Hochschule RheinMain
Fachbereich Design Informatik Medien
Prof. Dr. Robert Kaiser

Echtzeitverarbeitung

SS 2022
LV 4511 / LV 8481

Übungsblatt 4
Laborversuch
Abgabe: 9. Woche (8.06.2022)

Aufgabe 4.1. (Einführung):

Diese Aufgabe basiert auf der Aufgabe des PRLT - Echtzeit - Praktikum des Institut für Automatisierungstechnik von der Technischen Universität Dresden [1].

Das Ziel ist es, den Jitter im Scheduling von Threads auf echtzeitfähigen und nicht-echtzeitfähigen Linux-Betriebssystemen zu messen und miteinander zu vergleichen. Des Weiteren soll untersucht werden, wie über die POSIX-RT-API das Scheduling und somit das Zeitverhalten beeinflusst werden kann. Für diese Aufgabe werden diesmal keine I/Os verwendet.

Das bereitgestellte System-Image des Raspberry Pi bootet im Standardfall den normalen Linux-Kernel. Es ist aber auch ein Linux-Kernel installiert, der den RT-Patch enthält und daher echtzeitfähig ist. Um diesen zu booten muss in der Datei `/boot/config.txt` die Zeile 3 einkommentiert und das System neu gestartet werden (versuchen Sie nicht das kernel7.img zu booten, das funktioniert nicht und sie müssen die config.txt von einem anderen Rechner zurücksetzen). Verwenden Sie den Befehl "`uname -a`", um herauszufinden, welcher Kernel gerade läuft (RT = "4.19.71-rt24+" und Nicht-RT = "4.19.127+").

Im Unterschied zu dem Aufgabenblatt des PRLT-Praktikums der TU Dresden, gibt es das Programm `rtnice` unter dem normalen Linux nicht. Hier muss über den Programmcode und die POSIX-RT-API das Scheduling-Verfahren ausgewählt werden.

Aufgabe 4.2. (Cyclictest Benchmark):

Das Programm `cyclictest` [3] von Thomas Gleixner ist ein häufig verwendetes Werkzeug, um die Echtzeiteigenschaften von Linux zu vermessen. Es misst die Latenz der Antworten von Threads/Prozessen auf Stimuli. Dies geschieht in Schleifen über längere Zeit, sodass Aussagen über minimale, maximale und durchschnittliche Latenzzeiten getroffen werden können. Die Hintergründe und die Funktionsweise sind sehr schön in [4] erklärt.

Wiederholen Sie die folgenden Versuche mehrmals. Ggf. lassen Sie auch das Programm länger laufen. Machen Sie jeweils mehrere Läufe ohne Hintergrundlast und mit Hintergrundlast.

Booten Sie den Nicht-RT-Kernel (4.19.127+) und führen Sie `cyclictest` mit den Parametern `-p 80 -t 5 -n -l 10000` aus. Notieren Sie das Ergebnis. Was bedeuten die Parameter? Wie ist das Ergebnis zu interpretieren? Wie stehen die Werte im Verhältnis zueinander?

Booten Sie den RT-Kernel (4.19.71-rt24+). Wiederholen Sie den Test mit `cyclictest`. Was sind die Unterschiede? Was hat das ggf. mit dem RT-Patch zu tun?

Der Quellcode von `cyclictest` kann von [3] heruntergeladen werden. Übersetzen Sie den Code auf einem normalen Rechner. Wiederholen Sie den Test. Verändern Sie den Parameter des t-Flags in Abhängigkeit von der Anzahl an Cores, die Ihr Rechner hat (coreCount * 5). Was sind die Unterschiede? Wie sehen die Relationen zu den Messungen auf dem Pi aus?

Aufgabe 4.3. (Programm für die Messung von Jitter im Scheduling von POSIX-Threads):

Auf der Lehrveranstaltungswebseite finden Sie einen Programm-Stub, den Sie für diese Aufgabe verwenden können. Das Programm startet zwei Threads, die periodisch eine Aufgabe ausführen. Die Dauer für die Durchführung der Aufgabe kann in Grenzen schwanken, ist aber immer kürzer als die Zykluszeit. Nach der Durchführung der Aufgabe legen sich die Threads schlafen, bis der nächste Zyklus beginnt. Die Abweichung (Jitter) zwischen dem berechneten Zeitpunkt für den Beginn eines Zyklus und dem tatsächlichen soll in dieser Aufgabe gemessen werden.

Der bereitgestellte Programm-Stub übergibt den beiden Threads beim Start eine Reihe von Parametern. Jeder Thread führt die Funktion `doWork()` aus, die aktiv eine Zeit lang die CPU beansprucht. Jedem Thread wird eine Instanz der `timerMessStruct` Struktur übergeben. In dieser sind die relevanten Parameter enthalten, bspw. die jeweilige Zykluszeit. Standardmäßig hat der erste Thread hat eine Zykluszeit von 2 ms und der zweite von 4 ms.

Für die Messung der Zeit soll die Funktion `gettimeofday()` verwendet werden. Hier wird die Struktur `struct timeval` benötigt, die Zeitpunkte in Mikrosekundenauflösung darstellen kann. Des Weiteren sind die Makros „timerisset“, „timeradd“, „timercmp“ und „timersub“ aus „sys/-time.h“ sehr hilfreich.

Die Thread-Funktion im Stub führt bisher nur die doWork-Funktion wiederholt innerhalb einer Schleife aus. Erweitern Sie die Thread-Funktion so, dass die Schleifendurchläufe immer dieselbe Zeit benötigen, also der Zyklus immer gleich lang dauert. Dafür nehmen Sie jeweils einen Zeitstempel zu Beginn des Zyklus, bevor die doWork-Funktion aufgerufen wird und danach. Berechnen Sie zuerst die reale Dauer der Arbeitsfunktion und dann, wie lange der Thread bis zum Beginn des nächsten Zyklus schlafen muss.

Als nächstes berechnen Sie den eigentlich gewünschten Zyklusbeginn. Nehmen Sie dazu beim ersten Aufruf der Thread-Funktion einen Zeitstempel und addieren Sie dazu in jedem Zyklus die Zyklusdauer. Diesen Zeitwert können Sie danach mit dem realen Beginn des Zyklus vergleichen bzw. die Differenz bilden. Dies ist die Abweichung, der Jitter. Speichern Sie diese in dem `responseTimeArray`.

Als nächstes müssen Sie einige Ausnahmefälle berücksichtigen. Es kann sein, dass ein Thread so spät gestartet wird, dass er das Ende seines Zyklus überschreitet (negatives Ergebnis der Schlafzeitberechnung). In diesem Fall soll er nicht schlafen gelegt werden, sondern direkt den nächsten Zyklus beginnen. Diese Überschreitungen sollen in der Variable `deadlineMissed` gezählt werden.

Testen Sie ihre Anwendung. Machen Sie mehrere Messreihen, mit Load im Hintergrund und ohne. Welche Werte bekommen Sie? Wiederholen Sie den Versuch mit dem Nicht-Echtzeit-Kernel. Gibt es Unterschiede?

Aufgabe 4.4. (POSIX-RT-Scheduling):

Mit der POSIX-RT-API gibt es die Möglichkeit das Scheduling-Verhalten von Programmen zu beeinflussen. Es gibt zwei Echtzeit-Scheduling-Verfahren mit Prioritäten zur Auswahl: FIFO und Round Robin.

Erweitern Sie Ihr Programm so, dass über die Programmparameter konfiguriert werden kann, ob und wenn ja welches Scheduling-Verfahren verwendet werden soll. Die notwendigen Aufrufe der POSIX-API erledigen Sie dazu am besten ganz zu Beginn der Threads, bevor die zyklische Abarbeitung beginnt. Schauen Sie sich die Beschreibung in [5] und [2] dazu an.

Wiederholen Sie Ihre Messungen mit verschiedenen Einstellungen, sowohl unter dem Echtzeitkernel als auch dem normalen. Was sind die Ergebnisse? Wie sind sie zu bewerten?

Sie können auch die Zyklus- und Arbeitszeiten verändern. Wie wirkt sich dies auf die Ergebnisse aus?

Aufgabe 4.5. (Vorbereitung nächste Woche):

(a) Bereiten Sie die Vorlesung über die "Regelungstechnik" aktiv nach. Recherchieren Sie über die PID-Regler und seine möglichen Implementierungen in C.

A. (*):

Literatur

[1] `https://www.cs.hs-rm.de/~kaiser/2121_ezv/EZVPraktikum.pdf`

[2] `http://www.lug-erding.de/vortrag/Linux_Echtzeit-02-Programmierung.pdf`

[3] `https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start`

[4] `https://events.static.linuxfound.org/sites/events/files/slides/cyclictest.pdf`

[5] `http://retis.sssup.it/~lipari/courses/str09/10.rtprogramming-handout.pdf`

B. (Raspberry Pi GPIO Pins):

Raspberry Pi – GPIO-connector

HAT	WiringPi	GPIO	Name	Header		Name	GPIO	WiringPi	HAT
			3.3V	1	2	5V			
FanSoftPWM	8	2	SDA	3	4	5V			
Fan Tacho	9	3	SCL	5	6	GND			
	7	4		7	8	TxD	14	15	
			GND	9	10	RxD	15	16	
	0	17		11	12		18	1	PWM/GPIO18
SW2	2	27		13	14	GND			
SW1	3	22		15	16		23	4	
			3.3V	17	18		24	5	
	12	10	MOSI	19	20	GND			
	13	9	MISO	21	22		25	6	
	14	11	SCLK	23	24	CE0	8	10	
			GND	25	26	CE1	7	11	
				27	28				
DRV_A_en	21	5		29	30				
DRV_A_in1		6		31	32		12	26	DRV_A_in2
DRV_B_in1	23	13		33	34				
DRV_B_in2	24	19	MISO	35	36				
DRV_B_en	25	26		37	38	MOSI	20	28	GPIO20
			GND	39	40	SCL	21	29	GPIO21