

1. Aufgabenblatt zum Modul „Isolation und Schutz in Betriebssystemen“

Alle Materialien finden Sie in ILIAS

Lernziel

Verstehen wie bei der x86-Architektur Anwendungscode im User-Mode (Ring 3) ausgeführt wird und wie man damit verhindert, dass ein User-Thread privilegierte Instruktionen ausführt oder auf Ports zugreift.

1. Aufgabe: Global Descriptor Table (GDT)

Zunächst benötigen wir in der vorhandenen GDT (liegt bei der Marke `_gdt`) zwei weitere Einträge, einen für Ring 3 Code und einen für Ring 3 Daten. Das Limit der GDT muss ebenfalls angepasst werden, da wir die Anzahl der Einträge verändern. Das Limit liegt bei der Marke `_gdt_80`. Alle Arbeiten in dieser Aufgabe sind in Assembler in der Datei `boot.asm` durchzuführen.

Wichtige Information für diese Aufgabe finden Sie in Intel Software Developer's Manual Volume 3 in Kapitel 3.4.5 Segment Descriptors.

2. Aufgabe: Task State Segment (TSS)

Als nächstes benötigen wir ein TSS, damit der Prozessor den Kernel-Stack findet, wenn beispielsweise ein Thread im Ring 3 durch einen Interrupt unterbrochen wird. Hierfür nutzt der Prozessor im TSS den Eintrag `rsp0` (Stack-Zeiger für den Ring 0). In der Vorgabe befindet sich bereits ein TSS (ohne IO-Bitmap; lassen wir absichtlich weg, dadurch sind jegliche Portzugriffe im User-Mode unterbunden) an der Marke `_tss`. Da wir nur einen CPU-Kern nutzen, reicht ein einziges TSS für unser Betriebssystem.

Damit das TSS genutzt werden kann muss ein TSS-Deskriptor (TSSD) in der GDT hinzugefügt werden. Bis auf die Basis-Adresse können alle Informationen im TSSD direkt statisch eingetragen werden. Zusätzlich muss nochmals das Limit in `_gdt_80` angepasst werden (wie bei Aufgabe 1).

Achtung: der TSSD-Eintrag hat die doppelte Größe!

Die Basisadresse des TSS muss zur Laufzeit mithilfe der Funktion `_tss_set_base_address` in den TSSD eingetragen werden, diese Funktion wird in der Vorgabe bereits an der richtigen Stelle aufgerufen (nur einmal beim Bootvorgang). Um einfach an die Basisadresse des TSSD-Eintrags zu gelangen definieren Sie sich in der GDT vor dem TSSD-Eintrag in Assembler eine Marke (engl. label).

Nun benötigen wir noch eine Funktion zum Setzen des Kernel-Stack-Zeigers (`rsp0`) in unserem TSS. Hierfür gibt es in der Vorgabe die Marke `_tss_set_rsp0`. Diese Funktion wird bereits an der richtigen Stelle in der Vorgabe aufgerufen und hierbei wird als Parameter der Zeiger auf den Stack übergeben, welcher aktuell verwendet wird. Später nutzen wir diese Funktion in Rust, um bei jedem Thread-Wechsel den `rsp0` (Kernel-Stack-Pointer) im TSS zu sichern.

Zuletzt muss noch das TSS-Register (TSSR) mit dem Befehl `ltr` geladen werden. Die richtige Stelle ist in `boot.asm` durch einen Kommentar markiert.

Der Speicherplatz des initialen Stacks ist an der Marke `_init_stack` definiert. Er wird aber sobald der Scheduler läuft nicht mehr verwendet. Alle Stacks (User- und Kernel-Mode) aller Threads liegen dann im Heap und werden mit unserem Allokator alloziert.

Alle Arbeiten in dieser Aufgabe sind in Assembler in der Datei `boot.asm` durchzuführen.

Wichtige Information für diese Aufgabe finden Sie in Intel Software Developer's Manual Volume 3 in Kapitel 7.2 Task Management Data Structures

3. Aufgabe: Threads im Ring 3 starten

User-Level-Threads (laufen im Ring 3) benötigen immer zwei Stacks, einen für den User- und einen für den Kernel-Mode. Der Einfachheit halber allozieren wir immer zwei Stacks, auch für reine Kernel-Threads. Hierfür muss die `struct Thread` in `threads.rs` angepasst werden.

Die Thread-Umschaltung ist in `_thread_switch` fertig vorgegeben. Neben dem Sichern und Wiederherstellen der Register muss auch der Kernel-Stack im TSS umgeschaltet werden. Dafür ist der Parameter `then_rsp0_end`. Der User-Level-Stack-Zeiger muss nicht extra verwaltet werden, da er automatisch durch die Hardware bei einem Thread-Wechsel gesichert wird. Der Thread-Wechsel erfolgt nur aus dem Timer-Interrupt heraus und dafür wechselt der Prozessor automatisch von Ring 3 nach Ring 0 und sichert den Stack-Zeiger des unterbrochenen User-Level Threads auf dem Kernel-Stack.

Der vorhandene Thread-Start mithilfe von `prepare_kernel_stack` und `start` bleibt unverändert. Damit startet jeder Thread zunächst immer im Ring 0. In `kickoff_kernel_thread` muss der Kernel-Stack im TSS gesetzt werden, mithilfe der Funktion `_tss_set_rsp0`.

Als nächstes muss die Funktion `switch_to_usermode` in `thread.rs` implementiert werden. Sie soll die Assembler-Funktion `_thread_user_start` verwenden, um mit `iretq` einen Ring-Wechsel durchzuführen (anders geht das nicht). In `switch_to_usermode` muss ein Stackframe gebaut werden, wie er bei einem Interrupt mit Privilegienwechsel vorliegt. Für SS und CS sind die entsprechenden User-Mode-Einträge in der GDT zu verwenden (siehe Aufgabe 1).

Achtung: für die Stackeinträge CS und SS muss RPL = 3 gesetzt werden!

Hierdurch sollten wir dann in `kickoff_user_thread` „landen“ und der Prozessor sollte dann in Ring 3 sein.

Am besten das Umschalten in den Ring 3 vorerst nur mit einem Thread testen, beispielsweise dem Idle-Thread, also versuchen den Idle-Thread im Ring 3 laufen zu lassen. Später soll der Idle-Thread natürlich im Ring 0 ausgeführt werden.

Hinweis: wie kann man erkennen, ob man im User-Mode ist?

- Einen Port-Befehl ausführen, dieser sollte eine General Protection Fault (GPF) auslösen.
- Oder mit dem Debugger einen Breakpoint auf `kickoff_user_thread` setzen und prüfen, ob RPL=3 im `cs`-Register steht.

Alle Arbeiten in dieser Aufgabe sind in Rust in der Datei `thread.rs` durchzuführen.

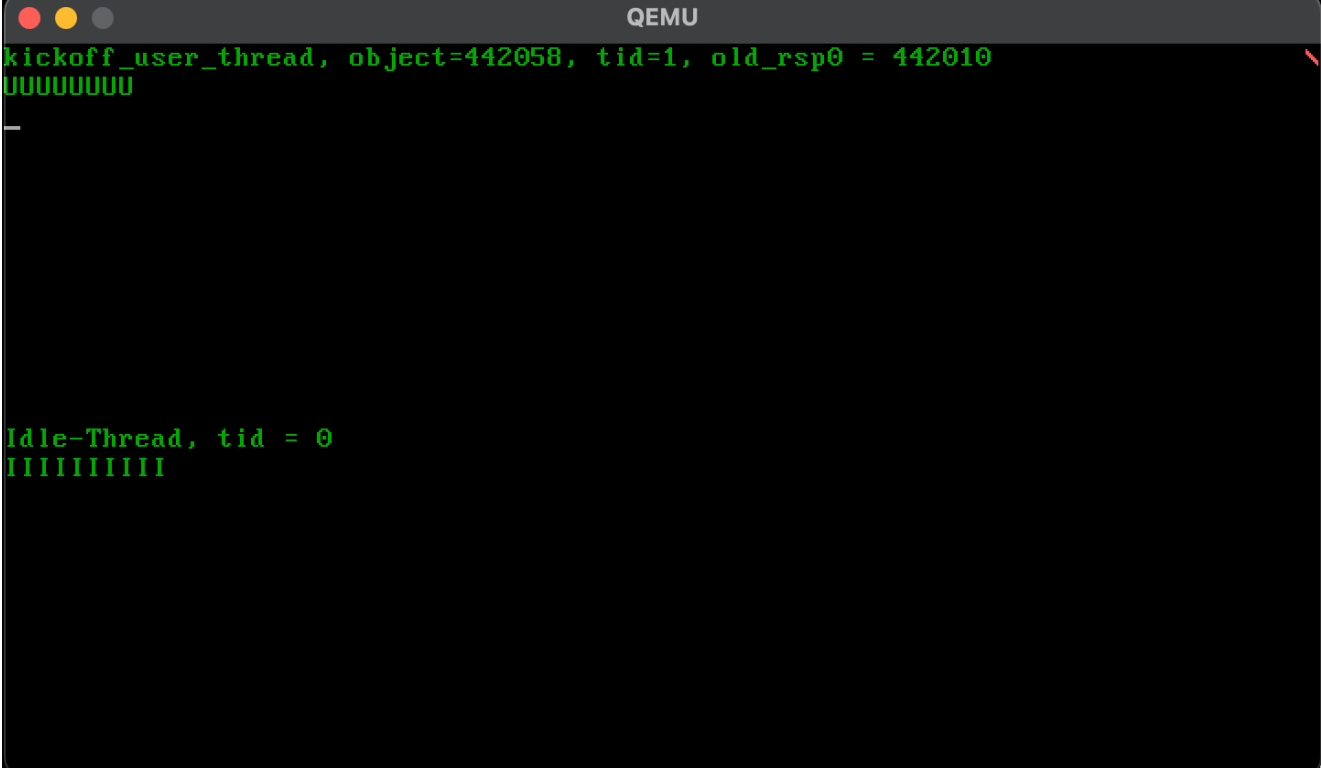
Wichtige Information für diese Aufgabe finden Sie in Intel Software Developer's Manual Volume 3 in Kapitel 6.12 Exception und Interrupt Handling. Der Aufbau des Registers RFLAGS findet man bei Intel oder hier: https://en.wikipedia.org/wiki/FLAGS_register. Wir benötigen für den gefälschten Interrupt-Stackframe kein Error-Code.

Mögliches Testszenario

Ein mögliches Testszenario für dieses Übungsblatt könnte wie folgt aussehen. Der Idle-Thread gibt fortlaufend, aber mit Verzögerung, ein I aus und ein User-Level-Thread immer ein U (auch mit Verzögerung). Dies ist so bereits in der Vorgabe enthalten.

Für den Idle-Thread könnte man ein eigenes `bprint!` Makro verwenden, um in der zweiten Bildschirmhälfte die Ausgabe zu platzieren, siehe nachstehendes Bild.

Als Verzögerung kann einfach eine lang-laufende Schleife verwendet werden, die nichts macht.



```
QEMU
kickoff_user_thread, object=442058, tid=1, old_rsp0 = 442010
UUUUUUUUUU

Idle-Thread, tid = 0
IIIIIIIIII
```