Heinrich Heine
Universität Düsseldorf

UNIVERSITÄT DÜSSELDORF
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Prof. Dr. Michael Schöttner
Dr. Fabian Ruhland

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

Alle Materialien finden Sie in ILIAS

Lernziel
Das Ziel dieser Aufgaben ist es virtuelle Adressräume abstrakt durch Virtual Memory Areas (VMAs) zu verwalten, sowie einen Heap im User-Mode für jeden Prozess bereitzustellen sowie den User-Mode Stack eines Threads bei Bedarf dynamisch zu vergrößern.

Aufgabe 1: Virtual Memory Areas (VMAs)
In dieser Aufgabe soll der virtuelle Adressraum eines Prozesses abstrakt mithilfe von Virtual Memory Areas (VMAs), ähnlich wie unter Linux, verwaltet werden. Eine VMA ist eine Region im virtuellen Adressraum, hat also eine Anfangs- und Endadresse (beide seitenaligniert) sowie einen Typ (Code, Stack, Heap) und eine VMA gehört immer genau zu einem Prozess.

Das Paging wird wie bisher verwendet, um separate Adressräume zu realisieren sowie den Speicherschutz des Kernel-Adressbereichs sicherzustellen.

Für die Verwaltung der VMAs verwenden wir die fertige doppelt verkettete Liste `linked_list` aus der Crate `alloc`. Der Anker für die Liste wird in `struct Process` gespeichert.

Wir legen VMAs nur für den User-Mode-Adressbereich an, also nicht für den Kernel. Wenn eine der folgenden Funktionen aufgerufen wird, so soll eine neue VMA angelegt und in die VMA-Liste des zugehörigen Prozesses eingetragen werden:

- `pg_mmap_user_app` (zusätzlicher Parameter `pid`, siehe Vorgabe)
- `pg_mmap_user_stack` (zusätzlicher Parameter `pid`, siehe Vorgabe)
- `pg_mmap_user_heap` (wird in Aufgabe 2 implementiert)

Das hinzufügen einer neuen VMA soll durch die Funktion `process::add_vma` realisiert werden. Diese Funktion soll prüfen, ob die neue VMA mit einer existierenden VMA überlappt und falls ja, einen Fehler zurückgeben und abbrechen. Falls keine Überlappung vorliegt, soll die neue VMA in der VMA-Liste des Prozesses gespeichert werden und anschließend die Page-Tables eingerichtet werden (hier sind keine Änderungen notwendig).

Damit die VMAs eines Prozesses ausgegeben werden können soll ein neuer Systemaufruf `usr_dump_vmas` implementiert und in einer Test-Anwendung ausprobiert werden. Dieser Aufruf soll mit `kprintln!` alle VMAs des aufrufenden Prozesses ausgeben.

Weitere Informationen zu VMAs in Linux sind hier gut erläutert:
https://manybutfinite.com/post/how-the-kernel-manages-your-memory/

Aufgabe 2: Einen Heap für Prozesse
In dieser Aufgabe soll die Crate `usrlib` um einen Allokator erweitert werden, der dann im User-Mode in einer Anwendung genutzt werden kann. Für die Implementierung des Heap-Allokators kann der List-Allokator der Kernel-Heapverwaltung verwendet werden. Der Speicherbereich, den der Allokator verwaltet soll über einen neuen Systemaufruf `mmap_user_heap` angefordert werden. Die Implementierung von `mmap_user_heap` muss in `pages.rs` erfolgen. Der allozierte Adressbereich soll im User-Mode-Bereich sein und für die Page-Frames des Heaps soll der User-Page-Frame-Allokator verwendet werden.

Für die Abgabe reicht es den Heap an einen festen Bereich im virtuellen Adressraum des User-Mode-Bereichs einzublenden.

Damit der Allokator in `usrlib` nicht mit dem Heap-Allokator im Kernel kollidiert muss noch die `user_api.rs` in eine eigene Crate `syscalls` verschoben werden. Die Crate `syscalls` wird dann von allen Anwendungen und dem Kernel eingebunden wohingegen die Crate `usrlib` nur noch von den Anwendungen eingebunden wird.

Testen Sie den Allokator in einer Test-Anwendung in dem Sie mit `Box::new` Daten auf dem Heap anlegen.

Optional:
Normalerweise ist es wünschenswert, dass beim Aufruf von `mmap_user_heap` eine gewünschte Startadresse für das Mapping übergeben werden kann oder 0, wenn der Kernel entscheiden soll, ab wo der Speicherbereich eingeblendet wird. Erweitern Sie die Funktion `mmap_user_heap` entsprechend um einen Parameter Startadresse.

Sofern eine virtuelle Startadresse >0 übergeben wird, muss überprüft werden, ob diese ab dem Start der User-Mode-Adressen liegt und ob, ab dort genügend Speicher frei ist. Diese Prüfung kann mithilfe von VMAs einfach realisiert werden. Die Prüfung auf eine potentielle Überlappung von VMAs haben wir bereits in der Funktion `process::add_vma` in Aufgabe 1 realisiert.

Aufgabe 3: Dynamische Vergrößerung des Stacks
Bisher wurde der Stack für jeden User-Thread mit einer festen Größe alloziert, dies soll nun angepasst werden, sodass der Stack bei Bedarf dynamisch wächst. Hierzu sind keine Änderungen im User-Mode notwendig, jedoch im Kernel.

Am einfachsten reserviert man zum Testen initial nur eine Seite (4 KiB) für den User-Mode Stack (siehe `USER_STACK_VM_SIZE` in `consts.rs`). Der Page-Fault-Handler muss erweitert werden und erkennen, ob der Fehler durch einen Zugriff auf eine User-Mode- Stack-Adresse ausgelöst wurde, falls ja soll eine Seite passend für die Stackerweiterung alloziert werden und dann aus dem Page-Fault-Handler zurückgekehrt werden, sodass der unterbrochene Thread weiterlaufen kann.

Ob ein Page-Fault durch einen Stack-Überlauf ausgelöst wird, kann mithilfe der VMAs überprüft werden, siehe Aufgabe 1.

Zudem benötigen wir eine Testanwendung. Am besten eine einfache rekursive Funktion, beispielsweise die Berechnung der n-ten Fibonacci-Zahl. N wird dann so gewählt wird, dass ein Page-Fault auftritt, wegen einem Stacküberlauf. Dann wird die dynamische Stackvergrößerung eingebaut und nochmals mit dem gleichen n getestet.