Heinrich Heine
Universität Düsseldorf

UNIVERSITÄT DÜSSELDORF
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Prof. Dr. Michael Schöttner
Dr. Fabian Ruhland

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

Alle Materialien finden Sie in ILIAS

Lernziel

Das Ziel dieser Aufgabe ist es, den Kernel-Code vom Anwendungs-Code zu isolieren. Insbesondere sollen Anwendungen separat compiliert werden und vom Betriebssystem nur noch die System-aufrufsschnittstelle verwenden können sowie evt. eigene Laufzeitfunktionen. Wir beschränken uns auf einen Thread pro Anwendung sowie vorerst auf eine einzige Anwendung.

Aufgabe 1: Getrennte Übersetzung von Anwendungen

Entfernen Sie die Anwendungen die sich im Benutzerverzeichnis `user` befinden in einen separaten Pfad und passen Sie das Build-System entsprechend an. Eine mögliche Ordnerstruktur finden sie auf der letzten Seite.

Verwenden Sie zum Testen eine einfache Anwendung, welche nur den Systemaufruf `usr_hello_world` aufruft.

Compilieren Sie die Anwendung als separate Library (static lib). Eventuelle Laufzeitfunktionen, müssen mitcompiliert werden, da wir keine Shared-Libraries haben. Insbesondere auch die Datei `user_api.rs` für die Systemaufrufe. Diese kann vorerst einfach im Quelltextbaum der Anwendung dupliziert werden.

Als Einstiegsfunktion für jede Anwendung empfiehlt sich als Funktionsname `main`.

Die Dateien: `Cargo.toml, Makefile, hhu_tosr_app.json` sowie `linker.x86_64-elf-ld` sind für eine Anwendung fast gleich wie für das Betriebssystem. Beim Linker-Skript muss als Startadresse 1 TiB eingetragen werden, hier beginnt der Adressbereich für den User-Mode, siehe Übungsblatt 4. Der Boot-Teil entfällt natürlich. Der Name der Einstiegsfunktion wird im Linker-Script in der ersten Zeile `ENTRY` festgelegt. Damit die Einstiegsfunktion auch sicher am Anfang des Images liegt muss vor die Einstiegsfunktion noch eine Section-Zuordnung vorgenommen werden, siehe unten. Zudem muss im Linker-Script diese Section am Anfang von `.text` stehen, siehe ebenfalls unten.

Auszug: `lib.rs`

```
#[link_section = ".main"]
#[no_mangle]
pub extern "C" fn main() -> ! {
```

Auszug: `linker.ld`

```
ENTRY(main)
...
.text :
{
  *(.main*)
  *(.text*)
}
```

Hinweise zu cargo-make finden Sie hier: https://github.com/sagiegurari/cargo-make
Und vorerst soll kein Allokator in Anwendungen verwendet werden. Dieser kommt erst später.

Aufgabe 2: Flat-Binary

Der Linker erzeugt für die Anwendung eine ELF-Datei. Mithilfe des Programms `objcopy` kann eine ELF-Datei in ein Flat-Binary umgewandelt werden. Hierdurch kann ein Lader umgangen werden und wir müssen nicht die ELF-Header parsen. Der Aufruf um eine ELF-Datei in ein Flat-Binary umzuwandeln sieht wie folgt aus:

```
objcopy -O binary hello.elf build/app.img
        --set-section-flags .bss=alloc,load,contents
```

Die einfachste Möglichkeit eine Flat-Binary-Datei beim Booten in unserem Kernel zu nutzen ist in Form eines Boot-Moduls in Multiboot. Das Einbinden eines Boot-Modules erfolgt als Datei, welche mit in die Iso-Datei geschrieben wird, siehe hier: **https://wiki.osdev.org/Initrd**. Boot-Modules müssen in der `grub.cfg` eingetragen werden, siehe Vorgabe.

Sobald mindestens ein Modul eingebunden wurde ist Bit 3 in den Flags des Multiboot-Headers gesetzt und entsprechend die Felder `mods_count` und `mods_addr`, siehe hier: https://www.gnu.org/software/grub/manual/multiboot/multiboot.html#Boot-modules

Wir unterstützen vorerst nur eine einzige Anwendung in einem Modul, das ist der einfachste Fall. Das Flat-Binary einer Anwendung muss anschließend in `kmain` gefunden werden. Hierzu muss die Funktion `get_app` in `multiboot.rs` geschrieben werden. Diese Funktion wird in `startup.rs` aufgerufen und dann ein Thread mithilfe der neuen Funktion `new_app_thread` in `thread.rs` erzeugt (weitere Infos in Aufgabe 3). Die Adresse der Einstiegsfunktion `entry` in `struct Thread` entspricht `USER_CODE_VM_START`. Anschließend wird in `kmain` der neu erzeugte Thread wie bisher im Scheduler mit `Scheduler::ready` registriert.

Um zu überprüfen, ob der Anwendungscode richtige gefunden wurde, kann im Debugger ab `AppRegion.start` reassembliert werden.

Hinweis: Unter MacOS müssen für den Befehl `objcopy` *die GNU binutils installiert werden und dann lautet der Befehl:* `x86_64-elf-objcopy`

Aufgabe 3: Ein Mapping für das Anwendungsimage

Nachdem das Flat-Binary einer Anwendung im Multiboot-Header gefunden wurde (siehe Aufgabe 2) muss nun noch ein Mapping dafür eingerichtet werden. Zur Erinnerung das Mapping für den User-Mode-Stack haben wir in Übungsblatt 3 programmiert.

Unsere Anwendung wird durch den Linker an die Adresse 1 TiB gelinkt, d.h. die Pages ab 1 TiB müssen auf das bereits geladene Anwendungsimage abbilden. Das Anwendungsimage befindet sich bereits im Speicher und im Kernel haben wir das 1:1 Mapping, d.h. die virtuelle Adresse, an die das Flat-Binary geladen wurde, entspricht der physikalischen. Für das Aufbauen der Seitentabellen muss in `pages.rs` die Funktion `pg_mmap_user_app` geschrieben werden. Diese Funktion wird dann in `Thread::new_app_thread` aufgerufen, um das Flat-Binary einzublenden, siehe Abbildung 1 auf der nächsten Seite.

Wichtig, im Multiboot-Header in `boot.asm` muss noch MULTIBOOT_PAGE_ALIGN gesetzt werden, damit wir das Flat-Binary direkt mappen können, ohne umkopieren.

Damit GDB die Symbolinformationen für eine Anwendung findet, müssen diese per Befehl `add-symbol-file filename address` nachgeladen werden. Dieser Befehl kann auch im `Makefile` verankert werden, unter dem Target `qemu-gdb`. Weitere Infos findet man hier: https://stackoverflow.com/questions/20380204/how-to-load-multiple-symbol-files-in-gdb

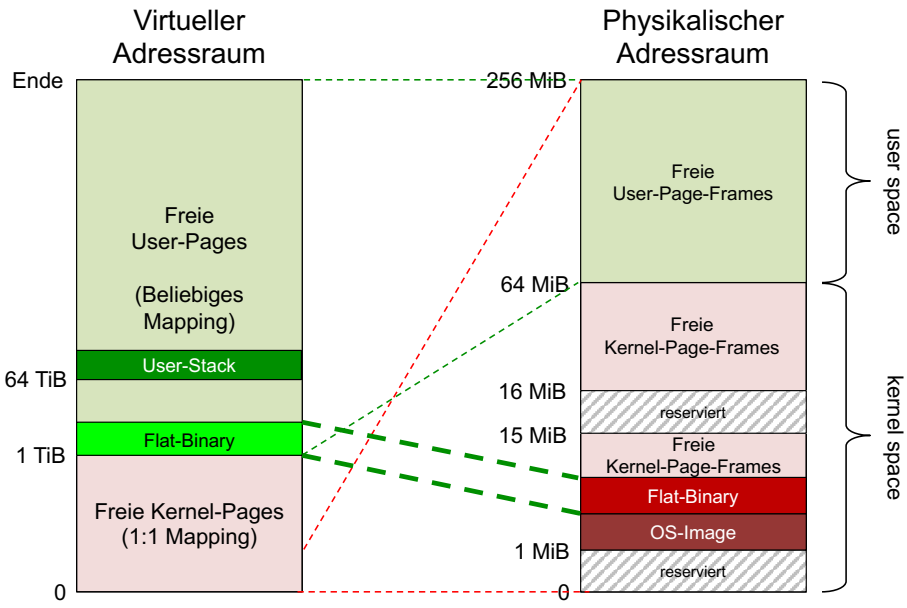

Abbildung 1, Mapping des App-Images

Aufgabe 4: Kernel-Space schützen

Bisher kann jede Anwendung auf den Kernel-Speicherbereich (Adressen kleiner als 1 TiB) zugreifen (lesend und schreibend). Nun soll der Kernel-Speicherbereich über das Paging geschützt werden. In alle Seitentabellen-Einträgen für den Kernel-Speicherbereich (das 1:1 Mapping von 0 – 256 MiB) soll nun das U/S-Bit gelöscht werden.

Anschließend muss noch der Startvorgang eines User-Threads anpasst werden, da wir in der Assemblerfunktion `_thread_user_start` nicht mehr nach `kickoff_user_thread` zurückspringen können, da dieser Code im Kernel-Image liegt, welches nun geschützt ist, und dies eine Page-Fault auslösen würde. Wir bauen daher die Funktion `switch_to_usermode` um, sodass als Rücksprung-Adresse auf dem selbst gebauten Stack direkt die Adresse der `main`-Funktion des Flat-Binary eingetragen wird (das ist 1 TiB, siehe Abb. 1). Somit wird durch die `iretq`-Instruktion in `_thread_user_start` direkt `main` angesprungen. (Die Funktion `kickoff_user_thread` kann gelöscht werden.)

Hinweis zu Aufgabe 1: Mögliche Ordnerstruktur (gelb markierte Dateien fehlen)

```
vorgabe05
├── apps/
├── Cargo.toml
├── hhu_tosr_app.json
├── hhu_tosr_os.json
├── loader/
├── Makefile
├── os
└── target

os
├── Cargo.toml
└── linker.x86_64-elf-ld
├── Makefile
└── src
    ├── boot
    ├── devices
    ├── kernel
    └── mylib

target
├── apps/
├── Cargo.toml
├── hhu_tosr_app/
├── hhu_tosr_os/
└── loesung05.iso

loader/
├── grub.cfg
└── isofiles
    ├── boot
    ├── grub
    │   └── grub.cfg
    ├── app.bin
    └── kernel.elf

apps
├── hello
    ├── Cargo.toml
    ├── Makefile
    └── src
        ├── kernel
        │   ├── mod.rs
        │   └── syscall
        │       ├── mod.rs
        │       └── user_api.rs
        └── lib.rs
└── linker.x86_64-elf-ld
```

Alle Aktionen ab sofort mit Argument: `--no-workspace`
Beispielsweise: `cargo make --no-workspace`