3. Aufgabenblatt mit Lösung zur Vorlesung

Betriebssysteme WiSe 2017

Sebastian Börner, Christian Hofmann, Simon Auch

Abgabe 7 Dezember 2017, 10 Uhr

Aufgabe 1 Threads *6 Punkte*

Grenzen Sie die folgenden Begriffe auf Grund der Vorlesung gegeneinander ab:

- gegenseitiger Ausschluss
- Signalisierung
- Synchronisation
- Koordination
- Kommunikation
- Kooperation

Gehen Sie bei Ihrer Diskussion insbesondere auf die Teilnehmer und ihre Beziehung zueinander ein.

Lösung:

gegenseitiger Ausschluss Gegenseitiger Ausschluss dient dazu sicherzustellen, dass nicht zwei oder mehr Prozesse/Threads zeitgleich auf eine bestimmte Ressource zugreifen.

Signalisierung Signalisierung wird benutzt um zum Beispiel bei gegenseitigen Ausschluss anderen Prozessen/Threads, die auf eine vom aktuellen Prozess/-Thread beanspruchte Ressource warten, zu signalisieren, dass diese wieder zu Verfügung steht.

Synchronisation soll sich das hier nur auf threads beziehen, also Prozesssynchronisation

Koordination Bei Koordination wird entschieden, wer wann worauf zugreifen kann. Bei Threads mit geteilter Ressource können dabei u.A. mehrere Threads (Leser) für die Ressource so koodiniert werden, dass diese zeitgleich drauf zugreifen können, sofern kein Schreiber für die Ressource grade aktiv ist (also anders als beim gegenseitigen Ausschluss wo je nur einer zugelassen wird).

Kommunikation

Kooperation

Aufgabe 2 Interrupts - System Timer und serielle Schnittstelle *15 Punkte*

Mit dem Abfangen von Ausnahmen ist unser „Betriebssystem“ noch kein wirkliches Betriebs- system, sondern eher ein Bare-Metal-Programm mit ein wenig Programmier-Komfort. Um diesen Umstand auszuräumen, unterstützen wir in dieser Aufgabe auch Hardware-Interrupts. Hardware-Interrupts bieten später die einzige Möglichkeit für das Betriebssystem die Kontrolle über den Prozessor wiedererlangen zu können, wenn mal ein Prozess in einer Endlosschleife hängen sollte. Außerdem geben uns Hardware-Interrupts die Möglichkeit, zeitnah auf Hardware zu reagieren, auch wenn der Prozessor gerade eigentlich mit etwas anderem beschäftigt ist.

Speziell in dieser Aufgabe stellen wir das Lesen (und optional das Schreiben) von der seriellen Schnittstelle (bei uns DBGU) von Polling auf Interrupt-getriebene Behandlung um. Interrupt- getriebenes Lesen verringert die Chance, dass wir ein Zeichen verpassen, weil wir es nicht rechtzeitig abgeholt haben. (Interruptgetriebenes Schreiben würde zudem ein Fortsetzen der Programmausführung ermöglichen, ohne dass auf die Fertigstellung der Übertragung des vorherigen Zeichens gewartet werden muss.)

Damit sichergestellt ist, dass uns ein Hardware-Interrupt ereilt (auch wenn der Benutzer keine Taste drückt), setzen wir außerdem proaktiv den System Timer (ST) ein, um so periodisch Interrupts zu erzeugen und unserem Betriebssystem eine garantierte Eingriffsmöglichkeit zu geben.

Lösung:

folgendes ist copy-paste (von zettel2)

Anleitung zum erstellen eines mit qemu benutzbaren Kernels:

(a) Wechseln in den Ordner `GrandiOS`

(b) Einen von beiden Optionen ausführen:

- Ausführen von `make`. Falls schon eine Rust-Umgebung vorhanden ist, stattdessen `make build`.
- (i) 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.
  (ii) Durch starten von `kernel_build.sh` kann nun der Kernel `kernel` erzeugt werden.
  (iii) Starten von qemu mit dem Kernel: `qemu-bsprak -piotelnet -kernel kernel`
  (iv) Starten einer telnet Verbindung zu qemu: `telnet localhost 44444`

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

Für den unwahrscheinlichen Fall, dass aus irgend einem Grund das bauen des Kernels fehlschlägt, ist unter `GrandiOS/kernels/kernel_02` noch ein von uns gebauter Kernel vorhanden.

Für diese Aufgabe kann der Parameter `-piotelnet` in Schritt 4 sowie das Starten von telnet in Schritt 5 ausgelassen werden.

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

- Remap Befehl: `GrandiOS/src/lib.rs::init`
- Stack pointer konfigurieren: `GrandiOS/src/lib.rs::init`
- Aktivieren der IRQ und FIQ im Prozessor: `Grandios/src/lib.rs::init`
- Zugriff auf die IVT sowie den AIC: `GrandiOS/src/driver/interrupts.rs`
- Zum registrieren der einzelnen interrupt/exception Handler können nun die folgenden Befehle benutzt werden (der Code ist in src/commands/test.rs):
  - `test interrupts_undefined_instruction` hinterlegt einen Handler für undefined instruction Ausnahmen
  - `test interrupts_software_interrupt` hinterlegt einen Handler für software interrupt Ausnahmen
  - `test interrupts_prefetch_abort` hinterlegt einen Handler für prefetch abort Ausnahmen
  - `test interrupts_data_abort` hinterlegt einen Handler für data abort Ausnahmen

  - `test interrupts_aic` hinterlegt einen Handler für die IRQ Leitung und konfiguriert den AIC sowie die DebugUnit einen Interrupt für das empfangen eines Zeichens zu erzeugen. Diese Funktion geht in eine Endlosschleife, da durch den Interrupthandler zurzeit die Tastendrücke abgefangen werden und die `pfush` damit unbrauchbar wird.

- Zum Betrachten was in die IVT geschrieben wurde, kann der Befehl `edit 0x0 0x40` genutzt werden. Wie der name nahelegt können hiermit auch Daten geschrieben werden, verlassen geht mit der Tastenkombination `strg-D`
- Zum Auslösen der einzelnen interrupts/exceptions können die folgenden Befehle benutzt werden:
  - `test undefined_instruction`
  - `test software_interrupt`
  - `test prefetch_abort` (noch nicht funktionsfähig da nur mit MMU möglich)
  - `test data_abort`