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** Synchronisation ist ein Mechanismus, der sicherstellt, dass mehrere Threads sich an einer definierten Stelle in ihrem Ablauf befinden.

**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** Kummunikation zwischen Threads (in Bezug auf Betriebssysteme) findet häufig über geteilten Speicher statt. Damit beim geteilten Zugriff auf die Daten keine Fehler entstehen wird für gewöhnlich ein Verfahren für gegenseitigen Ausschluss verwendet, wie das Sperren des Zugriffs auf den gemeinsamen Speicher via eines Mutex, eines Semaphors oder eines Monitors. Ggf muss den anderen Threads signalisiert werden, wenn Daten geschrieben wurden.

**Kooperation** In Bezug auf das Scheduling, ist Kooperation in der Art zu finden, dass Threads sich untereinander Zeit für den Kernel zuteilen, bzw. an andere Threads abgeben, wenn sie eine gewisse (in der Regel selbst definierte) Zeit den Prozessor beansprucht haben.
In Bezug auf das Lösen von Aufgaben von Problemen, können die Threads kooperativ zusammenarbeiten indem mehrere Threads unterschiedliche Teilaufgaben eines Problems angehen und anschließend mittels Kommunikations- und Synchronisationsmechanismen, wie denen von den vorigen Punkte, die Ergebnisse zusammenführen.

**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:** 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_03` 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:

- `GrandiOS/src/driver/system_timer.rs`
- `GrandiOS/src/utils/exceptions/irq.rs`
- `Grandios/src/lib.rs::init`
- `GrandiOS/src/driver/interrupts.rs`
- Die Funktionalität ist in dem Programm u3 zu betrachten, welches von der Shell aus gestartet werden kann. Die Ausgabe der Ausrufezeichen erfolgt aus Bequemlichkeit auch außerhalb dessen. Der entsprechende Code hierfür liegt in `GrandiOS/shell/src/commands/u3.rs`.

  Hierbei ist zu bemerken, dass das read!-Makro nicht wie im zweiten Zettel aus dem Code der DebugUnit verwendet wird, sondern dass unsere STD-lib (`GrandiOS/aids/src/lib.rs`) einen Softwareinterrupt mittels eines für den Kernel und das Userland geteilten Interfaces (`GrandiOS/swi/src/lib.rs`) auslöst, der dann das von dem Interrupt der DebugUnit gelieferte Zeichen erhält und wieder aufgeweckt wird.