Heinrich Heine
Universität Düsseldorf

UNIVERSITÄT DÜSSELDORF
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Prof. Dr. Michael Schöttner
Dr. Fabian Ruhland

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

Alle Materialien finden Sie in ILIAS

Lernziel
Das Ziel dieser Aufgaben ist es mehrere Anwendungen in einer Tar-Datei verarbeiten und starten zu können. Für jede Anwendung soll ein eigener Prozess, mit je einem Thread, gestartet werden.

Aufgabe 1: Systemaufrufsschnittstelle in einer eigenen Crate `usrlib`
Bisher haben wir die Systemaufrufsschnittstelle, in der Datei `user_api.rs`, dupliziert in der einen Anwendung (siehe Übungsblatt 5). Dies ist offensichtlich keine gute Idee, da Änderungen in jedem Duplikat in jeder Anwendung konsistent gehalten werden müssten. Daher soll der Code (User-Mode-Teil der Systemaufrufe) nun in einer eigenen Crate `usrlib` untergebracht werden. Die Crate soll im Kernel und jeder Anwendung eingebunden werden.

Dies geht in der jeweiligen `Cargo.toml` Datei mit: `usrlib = { path = "../usrlib" }`, die Verzeichnisstruktur ist auf der letzten Seite zu finden.

Für die Crate `usrlib` wird eine `Cargo.toml` Datei benötigt, aber nicht `Makefile.toml`. Die `usrlib` wird automatisch mitübersetzt, wenn sie in einer Anwendung oder dem Betriebssystem eingebunden wird. Nach dem Umbau sollte der Stand vom letzten Übungsblatt wieder funktionieren.

Aufgabe 2: Textausgabe an einer Position im User-Mode
Fügen Sie in `user_api.rs` (siehe Aufgabe 1) einen neuen Systemaufruf hinzu, welcher es ermöglicht utf8 Chars an einer gegebenen Bildschirmposition auszugeben. Die Signatur des Aufrufs sieht wie folgt aus: `usr_print(x: u64, y: u64, buff: *const u8, len: u64)`
Die Implementierung von `sys_print` im Kernel soll mithilfe von `cga::print_byte` erfolgen. Scrolling muss hier nicht implementiert werden und auch keine Verwaltung der Textcursor-Position.

Damit der neue Systemaufruf von den Anwendungen einfach genutzt werden kann soll die Crate `usrlib` aus Aufgabe 1 erweitert werden. Hierzu kann aus dem Kernel die Datei `cga_print.rs` kopiert und angepasst werden. Die Textcursor-Position kann im `WRITER` gespeichert werden, indem man zwei Variablen `x` und `y` einführt. Um die Cursor-Position zu setzen bietet es sich an ein eigenes neues Makro `print_setpos!` zu schreiben. In `write_str` muss der neue Systemaufruf `usr_print` verwendet werden, da wir nicht direkt auf den Bildschirmspeicher zugreifen können. Zudem muss hier dann die Cursor-Position vom `write_str` an `usr_print` übergeben und aktualisiert werden.

Testen Sie den neuen Systemaufruf mithilfe einer Counter-Anwendung. Diese soll an einer festen Position einen Counter ausgeben, der fortlaufend hochgezählt wird. Damit nicht ohne Pause ständig Systemaufrufe ausgelöst werden empfiehlt es sich eine künstliche Verzögerung, beispielweise eine Dummy-Schleife, in der Schleife für die Counter-Ausgabe einzubauen.

Aufgabe 3: Mehrere Anwendungen in einer Tar-Datei
Bisher haben wir eine Anwendung in einem grub-Bootmodul gespeichert. Nun möchten wir mehrere Anwendungen unterstützen. Hierfür bietet sich eine Tar-Datei an, welche als Bootmodul in grub übergeben wird. In ausgewachsenen Betriebssystemen würden die Dateien dann über eine Ramdisk bereitgestellt, siehe auch initrd: https://en.wikipedia.org/wiki/Initial_ramdisk. Wir haben weder ein Dateisystem noch eine Ramdisk, weswegen die Tar-Datei lediglich eingelesen und dann direkt für jede darin enthaltene Anwendung ein Prozess gestartet wird.

Jede Anwendung soll wie bisher an die Adresse 1 TiB gelinkt und als Raw-Image abgelegt werden, siehe Übungsblatt 4. Als Zielordner für die Images soll `/load/isofiles/boo/apps` verwendet werden (dies muss in den Makefiles der Anwendungen entsprechend eingetragen werden).

Das Makefile auf der obersten Ebene muss ebenfalls angepasst werden, sodass alle Raw-Images in einer Tar-Datei zusammengefasst werden. Dies geht mit folgendem Aufruf:
`tar -c -f initrd.tar /load/isofiles/boo/apps/`

Für das Einlesen der Tar-Datei `initrd.tar` verwenden wir die Crate `tar_no_std`. Lesen sie unbedingt die Hinweise zu `tar_no_std`, da es einige Einschränkungen gibt, die für uns jedoch nicht störend sind, siehe hier: https://docs.rs/tar-no-std/latest/tar_no_std/

Da die Crate `tar_no_std` für Fehlermeldungen ausschließlich einen Logger verwendet, muss die Crate `log` im Kernel eingebunden werden und in `kmain` möglichst früh initialisiert werden. In der Vorgabe ist eine passende `Cargo.toml` Datei sowie `startup.rs`.

Das Einlesen der Apps muss in der Funktion `get_apps_from_tar` in `multiboot.rs` programmiert werden. Die `struct AppRegion` wird um einen Dateinamen erweitert, damit hier der Name der Image-Datei (aus dem TarEntry) gespeichert wird. Somit kann später ein laufender Prozess unter anderem abfragen, wie seine Image-Datei heißt (siehe auch letzte Aufgabe). Zudem soll `start` und `end` in der `AppRegion` den Bereich des Flat-Binray im TarEntry speichern.

Zum Testen soll die Counter-Anwendung aus Aufgabe 2 dupliziert werden, sodass wir zwei Counter-Anwendungen in die Tar-Datei packen können. Jede Anwendung soll den Counter an einer anderen Stelle auf dem Bildschirm ausgeben und einen anderen Image-Namen haben.

In dieser Aufgabe sollen in `kmain` vorerst nur die gefunden AppRegionen (Meta-Daten) ausgegeben werden, um zu testen, ob wir der Inhalt der Tar-Datei richtig erkannt wird. Das eigentliche Starten der Apps respektive der Prozesse folgt in Aufgabe 4.

Aufgabe 4: Prozesse
In dieser Aufgabe wird eine Verwaltungsstruktur für Prozesse implementiert sowie der Start von Threads umgebaut, sodass nun Prozesse mit jeweils einem Thread gestartet werden.

Im Scheduler müssen folgende Funktionen zum Starten von Prozessen implementiert werden:

- `spawn_kernel`: für den Kernel-Prozess mit dem Idle-Thread
- `spawn`: erzeugt für eine Anwendung einen Prozess mit einem Haupt-Thread

Alle laufenden Prozesse sollen in `process.rs` in einem Key-Value-Baum (`btree_map`) verwaltet werden. Als Key dient die Prozess-ID `pid` und als Value wird die Prozess-Struktur `Box<Process>` gespeichert. Hierdurch können Prozessinformationen später schnell über die `pid` abgerufen werden. Die Baumstruktor gibt es fertig in der crate `alloc`, siehe Vorgabe.

In `kmain` wird `spawn_kernel` einmal aufgerufen und danach für jede in der Tar-Datei (siehe Aufgabe 3) gefundene Anwendung die Funktion `spawn`. Danach kann der Scheduler wie gewohnt mit `Scheduler::schedule` gestartet werden.

In `thread.rs` sind kleinere Anpassungen notwendig. In `struct Thread` wird nun auch die `pid` gespeichert, damit jeder Thread die Zuordnung zu seinem Prozess kennt. Die Funktion `Thread::new_app_thread` wird nicht mehr benötigt und die Funktion `Thread::new` wurde angepasst; im Wesentlichen wird der Adressraum nun in `spawn` respektive `spawn_kernel` erzeugt, also beim Anlegen des Prozesses und nicht beim Erzeugen eines Threads. Die Adresse auf den Adressraum (Variable `pml4_addr`) speichern wir weiterhin mit jedem Thread, damit die Thread-Umschaltung nicht angepasst werden muss.

Da das Image einer einzelnen Anwendung nun nicht immer auf Seitengrenzen beginnt, muss die Funktion `pg_mmap_user_app` in `pages.rs` angepasst werden. Statt wie bisher lediglich Seitentabelleneinträge auf die bestehenden Adressen der AppRegion einzurichten müssen nun auf der untersten Ebene alle Kacheln alloziert werden; dafür sollen User-Page-Frames verwendet werden.

Nachdem das Mapping eingerichtet wurde muss noch die AppRegion seitenaligniert umkopiert werden!

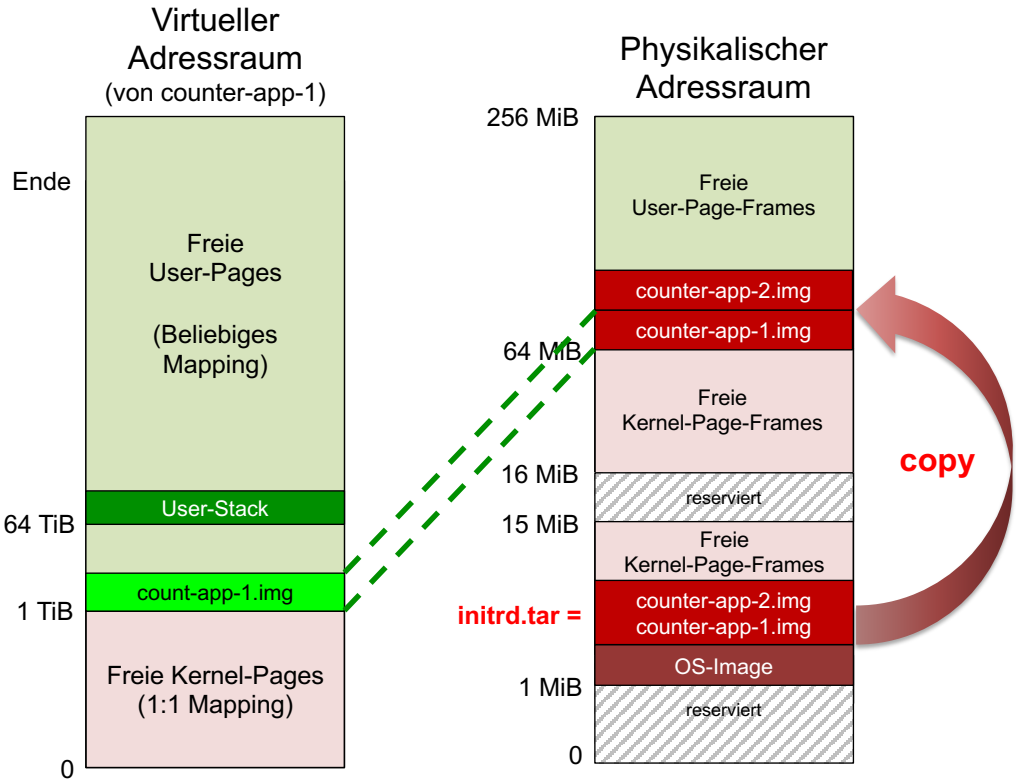

Abbildung 1, Mapping eines App-Images

(Im Prinzip kann man prüfen, ob die Startadresse der AppRegion zufällig schon seitenaligniert ist und dann unsere bisherige Funktion `pg_mmap_user_app` verwenden. Aus Gründen der Einfachheit kann aber auch immer nur eine Funktion mit Umkopieren benutzt werden.)

Wenn hier alles richtig realisiert wurde, sollte man nun die beiden Counter-Anwendungen aus Aufgabe 3 starten können.

Aufgabe 5: Weitere Systemaufrufe, um Prozessinformationen abzufragen
Nun sollen noch folgende beiden Systemaufrufe implementiert werden:

- `sys_getpid`: soll die eigene `pid` zurückgeben.
- `sys_get_app_name`: soll den Namen des Images aktuellen Prozesses zurückgeben.

Diese Aufrufe sollen in den beiden Counter-Testanwendungen verwendet werden. Am Ende könnten das Testszenario wie folgt aussehen.

Abbildung 2, Ausgaben der beiden Counter-Anwendungen

Mögliche finale Ordnerstruktur