Heinrich Heine
Universität Düsseldorf

UNIVERSITÄT DÜSSELDORF
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Prof. Dr. Michael Schöttner
Dr. Fabian Ruhland

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

Alle Materialien finden Sie in ILIAS

Lernziel

Das Ziel dieser Aufgabe ist es, eine grundlegende Paging-Funktionalität zu implementieren, damit User-Level-Threads in einem eigenen Adressraum ausgeführt werden können. Die Aufgaben in diesem Blatt dienen dazu das Paging zu realisieren, jedoch werden vorerst nur die Stacks der User-Level-Threads isoliert.

Grundlegende Hinweise zum Vorgehen

Unser bisheriges Betriebssystem verwendet bereits Paging, da dies für den Long-Mode beim x86_64 zwingend erforderlich ist. Dieses Paging ist in `boot.asm` und soll komplett ersetzt werden.

Wir verwenden einen sogenannten „lower-half kernel“, d.h. der untere Teil des virtuellen Adressraums (0 bis 1 TiB – 1) ist für den Kernel vorgesehen. Dies vereinfacht die weitere Entwicklung, da der Kernel unverändert weiterverwendet werden kann. Zudem ist der virtuellen Adressraums für den Kernel groß genug, um den gesamten physischen Adressraum durch ein 1:1 Mapping einfach direkt zugreifen zu können; auch das vereinfacht vieles. Der restliche virtuelle Adressraum ab 1 TiB ist für den User-Mode vorgesehen. Des Weiteren arbeiten wir ausschließlich mit 4 KB Seiten.

Auch der physische Adressraum wird fest aufgeteilt, 0 bis 64 MiB – 1 für den Kernel und der Rest für den User-Mode. Auch dies dient dazu die Entwicklung und das Debugging zu vereinfachen. Diese Aufteilungen ist durch Konstanten in `consts.rs` definiert.

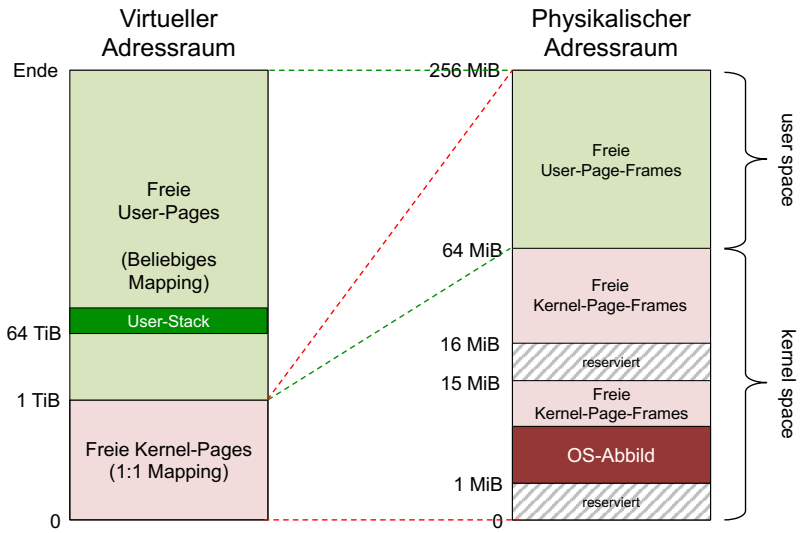

Abbildung 1, Aufteilung der Adressräume

Aufgabe 1: Seitentabellen für den Kernel

Wir verwenden ein vier-stufiges Paging mit ausschließlich 4 KB Seiten (keine 2 MB Seiten). Der gesamte physische Adressraum soll im Kernel 1:1 gemappt werden. Dadurch kann der Kernel immer alle physischen Adressen zugreifen. Hierfür soll in `pages.rs` eine Funktion `mmap_kernel` geschrieben werden (es empfiehlt sich eine rekursive Lösung). Die höchste physische Adresse ist nach dem Initialisieren des Page-Frame-Allokators bekannt. Für die Seitentabellen müssen Page-Frames alloziert werden, aber auf der untersten Eben nicht, da hier der bestehende physikalische Speicher nur „gemappt" wird.

Die Funktion `mmap_kernel` wird von `pg_init_kernel_tables` aufgerufen und letztere Funktion muss in `kmain` aufgerufen werden.

Bezügliche der Seitentabelleneinträge lassen wir vorerst alle Einträge im Ring 3 zugreifbar, löschen also nicht das User-Bit. Das ist noch notwendig, damit wir den Code im Ring 3 ausführen können, wird aber in einem späteren Übungsblatt abgeschafft. Zudem setzen wir alle Seiten auf schreibbar und sofern mit Page-Frames unterlegt auf „Präsent". Um andere mögliche Bits in den Seitentabelleneinträgen, wie Caching, No-Execute, Protection Keys etc., kümmern wir uns nicht.

Die erste Seite 0 sollte auf nicht-Präsent gesetzt werden, um Null-Pointer-Zugriffe abfangen und erkennen zu können.

Die Seitentabellen sollten zuerst ohne User-Mode Threads und ohne Interrupts getestet werden. Falls beim Einrichten der Seitentabellen ein Fehler vorhanden ist, führt dies i.d.R. nach dem Setzen des CR3-Registers sofort zu einem Neustart.

Wenn das funktioniert sollte der Nullpointer-Zugriff geprüft werden. Es sollte ein Page-Fault auftreten und die Adresse der Instruktion die den Page-Fault ausgelöst hat steht in CR2 und sollte in `print_exception` ausgegeben werden (Funktion ist in `int_dispatcher.rs`).

Sofern der Kernel weiterhin funktioniert, können wieder die Interrupts aktiviert werden sowie unsere bestehenden User-Threads getestet werden.

In folgenden Dateien muss gearbeitet werden: `frames.rs` und `int_dispatcher.rs` Nicht auch in Pages.rs?

Wichtige Information für diese Aufgabe finden Sie in Intel Software Developer's Manual Volume 3 in Kapitel 4.5 IA-32e Paging.

Aufgabe 2: Seitentabellen für User-Level-Threads

Nun soll eine erste Isolation der Threads im User-Mode erfolgen, zunächst nur für die User-Mode-Stacks. Der Stack jedes User-Mode Stacks soll an einem gegebenen virtuellen Adressbereich liegen (64 TiB bis 64 TiB + 64 KiB) liegen, siehe `consts.rs`. Der User-Stack soll also nicht mehr über den globalen Heap-Allokator alloziert werden. Damit dieser Adressbereich genutzt werden kann muss ein Mapping in den Seitentabellen eingerichtet werden. Hierfür muss in `pages.rs` die Funktion `pg_mmap_user_stack` implementiert werden, welche in `stack.rs` in `new` für das Allozieren des User-Stacks verwendet werden soll. Jeder Thread hat auch einen Kernel-Stack, dieser soll nach wie vor über den Allokator alloziert werden.

Jeder Thread bekommt so seinen eigenen Adressraum. Der Einfachheit halber empfiehlt es sich die Tabellen für den Kernel für jeden Adressraum zu duplizieren. Dadurch verschwenden wir etwas Speicher, aber die Entwicklung vereinfacht sich deutlich.

Bei jedem Thread-Wechsel muss nun auch der Adressraum umgeschaltet werden! Hierzu muss `_thread_switch` in `thread.asm` angepasst werden. Hier wird nun ein neuer Parameter übergeben `then_pml4`, der Einstieg in die Seitentabellen für den nächsten Thread.

Testen Sie diese Aufgabe mit zwei User-Threads, beispielweise Threads die einen Zähler ausgeben. Prüfen Sie auch mit gdb, dass die Stacks wirklich alle an der gleichen virtuellen Adresse beginnen und die Isolation funktioniert!

In folgenden Dateien muss gearbeitet werden: `thread.rs, stack.rs, paging.rs` und `thread.asm`

Abschließende Bemerkungen

- *Die Isolation des Codes folgt später. Wir haben zwar bereits eine Systemaufrufsschnittstelle, können aber daran vorbei noch beliebige Funktionen des Kernels aufrufen.*
- *Auch die Isolation des Heaps folgt später. Wir verwenden also vorerst unseren globalen Allokator für jeden User-Level Thread, welcher noch Speicher im Kernel-Space verwendet.*
- *Beides funktioniert noch, da wir die die Seiten für den Kernel noch nicht geschützt haben.*