Hochschule RheinMain
Studiengang Medieninformatik B. Sc.
Sommersemester 2016
Prof. Dr. Robert Kaiser

Betriebssysteme und Rechnerarchitektur

LV 4112
Übungsblatt 3
2. Juli 2016

In dieser Übung soll die μThread (Micro Thread) Bibliothek, erstellt werden, die ein kooperatives Multithreading auf Anwenderebene ermöglicht.

Auf der Webseite der Veranstaltung finden Sie dazu das Verzeichnis `uthread` als komprimierte tar-Datei. Entpacken Sie diese Datei auf Ihrem Rechner. Sie finden darin die Unterverzeichnisse `html`, `include`, `lib` und `man`, sowie die Dateien `main.c`, `Makefile` und `output.txt`.

Aufgabe 3.1 (Dokumentation):

(a) In den Verzeichnissen `man` und `html` finden Sie die Dokumentation der μThread Bibliothek. Diese wurde mit Hilfe des Tools `doxygen` aus den Headerdateien im Verzeichnis `include` automatisch generiert. Schauen Sie sich diese Headerdateien und die daraus generierte Dokumentation an. Worin liegt der Vorteil, wenn wie hier Dokumentation und Programmcode aus derselben Quelle generiert wird?

Auf diese Weise ist es zwar nicht sichergestellt, aber doch immerhin wahrscheinlicher, dass Quellcode und Dokumentation zusammen passen.

(b) Lesen und verstehen Sie die Dokumentation zur den Headerdateien. Wozu dienen die Headerdatei `jmp.h` und `uth.h` und wozu dient die C-Datei `jmp.c` im Verzeichnis `lib` ?

Die Headerdatei `jmp.h` beschreibt eine erweiterte Form des aus der Vorlesung bekannten Funktionspaares `setjmp()`/`longjmp()`. Die (Assembler-)Implementierung dieser Funktionen findet sich in der Datei `jmp.c` im Verzeichnis `lib`. Diese Funktionen sollen der μThread Bibliothek intern als Mechanismen zur Thread-Umschaltung dienen. Die Headerdatei `uth.h` beschreibt die Schnittstelle der μThread Bibliothek.

(c) Das Programm `main.c` ist ein Testprogramm für die zu erstellende μThread Bibliothek. Lesen und verstehen Sie den Quellcode. Die Datei `output.txt` enthält die von diesem Programm erzeugte Ausgabe als Referenz. Wie erklären Sie sich die Reihenfolge der darin protokollierten Ausgaben?

Offenbar handelt es sich hier um ein Programm, das mehrere Threads enthält. Jeder Thread gibt, wenn er gestartet wird, eine Meldung aus. Immer, wenn ein Thread blockiert, läuft ein anderer Thread weiter. Threads können sich offenbar selbst beenden, können aber auch von anderen Threads beendet werden.

Aufgabe 3.2 (Begriffe):

(a) Die μThread Bibliothek implementiert das so genannte *kooperative Multithreading*. Worin liegt der Unterschied zum *präemptiven* Multithreading, wie es die meisten heutigen Betriebssysteme implementieren?

Beim kooperativen Multithreading gibt es keine Möglichkeit, einem Thread die Kontrolle über den Prozessor zu entziehen, d.h. andere Prozesse kommen nur zur Ausführung, wenn der aktuelle Prozess den Prozessor freiwillig abgibt.

(b) Worin liegt der Unterschied zwischen Multitasking und Multithreading? Welche Adressbereiche werden beim Multithreading von allen Threads gemeinsam genutzt, welche sind für jeden Thread privat zu halten?

Beim Multitasking existiert jede Aktivität (jede „Task“) in ihrem eigenen Adressraum, während beim Multithreading alle Aktivitäten („Threads“) sich einen gemeinsamen Adressraum teilen. Jeder Thread verwendet einen eigenen Stack, der jedoch auch für andere Threads zugreifbar ist.

Aufgabe 3.3 (Implementierung der μThread Bibliothek):

Die Implementierung der in der Headerdatei `uth.h` beschriebenen Funktionen soll nun in die z.T. leere Datei `uth.c` im Verzeichnis `lib` eingefügt werden.

(a) Überlegen Sie sich eine oder mehrere Datenstrukturen, in der/denen die Zustände der von der μThread Bibliothek verwalteten Threads abgespeichert werden können. Welche Daten repräsentieren den vollständigen Zustand eines Threads?

Ein Thread wird repräsentiert durch seinen Stack und seinen Prozessorzustand. Letzterer wird hier als `jmpbuf` Struktur dargestellt. Darüber hinaus sind in der Regel noch Verwaltungsdaten erforderlich (in dieser Implementierung ist das nur das „busy-“ Flag). Alle drei genannten Teile werden bei dieser Implementierung in einer Datenstruktur namens TCB (*Thread Control Block*) zusammengefasst.

(b) Die Stacks der Threads sollten vorzugsweise im für Stack-Operationen vorgesehenen Adressbereich des umschließenden Anwenderprogrammes liegen. Wie kann die Funktion `uth_init()` einen solchen Stack-Speicherbereich reservieren?

Die Funktion deklariert ein hinreichend großes Array als automatic-Variable, das somit im Stack des Anwenderprogrammes liegt. Dieser so reservierte Speicherplatz wird unter den einzelnen Threads aufgeteilt. Der Platz bleibt während der gesamten Laufzeit des μThread-Systems erhalten, da das gesamte Thread-System als Unterprogramm von `uth_init()` ausgeführt wird.

(c) Implementieren Sie die noch fehlenden Funktionen der μThread Bibliothek in der Datei `uth.c`. Überprüfen Sie Ihr Ergebnis, indem Sie das Testprogramm `main.c` gegen Ihre Bibliothek linken und die von diesem Programm erzeugte Ausgabe mit dem Inhalt der Referenzdatei `output.txt` vergleichen.

Siehe Programm `uth.c` im Verzeichnis uebung_4/4.3c

Aufgabe 3.4 (Erweiterungen):

(a) Die Threads des Testprogrammes `main.c` terminieren sich jeweils selbst mit Hilfe der Funktion `uth_suicide()`. Was würde geschehen, wenn das nicht der Fall wäre, d.h. wenn sie zu ihrem Aufrufer zurückkehren würden?

Bei der Rückkehr zum Aufrufer (RET-Befehl) wird, was immer gerade zuoberst auf dem Stack vorgefunden wird, in den Programmzähler geladen. Normalerweise ist das die zuvor vom Aufrufer dort hinterlegte Rücksprungadresse. Hier allerdings ist der Inhalt des Stacks an dieser Stelle typischerweise zufällig, d.h. mit sehr großer Wahrscheinlichkeit wird das Programm abstürzen.

(b) Erweitern Sie Ihre μThreads Bibliothek derart, dass ein solches Zurückkehren zum Aufrufer den Thread terminiert.

Siehe Programm `uth.c` im Verzeichnis uebung_4/4.4b: Anstelle des zufälligen Werts auf dem Stack wird hier ein Zeiger auf die interne Funktion `suicide()` hinterlegt, sodass der Thread sich selbst terminiert.

(c) Überlegen Sie, ob bzw. wie die μThread Bibliothek ggf. für präemptives Multithreading erweitert werden könnte.

Im Prinzip wäre das möglich. Man müsste dafür sorgen, dass die Routine `uth_next()` periodisch in festen Zeitabständen von zum Beispiel 10 Millisekunden aufgerufen wird. Dies wäre zum Beispiel mit einem periodischen Timer (Interrupt) möglich.