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

Beispiel: Das setzen eines Bits einer bestimmten Adresse sorgt dafür, dass in dieser beim nächsten Lesen ein Zeichen der Benutzereingabe steht.

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. Der Nachteil dieses Vorgehens besteht darin, dass relativ viel Rechenzeit dafür verwendet wird, diese Abfragen zu tätigen. Als gegenteiliges Vorgehen lässt man sich z.B. durch einen Callback über die Bereitschaft informieren.

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,

   30 Abfragen pro Sekunde mit je 400 Taktzyklen ergibt 12kHz. 12kHz/200000kHz = 0,000060 = 0,006% Auslastung.

2. Diskettenlaufwerk: Datentransfer in 16-bit-Wörtern mit einer Datenrate von 50KB/sec,

   50kB (Kilobyte, nicht Kelvin-Byte wie in der Aufgabenstellung) entsprechen 3125 16-Bit-Wörtern. Wir erhalten wieder 400*3125 = 1,25MHz; 1,25MHz/200MHz = 0,00625 = 0,625%.

3. Plattengerät, das die Daten in 32-bit-Wörtern mit einer Rate von 2MB/sec transportiert.

   2MB entsprechen 62500 32-Bit-Wörtern. 62500*400 = 25MHz entspricht 12,5%.

Bei 2MB/s sind 500 Transfers von je 4kB nötig. Wir erhalten 2000*500=1MHz, zuzüglich Initialisierung 1,0004MHz.

Analog zu den vorherigen Berechnungen erhalten wir 1,0004MHz/200MHz = .005002 = 0,5002%.

Aufgabe 2 User-/Kernel-Interface *15 Punkte*

In dieser Aufgabe soll Ihr Betriebssystem um eine einfache Koordination einerseits und andererseits um eine effizientere Ressourcen-Nutzung erweitert werden.

Definieren Sie hierzu eine ordentliche Schnittstelle zwischen Anwendungen und Betriebs- system und führen Sie blockierende Systemrufe ein, damit wartende Threads nicht unnötig die CPU blockieren.

1. Legen Sie eine Aufrufkonvention für Systemrufe mittels SWI fest.
2. Definieren Sie Systemrufe für die Aufgaben „Zeichen ausgeben“, „Zeichen einlesen“, „Thread beenden“, „Thread erzeugen“ und „Thread für bestimmte Zeitspanne ver- zögern“.
3. Implementieren Sie die Systemrufe auf Seite des Kernels. Dabei sollen „Zeichen einlesen“ und „Thread verzögern“ blockierend arbeiten. Das heißt, der Thread wird so lange nicht mehr durch den Scheduler erfasst, bis ein Zeichen da ist bzw. die vorgegebene Zeitspanne vorbei ist. Überlegen Sie sich, ob ein aufwachender oder neuer Thread dem gerade laufenden Thread vorgezogen werden sollte oder nicht.

4. Implementieren Sie eine „Bibliothek“, die für Anwendungen eine leichter zu benutzende Schnittstelle zur Verfügung stellt, als direkt SWIs aufzurufen. Idealerweise sind der Code für Anwendungen/Anwendungsbibliotheken und der Code für das Betriebs- system/Betriebssystembibliotheken zum Schluss vollständig unabhängig voneinander.

Durch die SWI-Schnittstelle wird zum einen eine logische Trennung von Anwendungen und Betriebssystem erzeugt. Zum anderen wird – da nur einen Kern vorhanden ist – automatisch ein gegenseitiger Ausschluss sämtlicher Kernel-Funktionen realisiert (sofern Sie in privilegierten Modus [Modi] die Interrupts nicht demaskiert, siehe Zusatzaufgabe). Die Demonstration ist ähnlich wie in der letzten Aufgabe zu implementieren, diesmal jedoch mit eigenem Thread:

5. Schreiben Sie eine Anwendung, die stets auf Zeichen wartet. Wann immer ein Zeichen empfangen wird, erzeugt diese Anwendung einen neuen (Anwendungs-)Thread und gibt diesem das empfangende Zeichen als Parameter mit.
6. Der neu erzeugte Thread soll das Zeichen wiederholt (aber nicht endlos) mit kleinen Pausen ausgeben und sich anschließend beenden. Handelt es sich bei dem Zeichen um einen Großbuchstaben wird zur Erzeugung der Pause aktiv gewartet (wie bisher), bei anderen Zeichen wird die Pause durch den neu eingeführten Systemruf erzeugt.

Bei den daraus resultierenden Ausgaben ist insbesondere das zeitliche Verhalten spannend. Die folgenden Beispiele gehen davon aus, dass die Rechenzeit beim aktiven Warten genauso lang ist, wie die Dauer der Verzögerung via Systemruf. (Und beides ist ein Vielfaches der Zeitscheibenlänge.)

2x aktives Warten: ABABABABABABABABABABAB

1x aktiv + 1x passiv: AbAbAbAbAbAbAbAbAbAbAb

2x passiv: abababababababababab

1x aktiv + 2x passiv: AbcAbcAbcAbcAbcAbcAbcAbcAbcAbcAbc

2x aktiv + 1x passiv: ABcAcBcAcBcAcBcAcBcAcBABABABAB

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.