2 Übungsaufgabe #2: Paging in StuBSmI

Ziel dieser Aufgabe ist es, StuBSmI um grundlegende Paging-Funktionalität zu erweitern, Anwendungsprozesse voneinander zu isolieren und Kernel- und Anwendercode voneinander zu trennen. Für diese Aufgabe werden Hilfsdateien unter `/proj/i4bs/vorgaben/aufgabe2.tar.gz` zur Verfügung gestellt.
Um die Initialisierung des Paging einfach zu halten, soll StuBSmI einen „lower-half“ Kernel darstellen, d.h. die virtuellen Adressen von 0x0 bis z.B. 32 MiB gehören dem Kernel und entsprechen den darunterliegenden physikalischen Adressen. Anwendungen könnten also virtuell direkt dort anfangen wo der Kernelbereich aufhört.

2.1 Trennung von Kernel- und Usercode

Um später einfacher zwischen Kern und Anwendungen isolieren zu können, soll StuBSmI getrennt von den Anwendungen kompiliert werden. Das Buildsystem muss entsprechend angepasst werden. Zusätzlich soll eine Bibliothek „libsys“ erstellt werden, in der sich die Syscall-Stubs für die Anwendungen befinden. Mit Hilfe dieser Bibliothek soll nun jede Anwendung für sich kompiliert werden, ohne direkt gegen den Kernel zu linken oder Teile davon zu `#include`-en. Damit weiterhin die Konstruktoren von globalen Objekten im Anwendungscode ausgeführt werden, benötigt jede Anwendung die mitgelieferte Datei `init.cc`. Zusätzlich ist ein Linkerskript erforderlich, das den endgültigen Aufbau der ausführbaren Datei beschreibt. Dieses Skript sollte unter anderem eine sinnvolle Startadresse definieren, ansonsten ähnelt es aber im Wesentlichen dem Linkerskript des Kernels. Mithilfe des Programms `objcopy` lassen sich dann aus den einzelnen, im ELF-Format vorliegenden Anwendungs-Binaries sogenannte „flat“ Binaries generieren, also komplette Speicherabbilder der Programme, die direkt (ohne ELF-Loader zur Laufzeit) zur Ausführung gebracht werden können. Das BSS-Segment sollte dabei nicht vergessen werden (`--set-section-flags .bss=alloc,load,contents`).
Multiboot-konforme Bootloader wie GNU GRUB, sowie auch QEMU unterstützen das Laden einer sogenannten *initial RAM disk (initrd)* zusätzlich zum Kernelimage. Die Anwendungen sollen alle in eine *initrd* gepackt werden, die mit einem Header versehen wird, in welchem die Informationen zu Anzahl und Größe der Anwendungsbinaries vorliegen. Das mitgelieferte Werkzeug `imgbuilder.cc` übernimmt diese Aufgabe; das Auslesen zur Laufzeit muss allerdings selbst implementiert werden. Die Speicherposition und Größe der geladenen `initrd` wird vom Bootloader zur Verfügung gestellt, siehe Abschnitt 2.2.

2.2 Physikalischer Speicher

Um im weiteren Verlauf die Datenstrukturen für das Paging anlegen zu können, muss zunächst der verfügbare physikalische Speicher ermittelt werden.
Der Bootloader stellt dem Betriebssystem hierfür eine Liste aller vorhandenen, nicht durch Geräte oder Busse belegten Speicherbereiche in einem dokumentierten Format[1] zur Verfügung. Hier ist darauf zu achten, dass bereits von Kernel und *initrd* belegte Speicherbereiche nicht automatisch von dieser Liste ausgenommen werden, sondern explizit herausgefiltert werden müssen. Achtung: die angegebenen Bereiche können sich überlappen und widersprüchlich sein. Im Zweifelsfall hat *nicht-frei* (reserved) Präzedenz.

2.3 Paging-Datenstrukturen

Datenstrukturen, die folgende Informationen speichern, könnten sich als nützlich erweisen:

- Freie physikalische Seiten oberhalb der Kernelgrenze
- Freie physikalische Seiten unterhalb der Kernelgrenze
- Prozessekontrollblöcke und Kernelstacks (dynamisch oder statisch mit fester Obergrenze ist egal)

Nun sollen die *page directories* und *page tables* für die Anwendungsprozesse aufgebaut werden. Dabei ist auch zu beachten, dass die Prozesse einen les- und schreibbaren Stack benötigen und bei der Adresse, die im Linkerskript definiert wurde starten.
Soll nun im Dispatcher auf einen anderen Prozess gewechselt werden (`Dispatcher::go`, `Dispatcher::dispatch`) muss auch das Mapping des nächsten Prozesses aktiv werden.
Wenn alles funktioniert können Prozesse nicht auf die Speicherbereiche von anderen Prozessen zugreifen und auch nicht auf den immer eingeblendeten Kernelbereich. Syscalls und Interrupts sollten wieder funktionieren und der Kernel kann problemlos auf den Speicher des aktuell unterbrochenen Prozesses zugreifen.

[1] `http://www.gnu.org/software/grub/manual/multiboot/multiboot.html#Boot-information-format`

Hinweise:

- Genaue Informationen zum Paging finden sich im Intel-Handbuch in Kapitel 4.
- Die Multiboot-Referenz beschreibt einige Datenstrukturen sehr merkwürdig, daher benutzt man am besten gleich die angebotene Headerdatei, die auch schon mit Beispiel-Code veranschaulicht ist.
- Es ist sinnvoll, die erste Page (ab Adresse 0x0) dauerhaft nicht zu mappen und den restlichen Bereich bis 1 MiB nicht zu benutzen (dort sind viele Geräte eingeblendet).
- Um mehr Platz im Kernelbereich zu haben, bietet es sich an die *initrd* an geeigneter Stelle bei der Initialisierung zu verschieben, dieser Speicherbereich kann dann als frei markiert werden.
- Es empfiehlt sich, erstmal nur ein Mapping für den Kernelbereich aufzusetzen, der CPU zu übergeben und die korrekte Funktionalität zu testen.
- Der IOAPIC und der LAPIC greifen auf Adressen zu die u.U. außerhalb des Kernelbereichs liegen.
- Es ist in dieser Aufgabe nicht nötig, eine Behandlung für Pagefaults zu implementieren.

Abgabe: am 10.06.2015