1 Übungsaufgabe #1: Systemaufrufe in StuBSmI

Ziel dieser Aufgabe ist es, das aus der *Betriebssysteme*-Übung bekannte OOStuBS durch Einführung von Systemcalls um eine Trennung der Privilegien von Kernel und Userspace zu erweitern. Dies stellt den ersten Schritt auf dem Weg zu StuBSmI (Studenten-Betriebssystem mit Isolation) dar. Als Ausgangsbasis wird eine leicht angepasste Version von OOStuBS unter `/proj/i4bs/vorgaben/stubsmi.tar.gz` zur Verfügung gestellt.

1.1 Ausführen der Anwendungen auf Ring 3

Im ersten Schritt sollte das System zunächst so angepasst werden, dass der Code der Anwendungen stets auf Ring 3 ausgeführt und nur die Behandlung der Interrupts (insbesondere Zeitscheibenscheduling-Interrupts) auf Ring 0 stattfindet. Erst im zweiten Schritt wird dann eine Schnittstelle für Systemaufrufe eingeführt, die auch das synchrone Betreten des Kerns zur Ausführung privilegierter Operations ermöglicht.

1.1.1 GDT erweitern

Bisher setzt OOStuBS für den Betrieb im *Protected Mode* eine *Global Descriptor Table* (GDT) auf, welche in zwei Einträgen die Code- und Daten-Segmente für Ring 0 beschreibt. Für einen Betrieb in Ring 3 müssen zwei weitere Einträge angelegt werden, die analog dazu den gleichen Zugriff von Ring 3 aus ermöglichen. Ein weiterer neuer Eintrag bildet den TSS-Deskriptor (*Task Stack Segment*), der im Wesentlichen kontrolliert, auf welchen Wert der Stackzeiger gesetzt wird, sobald eine Programmunterbrechung einen Wechsel auf Ring 0 auslöst. Die Strukturen dieser Deskriptoren sind im dritten Band des dreiteiligen IA-32-Entwicklerhandbuchs[1] in den Abschnitten „Segment Descriptors“ (3.4.5) und „Task Management Data Structures“ (7.2.2) detailliert beschrieben.

1.1.2 Kernelstacks einführen

Die Ausführung von Ring-0-Code soll für jede Anwendung auf einem eigenen, vom normalen Stack getrennten Kernelstack stattfinden. Neben der entsprechenden Erweiterung der `Coroutine`-Klasse muss auch die `Dispatcher`-Klasse dahingehend angepasst werden, dass jeweils vor dem Einlasten einer anderen Anwendung der Kernelstackpointer dieser Anwendung im TSS-Deskriptor gesetzt wird.

1.1.3 Initiales Verlassen von Ring 0

Um nun beim Einlasten des ersten Fadens den Ring 0 zu verlassen, muss zum einen der bisherige `toc_settle()` auf den Kernelstack angewendet werden, und zum anderen die dadurch angesprungene `kickoff()`-Funktion anstatt des direkten Aufrufs der virtuellen `action()`-Methode den Ringwechsel veranlassen, indem sie einen Stack aufbaut, der quasi vorgibt, durch einen Wechsel von 0 nach 3 entstanden zu sein, und durch eine `iret`-Instruktion schließlich über eine neue Trampolinfunktion `kickoff_user()` die `action()`-Methode der Anwendung auf Ring 3 betritt. Eine Beschreibung dieses Aufbaus ist im Intel-Handbuch unter „Exception and Interrupt Handlung“ (6.12) zu finden.

1.2 Systemaufrufschnittstelle

Für einen synchronen Weg zurück aus der Anwendungsebene auf Ring 0 soll nun im zweiten Teil die eigentliche Schnittstelle für Systemaufrufe eingeführt werden.

1.2.1 Ausnahmebehandlung

Das Auslösen eines Traps per `int`-Instruktion ist standardmäßig eine privilegierte Operation, die nicht ohne weiteres für Code auf Ring 3 zulässig ist und mit einem `General Protection Fault` quittiert würde. Im zugehörigen Eintrag in der *Interrupt Descriptor Table* kann jedoch für einzelne Vektoren (für Systemaufrufe beispielsweise Nummer `0x42`) die Auslösung durch Usercode erlaubt werden (siehe wiederum Abschnitt 6.12 im Intel-Handbuch). Da ein Systemaufruf-Trap keine Geräteunterbrechung im Sinne der `Plugbox`+`Gate`-Konstruktion ist, sollte dafür auch eine separate Behandlungsfunktion angelegt werden.

1.2.2 Parameterübergabe

Durch den Wechsel auf den Kernelstack bei Behandlung eines Systemaufruf-Traps ist die Übergabe von Parametern auf dem Stack wie bei normalen Funktionsaufrufen nicht möglich. Stattdessen müssen die Aufrufstümpfe (engl. *stubs*) so gestaltet sein, dass die übergebenen Parameter in Registern über den Privilegebenenwechsel

[1] ist auf der Übungswebseite verlinkt

hinweg „gehievt“ werden. Passend dazu muss dann auch die Assembler-Behandlungsfunktion dasselbe umgekehrt machen, indem sie die Registerinhalte auf dem Stack (jetzt Kernelstack!) ablegt. Der *Syscall Dispatcher* wählt dann anhand eines identifizierenden Parameters die eigentliche, aufzurufende Implementierung aus.
Folgende Systemaufrufe sollen implementiert werden. (Die genaue Semantik darf nach eigenem Ermessen *sinnvoll* festgelegt werden.)

- `size_t write(int fd, const void *buf, size_t len, int x = -1, int y -1)`
- `size_t read(int fd, void *buf, size_t len)`
- `void sleep(int ms)`
- `int sem_init(int semid, int value)`
- `void sem_destroy(int semid)`
- `void sem_wait(int semid)`
- `void sem_signal(int semid)`

Hinweise:

- Es bietet sich an, den `write`-Syscall hinter einem `O_Stream`-kompatiblen Wrapper zu verbergen.
- Zur leichteren Fehlersuche empfiehlt sich, Makros für Assertions und Kernelpanics anzulegen, die den Fehlerort mit Hilfe `__LINE__`, `__FILE__` und `__func__` anzeigen.
- Wir empfehlen den mitgelieferten Header `types.h` als Quelle für die üblichen primitiven Typen fester Länge.

Abgabe: am 23.05.2013