2. Aufgabenblatt mit Lösung zur Vorlesung

Betriebssysteme

WiSe 2017

Sebastian Börner, Christian Hofmann, Simon Auch

Abgabe 23 November 2017, 10 Uhr

Aufgabe 1 Threads

6 Punkte

Wann ist es sinnvoll, nebenläufige Programmteile mit Hilfe von Threads anstatt Prozessen (heavyweight processes) zu implementieren?

Welche Vorteile bieten User-Level-Threads gegenüber den Kernel-Level-Threads? Gibt es auch Nachteile?

Welche Vorteile und Nachteile gibt es, wenn man Thread-Kontrollblöcke (TCB) als Skalare, in Arrays, Listen, Bäumen oder invertierten Tabellen speichert?

In welchem Adressraum (Prozess-Eigner, Dienste-Prozess, BS-Kern) wird ein TCB ge- speichert?

Lösung:

(a) Der größte unterschied zwischen nebenläufigen Programmteilen realisiert durch Threads anstatt von Prozessen ist, dass Threads im gleichen logischen Speicher liegen. Dies bedeutet insbesondere das Threads unter Umständen schneller miteinander kommunizieren können, da diese dafür keine Systemaufrufe benötigen. Von dieser Eigenschaft profitieren natürlich insbesondere Programmteile die viel miteinander kommunizieren müssen. Dies ist jedoch eine Abwägung gegen die verlorene Sicherheit durch getrennte Speicherbereiche.

(b) Der Vorteil von User-Level-Threads besteht in Abgrenzung gegenüber anderen Threads und Prozessen und erhöht dadurch die Sicherheit. Da der Scheduler aber Informationen aus dem Kernel-Space benötigt, muss für jeden Threadwechsel auch ein Wechsel zwischen diesen Kontexten vollzogen werden.

(c)
- Array
  Vorteil: kein Overhead, konstante Zugriffszeit
  Nachteil: feste Größe, muss zur Größenänderung komplett kopiert werden
- Listen
  Vorteil: in konstanter Zeit beliebig erweiterbar
  Nachteil: Zugriff hat lineare Laufzeit
- Bäume (Annahme: Suchbäume)
  Vorteil: schnelles finden eines bestimmten TCB
  Nachteil: Overhead, Einfügen/Entfernen dauert logarithmisch lang
- Invertierte Tabellen (gehen von invertierten Seitentabellen aus (?)):
  Vorteil: Schneller Zugriff auch bei verschiedenen Sichtweisen auf die TCBs (z.B. anhand von Prozess-ID oder Priorität) ohne Mehrfachspeicherung für

die Ansischten
Nachteil: Mögliche Page Faults durch mapping von virtuellen auf physische Adressen

(d) TCBs werden im Adressraum des Betriebssystems gespeichert, da die Prozesse selbst nicht (direkt) auf den TCB zugreifen können sollten. Die TCBs in einem Dienste-Prozess zu verwalten geht auch nicht direkt, da zumindest der TCB dieses Prozesses im Betriebssystem gespeichert werden muss.

Aufgabe 2 Behandlung von Ausnahmen *15 Punkte*

Nach der ersten Kontaktaufnahme mit der Zielplattform soll nun der ARM-Kern vollständig initialisiert werden und Ihr Betriebssystem-Code erste Aufgaben übernehmen, die über Low- level-Anwendungsentwicklung hinausgehen. Konkret sollen Sie Ausnahmesituationen abfangen, die entstehen, wenn Ihr Anwendungsprogramm derart Unsinn macht, dass der Prozessorkern an der weiteren Ausführung von Instruktionen gehindert wird.

Die Erkennung von solchen Ausnahmesituationen erledigt der ARM-Kern von selbst; Sie sollen daran anknüpfen, eine Meldung über Art und Ort der Ausnahme ausgeben und das System anhalten. Es darf auch gerne eine hilfreichere Meldung mit weiteren Informationen sein.

Um dies zu erreichen, sind folgende Teile zu erledigen (nicht unbedingt in dieser Reihenfolge):

(a) Entwickeln Sie entsprechende Handler für die verschiedenen Ausnahmen.

(b) Bei der Ausführung eines Handlers befindet sich der ARM-Kern in einem anderen Modus. Bei ARM haben die verschiedenen Modi unterschiedliche Stacks. Überlegen Sie also, wo Sie die Stacks im Speicher ablegen wollen und initialisiert sämtliche Stackpointer des Prozessors. (Stacks sind bei ARM übrigens full-descending gemäß des Procedure Call Standard for the ARM Architecture, an den sich der gcc hält.)

(c) Wenn eine Ausnahme auftritt, setzt der ARM-Kern den program counter (PC) auf eine feste, Ausnahme-spezifische Adresse. Bei ARM ist die Interrupt Vektor Tabelle (IVT) hart verdrahtet, sodass Sie mit den vorgegebenen Adressen zurechtkommen müssen, die sehr eng beieinander liegen. Schreiben Sie entsprechende Instruktionen in diesen Speicherbereich (der mangels besserer Begriffe dennoch als IVT bezeichnet wird), sodass die tatsächlichen Handler ausgeführt werden.

(d) Im gegenwärtigen Zustand befindet sich im Speicherbereich der IVT kein RAM! Bevor Sie also Ihre Handler installieren, führen Sie ein Memory-Remap durch, um dort änderbaren Speicher einzublenden.

Zum Testen bzw. zur Demonstration von zumindest einem Teil Ihrer Handler müssen Sie Code schreiben, der entsprechende Ausnahmen provoziert. Also:

(e) Schreiben Sie eine „Anwendung“, die in Abhängigkeit von einer Benutzereingabe entweder einen Data abort erzeugt, einen Software interrupt auslöst oder

eine Undefined instruction ausführt. (Ein Prefetch abort ist derzeit noch nicht möglich. Interrupts sind Teil der nächsten Aufgabe.)

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_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`