1 Übungsaufgabe #1: SMP-fähige Gastebene

Ziel dieser Aufgabe ist es, eine *abstrakte Maschine* zu implementieren, die oberhalb des Betriebssystems **Linux** ausgeführt wird. Dabei wird besonderer Wert auf die *minimale Abstraktion* eines allgemeinen Mehrprozessorsystems gelegt.
Das Ergebnis dieser Aufgabe wird die Basis der folgenden Übungsaufgaben sein.

1.1 Minimale Abstraktion eines Mehrprozessor-Systems

Unter *minimaler Abstraktion* wird eine abstrakte Maschine verstanden, die mit möglichst wenig Aufwand auf möglichst viele verschiedene Plattformen und Betriebssysteme portiert werden kann. Hierzu bietet es sich an die API der abstrakten Maschine minimal zu halten.

1.2 Virtuelle Prozessoren

Ein typisches SMP-System besteht aus einem ausgezeichnetem Prozessor, im Folgenden *Bootprozessor* genannt, und einem oder mehreren *Applikationsprozessor*en. Der *Bootprozessor* übernimmt beim Systemstart (der Initialisierung der Bibliothek) eine Sonderrolle. Er ist dafür zuständig das System zu initialisieren und die *Applikationsprozessor*en zu starten. Die eigentlichen Anwendungsfäden sollen dann auf den *Applikationsprozessoren* laufen. Der *Bootprozessor* hat somit nach dem Starten der *Applikationsprozessoren* eigentlich nichts mehr zu tun. Jedoch darf die `main`-Funktion nicht einfach zurückkehren, da sonst der ganze Prozess mit allen Fäden beendet wird. Abhilfe schafft hier der Systemaufruf `exit`, der nur den aufrufenden Faden beendet. Dieser muss jedoch manuell aktiviert werden (siehe Hinweise), da die Implementierung der C-Funktion `exit` den Systemaufruf `exit_group` verwendet, der die ganze Prozessgruppe beendet.
Im Rahmen dieser Aufgabe sollen diese virtuellen Prozessoren mit Hilfe der Gastebene auf *reale* Prozessoren abgebildet werden, damit *echte* Parallelität zwischen den virtuellen Prozessoren beobachtet werden kann. Das Mittel der Wahl sind dabei Threads, die mittels Funktionen des darunterliegenden Betriebssystems fest an physikalische Prozessoren gebunden werden und so innerhalb der Userspace-Bibliothek als virtuelle Prozessoren dienen.

1.3 Die Hardwareabstraktionsschicht (HAL)

Die Bibliothek wird in *Schichten* entworfen. Die unterste Schicht wird dabei die *Hardwareabstraktionsschicht* genannt und implementiert alle betriebssystem- und plattformabhängigen Funktionen. Damit soll erreicht werden, dass nur diese Schicht angepasst werden muss, wenn die Bibliothek auf ein anderes Betriebssystem und/oder auf eine andere Plattform portiert werden soll.
Für diese Übung wird dabei die Verwendung einer evtl. vorhandenen `pthread`-Bibliothek untersagt. Stattdessen soll die Funktionalität von Linux direkt verwendet werden. Es bieten sich dabei die Systemaufrufe `clone(2)` zum Erzeugen eines neuen Aktivitätsträgers, `sched_setaffinity(2)` zum Binden eines Aktivitätsträgers an eine *reale CPU* und `getcpu(2)` zur Identifizierung der aktuellen CPU an.
Auch die Verwendung von dynamischer Speicherverwaltung (`new, delete, malloc, free`) ist im Rahmen einer Gastebene nicht zu empfehlen, da die vorhandenen Bibliotheken für C/C++ intern `pthread`-Mechanismen zur Synchronisation benutzen, die bei Verwendung der nativen Linux-Systemaufrufe natürlich nicht funktionieren. Stattdessen ist es besser notwendigen Speicher für Objekte oder auch für die Stacks der Applikationsprozessoren mit Hilfe des Systemaufrufs `mmap` zu allokieren.

1.4 Optimierte Bestimmung der Prozessornummer

Das Bestimmen der aktuellen Prozessornummer ist eine Operation, die spätestens bei prozessorlokalen Datenstrukturen innerhalb eurer Bibliothek häufiger vorkommen wird. Da `getcpu` ein Systemcall ist, wird dazu jedes mal in den Kern gewechselt, was diese Operation recht teuer macht. Das Ziel dieser Teilaufgabe ist es, die Bestimmung der Prozessornummer ohne einen Systemaufruf durchzuführen, indem diese über den aktuellen Wert des Stapelzeigers ermittelt wird. Dazu bietet es sich an alle Stacks mit einem enstprechenden Alignement im Speicher abzulegen.

1.5 Nebenläufig benutzbare Ausgabeoperationen

Zum Testen der Implementierung könnt ihr Bildschirmausgaben mit den C++ Stream-Operatoren oder mit `printf` nur eingeschränkt benutzen, da die nebenläufige Benutzung des Terminals in unterschiedlichen Fäden eine reentrante Implementierung der Ausgabeprimitiven erfordert. Problematisch ist hier v.a. die Pufferung, die durch die C Bibliothek erfolgt und auch wieder malloc verwendet.
Implementiert daher einen eigenen, reentranten Ausgabemechanismus. Dafür bietet es sich an einen Puffer beim `printf`-Aufruf temporär auf dem Stack anzulegen und dann die Funktion `vsnprintf (3)` zu verwenden.

Zur Ausgabe der so formatierten Daten auf der Konsole könnt ihr den Systemaufruf `write` verweden, der auch problemlos von mehreren Threads parallel aufgerufen werden kann.

1.6 Testen der Implementierung

Um die Implementierung zu testen, programmiert eine Schleife, die auf jedem Applikationsprozessor sowie auf dem Bootprozessor die jeweilige Prozessornummer ausgibt. Diese simple Implementierung einer Startroutine wird in den nächsten Aufgaben erweitert.

Aufgaben:

→ Identifikation einer minimalen API der abstrakten Mehrprozessormaschine.

→ Implementierung dieses HALs für Linux

→ Entwickeln einer Demonstrationsanwendung (vgl. 1.6)

Hinweise:

- Es soll ein SMP System simuliert werden, das heißt, dass alle Prozessoren **denselben** (virtuellen) Adressraum teilen. Dies gilt auch für diverse andere Systemdatenstrukturen wie die Dateideskriptortabelle oder die Signalmaske. Was genau zwischen den Threads geteilt wird, kann man über die Flags beim Aufruf von `clone` einstellen. Für unsere Problemstellung ist folgende Menge von Flags sinnvoll:

  `CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND | CLONE_THREAD`

- Es ist zweckmäßig, eine Funktion `int boot_cpus(void (*fn)(void), int maxcpus)` vorzusehen, welche die Applikationsprozessoren startet. Diese Funktion bekommt einen Funktionszeiger `*fn` übergeben, welcher die Startroutine der Applikationsprozessoren referenziert. Der Parameter `maxcpus` begrenzt die Anzahl der zu simulierenden Applikationsprozessoren.

- Obwohl sich alle Prozessoren den selben Adressraum teilen, benötigen alle virtuellen CPUs einen eigenen Stapel (engl. *Stack*)! Die Funktion zum Starten der Applikationsprozessoren muss daher Speicher bereitstellen und dafür sorgen, dass die Funktion `*fn` auf den jeweiligen Applikationsprozessoren mit jeweils eigenen Stapeln ablaufen.

- Unter Linux lässt sich Stapelspeicher mit dem Systemaufruf `mmap(2)` am Besten anfordern. Im Gegensatz zur Bibliotheksfunktion `malloc(3)` wird so die Verwendung der Speicherverwaltung der `libc` vermieden. Um Stacks mit `mmap` zu allokieren sind folgende Flags sinnvoll:

  `MAP_PRIVATE | MAP_ANONYMOUS`

- Obwohl die Funktion `*fn` auf allen Applikationsprozessoren dieselbe ist, soll diese aus Gründen der Nachvollziehbarkeit für die Testanwendung trotzdem unterschiedliche Ausgaben erzeugen. Zweckmäßig wäre hier zum Beispiel die Ausgabe der Prozessornummer.

- Im Laufe der Aufgaben werdet ihr Systemaufrufe benutzen müssen, für die es in der `GLIBC` keine C-Wrapperfunktion gibt. Wie man solche Systemaufrufe aus einem C/C++ heraus auslöst, findet ihr in der Manpage `syscall(2)`. Eine Übersicht über alle vorhandenen Systemaufrufe mit den entsprechenden `define`s findet ihr in `syscalls(2)`.

  Im Rahmen dieser Aufgabe betrifft das die Aufrufe von `getcpu` und `exit`.

  Da die Man Pages an dieser Stelle manchmal etwas verwirrend erscheinen, hilft oft ein Blick in die Header-Dateien weiter.

- Die Anzahl der in einem System vorhandenen Prozessoren kann man über `sysconf(_SC_NPROCESSORS_ONLN)` herausfinden.

- Die Implementierung der Aufgabe erfolgt in der Programmiersprache `C++`.

1.7 Abgabe: am 09.05.2012