DHBW
Duale Hochschule
Baden-Württemberg
Ravensburg
Campus Friedrichshafen

Programmentwurf

Entwicklung eines rudimentären Zeitscheibensystems in C

Studiengang: Embedded Systems

Studienrichtung: Automotive Engineering

Duale Hochschule Baden-Württemberg Ravensburg, Campus Friedrichshafen

von

Fabian Klotz

Abgabedatum:	09.04.2023
Kurs:	TSA22
Studiengangsleiter:	Prof. Dr. Ing. Florian Leitner-Fischer

Erklärung

gemäß Ziffer 1.1.13 der Anlage 1 zu §§ 3, 4 und 5 der Studien- und Prüfungsordnung für die Bachelorstudiengänge im Studienbereich Technik der Dualen Hochschule Baden-Württemberg vom 29.09.2017 in der Fassung vom 25.07.2018.

Ich versichere hiermit, dass ich meinen Programmentwurf mit dem Thema:

Entwicklung eines rudimentären Zeitscheibensystems in C

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Weingarten, den 27. März 2023

_____

Fabian Klotz

Kurzfassung

Im Rahmen des Programmentwurfs der Vorlesung Programmieren soll ein rudimentäres Zeitscheibensystem entworfen werden. Hierbei sind einige Anforderungen zu erfüllen, wie zum Beispiel eine sehr große Variabilität des Programms, sodass mit möglichst wenig Aufwand neue Tasks angelegt, oder zu anderen Zykluszeiten umgesetzt werden können.

Das vorliegende Programm basiert hierbei auf dem Konzept des Threadings um die zeitliche Abfolge zu organisieren. In den einzelnen Tasks können mathematische Funktionen mit einer bestimmten Priorität, also Vorgabe der auszuführenden Reihenfolge, konfiguriert werden. Außerdem können sich die einzelnen Funktionen selbst für die Ausführung in einem Task mit einer gewissen Priorität registrieren. Braucht eine Funktion länger für die Ausführung wie die angegebene Zykluszeit, wird der Task erst dann neu gestartet, wenn er regulär wieder an der Reihe ist.

Abschließend wird der Worst-Case Jitter der einzelnen Tasks berechnet.

Inhaltsverzeichnis

1 Konzeptentwurf

Um ein passendes Konzept für die Umsetzung des Programms zu finden, wurden im Voraus einige Überlegungen, vor allem zur Steuerung der zeitlichen Abfolge angestellt. Mit dem Konzept des Threadings kann sowohl die zeitliche Abfolge, als auch der Aufruf der einzelnen Funktionen in den Tasks effizient gesteuert werden.

Die Abbildung soll Threading bildhaft darstellen[1] .

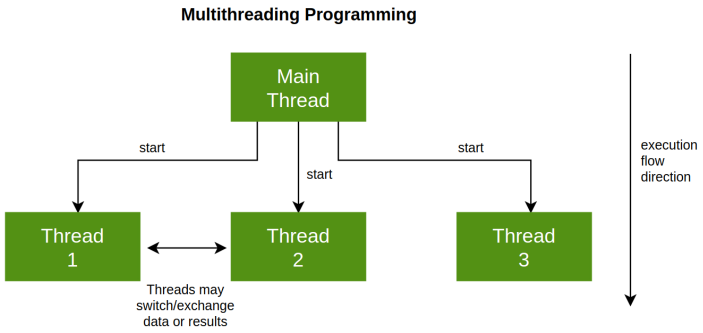

Abbildung 1.1: Threading

Der „Main Thread“ startet mehrere untergeordnete Threads. Alle Threads greifen auf die gleichen Ressourcen und globalen Variablen zu. Da alle Threads gleichzeitig ausgeführt werden, werden auch die Aufgaben in den Threads gleichzeitig ausgeführt. Im Programm werden in den Threads jeweils die einzelnen konfigurierten Funktionen aufgerufen und abgearbeitet. Anschließend wird leere Schleife so lange ausgeführt, bis die vorgegebene Zykluszeit abgelaufen ist. Anschließend wird der Task unverzüglich neu gestartet, da er sich in einer Endlosschleife befindet.

[1] https://www.baeldung.com/wp-content/uploads/sites/4/2020/07/multithreading.png

Die Konfiguration der einzelnen Funktionen (es handelt sich in diesem Programm um beispielhafte mathematische Operationen) soll über eine Art „Konfigurationstabelle“ geregelt werden. Im Programm wird ein 2D-Array erstellt, das den Tasks die Funktionen zuweist.

Für die Selbstregistrierung von Funktionen in einem Task, wird eine weitere Funktion aufgerufen, die das Konfigurationsarray so manipuliert, dass die angegebene Funktion im jeweiligen Task mit einer bestimmten Priorität ausgeführt wird. Generell ist der Aufbau des Programms variabel gestaltet. So können zum Beispiel beliebig viele Threads (also Tasks) oder Funktionen erstellt werden.

2 Umsetzung

2.1 pthread

Um erste Versuche mit Threading zu starten, wird ein Programm erstellt, das zwei Threads startet. Dabei wurde die Bibliothek 'pthread.h' verwendet.

`pthread` steht für ' POSIX Threads' und ist eine API (Application Programming Interface), die es ermöglicht, in C und C++ Threads zu erstellen und zu verwalten. POSIX steht dabei für "Portable Operating System Interface", was bedeutet, dass diese API auf vielen verschiedenen Betriebssystemen verfügbar ist und somit portabel ist. Pthreads werden unter UNIX-basierten Betriebssystemen wie MacOS oder Linux unterstützt.

Die `pthread`-API stellt eine Reihe von Funktionen bereit, mit denen Threads erstellt, gesteuert und synchronisiert werden können. Einige wichtige Funktionen der `pthread`-API sind:

- `pthread_create`: Diese Funktion erstellt einen neuen Thread und führt eine bestimmte Funktion aus, die dem Thread als Argument übergeben wird.

- `pthread_join`: Diese Funktion blockiert den aufrufenden Thread, bis der angegebene Thread beendet wurde.

Die `pthread`-API ist sehr leistungsfähig und wird häufig in Anwendungen eingesetzt, die viele gleichzeitig ausgeführte Threads erfordern, wie beispielsweise Serveranwendungen

oder Multimedia-Anwendungen[2].

Im fertigen Programm wird automatisch die richtige Anzahl an Threads erstellt. Die untenstehende for-Schleife wird so oft wiederholt, wie es Tasks im Programm geben soll. In der Schleife wird jeweils 200ms gewartet, damit die Threads leicht zeitversetzt laufen, um später die Ausgabe auf der Konsole lesbarer zu machen.

```
pthread_t threads[num_threads];
int thread_args[num_threads];

for (int i = 0; i < num_threads; i++) {
  thread_args[i] = i;
  pthread_create(&threads[i], NULL, thread_function, &thread_args[i]);
  wait_200ms();
}
```

Beim erstellen eines Threads wird immer die `thread_function` aufgerufen. Hier wird das Grundgerüst der einzelnen Threads festgelegt. Es wird eine `while(1)`, also eine Endlosschleife angelegt. In dieser Schleife wird zuerst ein Timer gestartet und danach mit einer weiteren Funktion überprüft, ob im Konfigurationsarray eine Funktion für diesen Task hinterlegt ist. Sollte das der Fall sein, werden die Aufgaben je nach Priorität abgearbeitet. Laufen die auszuführenden, konfigurierten Funktionen zu lange, also länger wie die vorgegebene Zykluszeit, wird so lange gewartet, bis der Task wieder regulär an der Reihe ist.

[2]https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html

2.2 Konfiguration der Funktionen

2.2.1 Konfigurationsarray

Das Konfigurationsarray um die Funktionen den Tasks zuzuordnen ist folgendermaßen aufgebaut:

```
double array[NUM_FUNC][NUM_TASKS] = {
    //          T1  T2  T3  T4
    /*add*/     {3,  0,  0,  0},
    /*multi*/   {2,  2,  0,  0},
    /*div*/     {1,  1,  1,  0},
    /*mod*/     {0,  0,  0,  1},
    /*div2*/    {0,  3,  0,  0}
};
```

Wie an den Kommentaren zu sehen ist, geben die Zeilen immer die Funktionen an, welche in den Tasks auf gerufen werden sollen und die Spalten die dazugehörigen Tasks. Die Zahlen stehen hierbei für die Priorität, also die Reihenfolge. (1 = höchste Priorität, 0 = nicht Konfiguriert)

2.2.2 Selbstregistrierung von Funktionen zur Ausführung in einem Task

Damit wie gefordert eine Selbstregistrierung der Funktionen zur Ausführung in einem Task (bzw. Thread) ausgeführt werden kann, wird beim Aufruf der mathematischen Funktionen unter anderem ein Übergabewert übermittelt, der dazu führt, dass über eine if-Bedingung eine Funktion `registration()` aufgerufen wird. Als Übergabewert erwartet diese Funktion den Task, die auszuführende Funktion und die Priorität.

```
void registration(int task, int priority, int func){
        printf("\nTask %d registriert Funktion%d mit Priorität %d \n\n",
    task, func, priority);
        array[func-1][task-1] = priority;
}
```

2.2.3 Ausführung der Funktionen in den Tasks

Die Funktion `checkForTasks` läuft folgendermaßen ab:

```
void checkForTasks(int thread, double array[][NUM_TASKS]) {
    for(int j = 0; j < NUM_FUNC; j++){
        double *p = &array[0][thread-1];
        int i;
        for (i = 0; i < NUM_FUNC; i++) {
            if (*p == j+1) {
                fktPointer[i%NUM_FUNC](a, b, 0);
                printf("\n");
            }
            p = p + NUM_TASKS;
        }
    }
}
```

Beim Aufruf der Funktion wird der Thread und das Konfigurationsarray übergeben. Eine for-Schleife wird so oft durchlaufen, wie es Funktionen für die Tasks gibt. Danach wird ein Pointer auf das Array erstellt, der auf den ersten, für den Thread relevanten Eintrag zeigt. Es wird eine zweite for-Schleife erstellt, die ebenso so oft durchlaufen wird, wie es relevante Funktionen für die Tasks gibt. Danach wird überprüft, ob der Pointer auf ein Element zeigt, das der Schleifenvariablen der äußeren for-Schleife + 1 entspricht. Sollte das der Fall sein, wird ein Pointerarray aufgerufen, in dem Funktionspointer für die jeweiligen Funktionen hinterlegt sind. Durch die Rest-berechnung, innere Schleifenvariable modulo Anzahl der Funktionen wird die richtige Stelle im Array ermittelt und ausgeführt.

3 Bedienung

3.1 Kompilierung

Zur Kompilierung des vorliegenden Programms muss beachtet werden, dass pthreads nur auf UNIX-basierten Betriebssystemen verfügbar ist. Getestet wurde die Kompilierung ausschließlich in folgendem Docker Container:
`leitnerfischerdhbw/es-ubuntu-x86`

Pthread muss manuell für das Kompilieren hinzugefügt werden. Der Befehl für die Kompilierung lautet als:
`gcc -pthread 'Dateiname'`

3.2 Änderung der Anzahl auszuführender Tasks

Um die Anzahl der Tasks zu ändern muss zuerst folgende Zeile geändert werden und auf die gewünschte Anzahl erhöht oder erniedrigt werden:
`define NUM_TASKS 4`
Danach muss die Zykluszeit festgelegt werden. Hierfür wird in diesem Array ein Element mit der gewünschten Zeit hinzugefügt werden (Zeit in ms):
`int cycleTimes[NUM_TASKS] = {1000, 5000, 10000, 100000};`
Zusätzlich sollte im Konfigurationsarray eine Spalte mit den gewünschten Funktionen und Priorität zur Ausführung in diesem Task hinzugefügt werden. Sollte das Array nicht

erweitert werden, kann es zu unschönen Ergebnissen führen, da die nicht initialisierten Elemente nicht automatisch '0' sind.

3.3 Änderung der Zykluszeiten der Tasks

Um die Die Zykluszeiten zu ändern muss lediglich in diesem Array die gewünschte Zeit geändert werden (Zeit in ms).
`int cycleTimes[NUM_TASKS] = {1000, 5000, 10000, 100000};`

3.4 Hinzufügen von Funktionen

Um eine Funktion hinzuzufügen muss im ersten Schritt folgender Wert erhöht bzw. erniedrigt werden:
`define NUM_FUNC 5`
Anschließend wird die neue Funktion analog zu den bestehenden Funktionen angelegt. Die Übergabewerte sind hierbei die double Werte a und b und eine int Variable um eine potentielle Selbstregistrierung zu ermöglichen. Im Array `double (*fktPointer[NUM_FUNC])(double, double, double)` muss außerdem ein neues Element angelegt werden, das gleich zu benennen ist, wie die erstellte Funktion Abschließend muss das Konifgurationsarray um eine (oder mehr) Zeilen erweitert und initialisiert werden.

4 Jitter

Obwohl Multithreading ein sehr Effizientes Konzept der Informatik ist, kann es sein, dass die Systemauslastung so hoch ist, dass die Threads nicht gleichzeitig ausgeführt werden können. Das kann verschiedene Gründe, wie zum Beispiel der Anzahl der verfügbaren Prozessorkerne, die Auslastung des Arbeitsspeichers etc., haben. Im schlimmsten Fall, sind so wenige Systemressourcen vorhanden, dass alle Threads nacheinander ausgeführt werden müssen. In diesem Fall würde sich der Jitter folgendermaßen berechnen:

$\sum Zykluszeiten$

In diesem Fall ist der Worst-Case-Jitter also 135 Sekunden. Um die Behauptung zu validieren, dass der Jitter von der Verfügbarkeit von Systemressourcen abhängt, wurde während der Ausführung des Programms ein CPU-Benchmark gestartet. Tatsächlich kannte beobachtet werden, dass die Ausführungszeit der einzelnen Threads teilweise deutlich erhöht wurde. Getestet wurde auf einem MacOS System mit Apple M1 und 16gb RAM. Die maximal zu beobachtende Abweichung ist 99ms.

5 Tests

Um das vorliegende Programm zu testen wird dieses als erstes Kompiliert, um sicherzugehen, dass keine Kompilerwahrnungen oder Errors auftreten. Ein weiterer Testfall ist das Einfügen eines `sleep` in eine Funktion, das eine aufgerufene Funktion so lange blockiert, dass die eigentliche Zykluszeit abgelaufen ist. Das Programm reagiert wie erwartet und gibt ein Fehler auf der Konsole aus, dass die Funktion zu lange für die Ausführung benötigt hat und wird erst wieder neu gestartet, wenn der nächste reguläre Zyklus ansteht. Außerdem wird die Erweiterbarkeit der Tasks und Funktionen überprüft. Die Selbstregistrierung von Funktionen wird getestet und funktioniert ordnungsgemäß. Beim Starten des Programms wird auf der Konsole eine Tabelle ausgegeben, welche genau zeigt, welche Funktionen in welchem Task konfiguriert sind.

6 Quellen und Hilfsmittel

Für die Umsetzung der Anforderungen in den Quellcode wurden unter anderem folgende Hilfsmittel verwendet: "Fit fürs Studium Informatik Rheinwek Computing - Bockmeyer|Fischbeck|Neubert

Vorlesungsskripte aus der Vorlesung 'Programmieren' von Herr Prof. Dr. Ing. Florian Leitner-Fischer

`https://hpc-tutorials.llnl.gov/posix/`

`chat.openai.com`