bekommt, nicht mehr von Task 2 blockiert wird, und somit den Mutex wieder freigeben kann. Somit kann der Task mit der höchsten Priorität, Task 3, seine Arbeit beenden.

Für die Messung ist es irrelevant, ob Task 1 zwischen der Prioritätsinversion und dem Zurückgeben des Mutex noch Arbeit verrichtet. Um die Messung so genau wie möglich zu halten, wird der Mutex direkt zurückgegeben.

FreeRTOS

Linux

2.2.7 Message Passing Latency

Die Message Passing Latency ist die Zeit, die ein Task braucht, um eine Message von einem anderen Task zu empfangen. In FreeRTOS werden diese Messages über Queues transportiert. Die Zeit messen kann mit folgendem Aufbau:

Task 1 wird gestartet und wartet an der Queue auf eine Nachricht. Dabei wird der Task blockiert. Dann startet Task 2 und schreibt eine Nachricht in die Queue. Dieses bewirkt, dass der erste Task deblockiert wird. Die Zeitmessung beginnt vor dem Versenden der ersten Nachricht und endet mit dem Empfang.

FreeRTOS

Linux

In Linux gibt es mehrere Möglichkeiten, um Queues zu erzeugen. Da die POSIX message queues nicht von dem System unterstützt werden, wird die die System V message queue API benutzt.

2.3 Speicherzugriffe

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?

2.3.1 FreeRTOS

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.

Heap_1.c

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.

Heap_2.c

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.

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.

Heap_3.c

Maskierte malloc und free Aufrufe des jeweiligen Compilers.

Zugriffszeit	
Worst Case	
Schlussfolgerung	
Testfall	

Heap_4.c

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

2.3.2 Verifizierung

- Unter welchen Voraussetzungen ist eine Verifizierung möglich?

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.

- Verifizierung bei einem ganz bestimmten Szenario

2.3.3 Multiprozessorunterstützung

3 Testszenarien

- 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

4 Auswertung der Ergebnisse

5 Zusammenfassung und Ausblick

6 Notizen

- 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?

Literaturverzeichnis