3. Aufgabenblatt mit Lösung zur Vorlesung

Betriebssysteme WiSe 2017

Sebastian Börner, Christian Hofmann, Simon Auch

Abgabe 7 Dezember 2017, 10 Uhr

Aufgabe 1 Nachrichten unterschiedlicher Länge *5 Punkte*

Erläutern Sie die Begriffe:

- memory mapped I/O,
- DMA (direct memory access) und
- polling.

Gegeben sei nun ein Prozessor mit einer 200MHz Taktfrequenz. Eine Polling-Operation benötige 400 Taktzyklen. Berechnen Sie die prozentuale CPU-Auslastung durch das Polling für die folgenden Geräte. Bei 2 und 3 ist dabei die Abfragerate so zu wählen, dass keine Daten verloren gehen.

1. Maus mit einer Abfragerate von 30/sec,
2. Diskettenlaufwerk: Datentransfer in 16-bit-Wörtern mit einer Datenrate von 50KB/sec,
3. Plattengerät, das die Daten in 32-bit-Wörtern mit einer Rate von 2MB/sec transportiert.

(Hinweis: Wie viele Taktzyklen gibt es pro Sekunde? Wie viele Taktzyklen werden für eine Abtastrate von 30/sec benötigt? Was für einer Abtastrate entsprechen 50KB/sec bei 16-bit- Wörtern ...)

Für das Plattengerät soll jetzt DMA eingesetzt werden. Wir nehmen an, dass für das Initialisieren des DMA 4000 Takte, für eine Interrupt-Behandlung 2000 Takte benötigt werden und dass per DMA 4KB transportiert werden. Das Plattengerät sei ununterbrochen aktiv. Zugriffskonflikte am Bus zwischen Prozessor und DMA-Steuereinheit werden ignoriert.

Wie hoch ist nun die prozentuale Belastung des Prozessors? (Hinweise: Wie viele DMA- Transfers pro Sekunde sind bei 2MB/sec und 4KB Transfergröße notwendig?)

Lösung:

memory mapped I/O Teil des Adressraums wird für Speicher und I/O verwendet; Belegt also Teil des verfügbaren Adressraums; I/O Geräte reagieren auf Zugriffen des Bereichs

DMA Erlaubt Zugriff auf den Speicher ohne I/O -Operationen via CPU nutzen zu müssen;

polling Das Abfragen eines Geräts (o.Ä.), ob entsprechendes bereit ist

(a)

(b)

(c)

Aufgabe 2 Kontextwechsel und präemptives Multitasking *15 Punkte*

In dieser Aufgabe soll Ihr Betriebssystem um die Fähigkeit des präemptiven Multitaskings erweitert werden, um so mehrere Threads quasi-parallel ausführen zu können. Der System-Timer dient dabei als Zeitgeber und gibt das Intervall vor, in dem der Kern den gerade ausgeführten Thread automatisch wechselt.

Lösung: Anleitung zum Erstellen eines mit qemu benutzbaren Kernels:

(a) Wechseln in den Ordner `GrandiOS`

(b) Zunächst wird mit dem Script `setup_env.sh` die Toolchain von Rust vorbereitet. Unter Umständen sind Benutzereingaben erwartet, diese können einfach mit Enter bestätigt werden.

(c) Ausführen von `make run` zum Erstellen eines Kernels und starten von Qemu mit diesem.

Das Vorbereiten der Toolchain mittels `setup_env.sh` ist nur einmal notwendig.

Für den unwahrscheinlichen Fall, dass aus irgendeinem Grund das Bauen des Kernels fehlschlägt, ist unter `GrandiOS/kernels/kernel_04` noch ein von uns gebauter Kernel vorhanden.

Eine kleine Übersicht, wo die für diese Aufgabe relevanten Codesegmente zu finden sind:

- `GrandiOS/src/utils/scheduler.rs` Implementation der Thread-Verwaltung. Unsere Zeitscheibenlänge beträgt standardmäßig 2 Sekunden, es sei denn ein Interrupt verkürzt diese. Insbesondere werden jedoch auch bei Benutzung eines Sleep-SWI die Zeitscheibe so eingestellt, dass der nächste Thread pünktlich gestartet werden kann. Falls es mehrere Threads der gleichen Priorität gibt, ergibt sich durch die Verwendung der Priority-Queue automatisch ein Round-Robin-Scheduling. Die maximale/minimale Zeitscheibenlänge kann hierbei zu Beginn/am Ende der Switch-Funktion eingestellt werden (in Ticks, wobei ein Tick als 1/1024 Sekunden eingestellt ist). Der Fall, dass es keinen Thread zum Ausführen gibt, kann nicht eintreten, da wir immer einen Idle-Thread haben, welcher immer ausgeführt werden kann (und nicht beendet werden kann). Der Idle-Thread pausiert insbesondere auch die CPU (siehe ganz unten in der Datei). Ein Thread kann mittels des Exit-Syscalls beendet werden, dies entfernt ihn aus allen Warteschlangen aus dem Scheduler und gibt später vielleicht irgendwann mal benutzte Ressourcen frei.
- `GrandiOS/src/utils/exceptions/common_code.rs` Beinhaltet den Trampolincode für unsere Interrupts (sowohl swi als auch irq). Hierbei werden alle

relevanten Register auf dem Interrupt-Stack gesichert. Falls der Thread gewechselt werden soll müssen diese nur woanders gespeichert werden und im Stack überschrieben werden.

- `GrandiOS/src/utils/exceptions/irq.rs` Beinhaltet die Implementierung des Threads, den wir starten, als auch die Ausgabe des Ausrufezeichens. Der Thread benutzt leider kein Sleep-Aufruf, da er hierfür in der aktuellen implementation einen Allocator benutzen muss, was jedoch im Kontext des Kernels (zu dem diese Datei gehört) leider der gleiche Allocator wie vom Scheduler ist und daher nicht ohne Kernellock benutzt werden sollte. Deswegen machen wir stattdessen einfach busy-waiting.