Praktikumsaufgabe: Multithreading

Prof. Dr. Ralf Gerlich
Hochschule Furtwangen
Fakultät Computer Science & Applications
Robert-Gerwig-Platz 1
D-78120 Furtwangen
ralf.gerlich@hs-furtwangen.de

30. Mai 2025

Zusammenfassung

In diesem Praktikum steigen Sie in das Multithreading mit `pthreads` (Kurzform für „POSIX Threads“) ein. Sie lernen, wie Sie Threads erstellen und auf ihre Beendigung warten, wie Sie den gegenseitigen Ausschluss („mutual exclusion“) umsetzen, wie Sie Condition Variables zur Signalisierung einsetzen und wie Sie das „Dining Philosophers“-Problem lösen.

Inhaltsverzeichnis

1 Einleitung

POSIX Threads – kurz `pthreads` – ist ein Ausführungsmodell für Multithreading, d.h. für die Ausführung von mehreren Threads im selben Prozess mit geteiltem Adressraum. POSIX ist das Portable Operating System Interface,

eine Familie von Standards, die durch die Computer Society des Institute of Electrical and Electronics Engineers (kurz: IEEE) definiert wurde.

Der POSIX-Standard wird vorwiegend von Betriebssystemen der UNIX-Familie wie Linux, den Varianten der Berkeley Software Distribution (FreeBSD, OpenBSD oder NetBSD) sowie macOS, Android und einigen weiteren implementiert. Diese bieten somit eine weitgehend standardisierte Programmierschnittstelle. Da Windows den POSIX-Standard nicht bzw. nicht vollständig unterstützt, empfiehlt sich für die Verwendung von `pthreads` unter Windows der Einsatz eines Systems wie MinGW oder MSYS.

Sowohl für CLion als auch für VSCode und das für Windows benötigte Zusatzpaket MSYS finden Sie Installationsanleitungen auf der FELIX-Seite. Alternativ können Sie die Linux-VMs für die Veranstaltung in den Rechnerpools der Fakultät I (C2.03/C2.04) verwenden.

2 Beispiel: Threads erstellen und beenden

Das Softwarepaket für diese Übungsaufgabe enthält mehrere Code-Dateien für verschiedene Programme.

Betrachten Sie zunächst das Programm `examples/thread_creation.c`. Hier werden zwei Threads gestartet, die jeweils 200 mal ihre eigene Bezeichnung („Thread 1“ oder „Thread 2“) auf die Konsole ausgeben.

Übersetzen und starten Sie das Target `thread_creation` und beobachten Sie die Ausgabe. Sie sollten erkennen, dass stellenweise die Bezeichnungen beider Threads vermischt sind. Die Threads werden also quasigleichzeitig ausgeführt.

Um einen Thread zu erstellen, müssen Sie die Funktion `pthread_create` aufrufen. Diese ist – wie alle anderen Teile von `pthread` – in der Include-Datei `<pthread.h>` deklariert.

Die Funktion `pthread_create` erwartet vier Parameter:

- einen Zeiger auf eine Variable vom Typ `pthread_t`. In diese wird der sogenannte *Thread Identifier* geschrieben, anhand dessen der so erstellte Thread eindeutig identifiziert werden kann. Der Großteil der weiteren `pthread`-Funktionen benötigt diesen Identifier, um den zu bearbeitenden Thread zu bestimmen.

- einen Zeiger auf eine Struktur mit Attributen für den zu erstellenden Thread. Hier können weitere Eigenschaften des Threads festgelegt werden, wie z.B. die Thread-Priorität. Hier kann ein Null-Zeiger (`NULL`) angegeben werden, um die Standardeinstellungen zu verwenden. Für das Praktikum ist dies vollkommen ausreichend.

- ein Zeiger auf die *Hauptfunktion* des Threads. Diese Funktion erhält als einziges Argument einen `void`-Zeiger (Typ `void*`) und kann einen ebensolchen zurückgeben.

- den Wert für das einzige Argument der Hauptfunktion. Hierüber können Daten übergeben werden, die durch den Thread verarbeitet werden sollen. Dies ist insbesondere dann wichtig, wenn mehrere Threads mit derselben Hauptfunktion erstellt werden sollen.

Jeder Thread erhält implizit Platz für seinen eigenen Stack. Allerdings teilen sich die Threads allesamt denselben Adressraum, können also zum Beispiel gemeinsam auf globale Variablen zugreifen.

Mit der Funktion `pthread_join` kann auf die Beendigung eines bestimmten Threads gewartet werden. Ihr wird die Thread-ID des Threads übergeben, auf den gewartet wird, sowie ein Zeiger auf `void*`, der angibt, wo der Rückgabewert des Threads abgelegt werden soll. Wenn der Rückgabewert nicht benötigt wird, kann an dieser Stelle auch `NULL` angegeben werden.

Threads können auch zwangsweise mit `pthread_cancel` beendet werden. In jedem Fall werden alle Threads abgebrochen, wenn das Programm beendet wird. Dieses Verhalten kann mit der Funktion `pthread_detach` geändert werden. Ein „detached thread“ läuft auch weiter, wenn das Hauptprogramm beendet wurde. Weder `pthread_cancel` noch `pthread_detach` wird

2.1 Fehlerrückmeldungen in `pthread`

Die Funktionen der `pthread`-Library liefern fast alle einen Wert vom Typ `int` zurück, der `0` ist, wenn die Operation erfolgreich ausgeführt wurde. Ist der Rückgabewert nicht `0`, so handelt es sich um einen der POSIX-Fehlercodes. In den Manual Pages findet man eine entsprechende Sektion **ERRORS**, unter der die Fehlercodes für die Funktion aufgelistet und erläutert werden.

Die dort aufgeführten Namen sind in der Include-Datei `<errno.h>` als Makros mit Ganzzahlwerten deklariert. Man kann also den Rückgabewert direkt mit den Makros dort vergleichen. So steht der Rückgabewert `EPERM` für „Permission denied“ – Ihr Prozess bzw. Nutzer besitzt also nicht genügend Rechte, um die Operation auszuführen.

Einen etwas nutzerfreundlicheren String mit einer Beschreibung liefert die Funktion `strerror`, die in der Header-Datei `<string.h>` deklariert ist. Ein Beispiel für deren Anwendung finden Sie im Beispielcode.

2.2 Informationen zur `pthread`-Library

Übrigens: Informationen über die hier genannten Funktionen erhalten Sie unter Linux über `man`. So zeigt der Befehl `man pthread_create` die sogenannte Manual Page für die Funktion `pthread_create` an.

Alternativ können Sie im Internet das Linux Man Page Projekt verwenden: `https://www.man7.org/linux/man-pages/man7/pthreads.7.html` Auch bei der Single UNIX® Specification finden sich Informationen zur Bibliothek: `https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html`

3 Beispiel: Gegenseitiger Ausschluss

Die `pthreads`-Bibliothek unterstützt auch die Absicherung von gemeinsam genutzten Ressourcen über sogenannte *Mutexes* (von engl. „mututal exclusion“ – „gegenseitiger Ausschluss“). Im Beispiel `examples/mutex.c` sehen Sie hierfür eine Anwendung.

Im Code werden zwei Threads mit derselben Hauptfunktion angelegt, die jeweils einen Zähler in der globalen Variablen `counter` erhöhen – und zwar je-

weils 100̇000 Mal. Am Ende wird der Wert von `counter` ausgegeben und müsste nun natürlich den Wert 200̇000 haben.

Am Anfang des Codes finden Sie das Makro `USE_MUTEX`. Setzen Sie dessen Wert einmal auf `0`, übersetzen Sie das Programm und führen Sie es ein paar Mal aus.

Sie sollten beobachten können, dass der Endwert häufig kleiner als 200̇000 ist. Dies liegt daran, dass das Erhöhen des Werts einer Variablen keine *atomare* Operation ist. Sie besteht aus mehreren Assemblerinstruktionen, und nach jeder Instruktion kann der Thread – ähnlich wie bei einem Interrupt – unterbrochen werden.

Somit kann es zu einem *Lost Update* kommen: Der neue Wert wird erst geschrieben, wenn der Thread weiter ausgeführt wird. Änderungen an der Variablen, die zwischenzeitlich durch den anderen Thread erfolgt sind, werden dabei überschrieben.

Setzen Sie nun `USE_MUTEX` auf einen Wert ungleich `0` (zum Beispiel `1`), übersetzen Sie das Programm erneut und führen Sie es ein paar Mal aus. Nun wird die Variable `counter` den Wert 200̇000 jedes Mal erreichen.

Der Zugriff auf die Variable ist durch ein Mutex abgesichert:

- Vor dem Zugriff wird das Mutex mit Hilfe von `pthread_mutex_lock` gesperrt. Zwar kann die Ausführung des Threads nun immer noch unterbrochen werden. Der andere Thread wird aber ebenfalls versuchen, das Mutex zu sperren. Wenn es bereits gesperrt ist, wartet der andere Thread nun, bis das Mutex wieder freigegeben ist. In der Zwischenzeit kann der erste Thread weiter ausgeführt werden und seine Operation ungestört ausführen.

- Nach dem Zugriff wird das Mutex mit Hilfe von `pthread_mutex_unlock` wieder freigegeben. Durch die Freigabe können nun auch alle Threads weiterlaufen, die am Mutex gewartet haben.

Beide Funktionen erhalten als Parameter einen Zeiger auf das zu verwendende Mutex. Dieses ist als globale Variable vom Typ `pthread_mutex_t` deklariert:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

`PTHREAD_MUTEX_INITIALIZER` initialisiert den Mutex mit Standardeinstellungen. Werden andere Einstellungen benötigt, muss der Mutex stattdessen durch einem Aufruf von `pthread_mutex_init` initialisiert werden. Im Praktikum ist dies aber nicht notwendig.

4 Aufgabe: Producer Consumer

Manche Aufgaben sind I/O-intensiv, erfordern also ständige Zugriffe auf Hardware – z.B. die Festplatte. Während die Hardware beschäftigt ist, müssen die Berechnungen pausieren. In dieser Zeit können die Prozessorkerne stattdessen für andere Aufgaben verwendet werden.

Hingegen sind manch andere Aufgaben CPU-intensiv, d.h. es wird ständig gerechnet. Ein einzelner Prozess oder Thread kann dabei nur einen CPU-Kern auslasten. Stehen mehrere CPU-Kerne zur Verfügung, ist es deshalb sinnvoll, die Aufgabe auf mehrere Threads oder Prozess aufzuteilen. So kann jeder Thread

eine Teilaufgabe erfüllen, oder bei vielen gleichförmigen Aufgaben jeder Thread eine dieser Aufgaben abarbeiten.

In diesem Fall steht man häufig vor dem Producer-Consumer-Problem: Ein oder mehrere *Producer* produzieren Ergebnisse für ein oder mehrere *Consumer*. Ein Producer kann z.B. eine Berechnung durchführen und der Consumer sammelt diese dann von den Producern ein. Alternativ kann ein Producer auch Aufgabenbeschreibungen erstellen und die Consumer holen sich jeweils eine Aufgabenbeschreibung und führen dann die erforderlichen Berechnungen verteilt durch. Ist ein Consumer mit einer Berechnung fertig, so holt er sich die neue Aufgabe vom Producer.

Darüber hinaus können diese Konzepte auch miteinander kombiniert werden: In einer Renderfarm etwa können spezialisierte Producer einzelne Bilder eines computergenerierten Films erzeugen. Mehrere Consumer nehmen ihnen diese Bilder ab und führen parallelisiert weitere Bearbeitungen durch – zum Beispiel eine Farbkorrektur. So sind Prozesse Producer und Consumer zugleich.

Die Liste der Ergebnisse ist dabei eine gemeinsam genutzte Ressource, und das Einstellen oder Entfernen eines Eintrags ist kein atomarer Vorgang:

- Der zu entfernende Eintrag muss aus der Liste kopiert werden.
- Der Zeiger auf den nächsten zu entfernenden Eintrag muss auf dessen Nachfolger verschoben werden.
- Die Anzahl der Einträge in der Liste muss aktualisiert werden.

Damit der Zustand der Liste – vorhandene Einträge, Zeiger auf den nächsten Eintrag, Anzahl der Einträge – konsistent bleibt, müssen diese Aktionen aber zusammen ausgeführt werden. Es muss also ausgeschlossen werden, dass zwischen diesen Schritten ein anderer Prozess Änderungen an der Liste vornehmen kann.

4.1 Der Beispielcode

Unter `exercises/producer_consumer.c` finden Sie eine naive Implementierung des Producer-Consumer-Problems. Es wird eine Anzahl von Producer-Threads erstellt, die ihre Ergebnisse in einen Ringpuffer `buffer` schreiben. Zum Ringpuffer gehören der Lesezeiger `read_index`, der Schreibzeiger `write_index` und der Füllstand des Puffers `buffer_level`.

Da der Puffer nur Platz für `BUFFER_SIZE` Einträge bietet, müssen die Producer bei einem vollen Puffer ggf. warten, bis wieder Platz im Puffer ist. Dazu fragen Sie den Füllstand `buffer_level` in einer Schleife ab. Ist der Füllstand unter das Maximum gefallen, tragen Sie ihr neues Berechnungsergebnis ein und aktualisieren den Füllstand und den Schreibzähler.

Der Consumer muss seinerseits darauf warten, dass im Puffer mindestens ein Element steht. Dazu wartet er in einer Schleife, bis `buffer_level` größer als Null geworden ist. Dann kopiert er das vorderste Element aus dem Puffer und aktualisiert den Lesezähler `read_index` und den Füllstand `buffer_level`.

Die Producer sollen in diesem Beispiel gemeinschaftlich die Zahlen von `0` bis `PRODUCED_ITEMS-1` erzeugen. Der dafür verwendete Zähler `total_count` ist in der Code-Vorlage bereits über den Mutex `counter_mutex` abgesichert. In der Realität wäre die Aufgabe natürlich komplexer.

4.2 Ihre Aufgabe

Im Beispielcode sind keine Absicherungen gegen Race Conditions enthalten. Wenn Sie den Code ausführen, kann es dennoch sein, dass alles korrekt abläuft. Sicher ist der Code so aber nicht.

Führen Sie einen Mutex für die Absicherung des Puffers (`buffer`, `read_index`, `write_index` und `buffer_level`) ein und fügen Sie dem Code an den entsprechenden Stellen die erforderlichen Aufrufe von `pthread_mutex_lock` und `pthread_mutex_unlock` hinzu!

5 Aufgabe: Condition Variables

In der vorigen Aufgabe warten die Producer und Consumer jeweils *aktiv* darauf, dass der Puffer einen freien bzw. belegten Eintrag hat. Dabei wird kontinuierlich Prozessorzeit verbraucht. Besser wäre es, wenn ein Producer die wartenden Consumer benachrichtigt, wenn im Puffer wieder ein Element verfügbar ist, bzw. die Consumer die Producer benachrichten, wenn im Puffer wieder ein freier Platz vorhanden ist. Bis diese Benachrichtigung vorhanden ist, sollen die jeweiligen Threads passiv warten, d.h. sie sollen die CPU nicht nutzen.

Hierfür bietet `pthread` sogenannte *Condition Variables* an. Eine Condition Variable repräsentiert eine Warteschlange, in der sich Threads „anstellen“, um auf das Eintreffen eines Ereignisses zu warten. Während sie anstehen, sind sie im Wartezustand, d.h. sie verbrauchen keine CPU-Zeit.

Ein anderer Thread kann nun einen oder alle Threads aus der Warteschlange entlassen, wenn eine seiner Aktionen dazu führt, dass das Ereignis eintritt bzw. eingetreten sein könnte. Die nun wieder aktiven Threads müssen aber die Erfüllung der Bedingung selbstständig erneut prüfen.

Auf unser Producer-Consumer-Problem übertragen könnte man nun zwei Condition Variables anlegen:

- Eine Condition Variable `cond_items_available` für die Consumer, die auf neue Einträge in der Liste warten.
- Eine Condition Variable `cond_places_availabel` für die Consumer, die auf freie Plätze in der Liste warten.

Betrachten wir dies am Beispiel von `cond_items_available`:

- Ein Consumer wird zu Beginn seiner Hauptschleife prüfen, ob mindestens ein Eintrag vorhanden ist. Wenn ja, wird er diesen aus der Liste entfernen und wieder von vorne anfangen. Ist kein Eintrag vorhanden, so wird er sich mit Hilfe von `pthread_cond_wait` in die Warteschlange `cond_items_available` einreihen. Wenn `pthread_cond_wait` zurückkehrt, muss der Consumer erneut auf einen verfügbaren Eintrag prüfen – und die Warteschleife nun also erneut von vorne beginnen.
- Ein Producer wird in jedem Durchlauf seiner Hauptschleife einen neuen Eintrag generieren. Danach wird er mit `pthread_cond_signal` den ersten Thread in der Warteschlange von `cond_items_available` aufwecken, und dann seine Hauptschleife fortsetzen, um den nächsten Eintrag zu produzieren.

Alternativ kann der Producer übrigens auch `pthread_cond_broadcast` verwenden, um alle Consumer in der Warteschlange aufzuwecken. Dies ist zum Beispiel dann sinnvoll, wenn die Consumer nicht ihrer Reihenfolge in der Warteschlange entsprechend auf die Einträge im Puffer zugreifen sollen, sondern anhand ihrer Priorität.

Als Besonderheit ist zu beachten, dass `pthread_cond_wait` zwei Parameter erhält:

1. Den Zeiger auf die Condition-Variable, anhand der gewartet werden soll.
2. Einen Zeiger auf einen Mutex.

Die Funktion wird atomar – d.h. nicht durch andere Threads unterbrechbar – zunächst den Mutex freigeben (ähnlich dem Aufruf von `pthread_mutex_unlock`), den aktuellen Thread in die Warteschlange der angegebenen Condition-Variable eintragen und dann den aktuellen Thread in den Wartezustand versetzen.

Dies ist sinnvoll, da dem Aufruf von `pthread_cond_wait` ja eine Prüfung einer oder mehrerer Variablen vorangegangen sein wird (z.B. `buffer_level`), die durch einen Mutex geschützt sind. Würde man den Mutex vor dem Aufruf an `pthread_cond_wait` freigeben, könnte der aktuelle Thread durch einen anderen Thread unterbrochen werden, der auf den Mutex gewartet hat, und der dann zwischenzeitlich einen Eintrag in den Puffer schreibt. In dem Fall wäre das Warten unnötig: Es ist ja ein Eintrag vorhanden.

Nach der Rückkehr von `pthread_cond_wait` ist der Mutex dann wieder durch den aufrufenden Thread gesperrt. Auch dies geschieht atomar. Die Gründe hierfür sind dieselben wie beim atomaren Entsperren des Mutex bei Eintritt in `pthread_cond_wait`.

Achten Sie also darauf, beim Einfügen von `pthread_cond_wait` eventuell zugehörige Aufrufe von `pthread_mutex_unlock` und `pthread_mutex_lock` innerhalb der Warteschleife zu entfernen, und den entsprechenden Mutex stattdessen an `pthread_cond_wait` zu übergeben!

Condition Variables haben den Typ `pthread_cond_t`. Wie bei Mutexes werden Sie einfach als Variablen deklariert:

```
pthread_cond_t cond_variable = PTHREAD_COND_INITIALIZER;
```

`PTHREAD_COND_INITIALIZER` initialisiert die Condition Variable dabei mit Standardwerten. Wenn davon abweichende Einstellungen erforderlich sind, können Sie die Funktion `pthread_cond_init` verwenden. Für das Praktikum ist dies aber nicht notwendig!

5.1 Ihre Aufgabe

Ändern Sie Ihre Implementierung von `producer_consumer.c` so, dass statt des aktiven Wartens auf neue Einträge oder freie Plätze im Puffer passiv mit Hilfe einer entsprechenden Condition-Variable gewartet wird!

6 Aufgabe: Philosophenproblem

Bei der Verwendung von Mutexes und ähnlichen Mechanismen zum gegenseitigen Ausschluss kann es auch zu Verklemmungen kommen. Eines der bekanntesten Beispiele hierfür ist das *Philosophenproblem*:

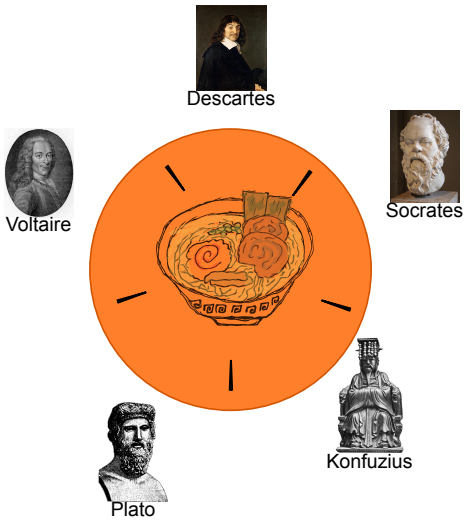

Abbildung 1: Das Philosophenproblem

> Mehrere Philosophen wohnen gemeinsam in einem Haus. Sie verbringen den Tag damit, abwechselnd zu philosophieren und zu essen. Da sie nichts anderes zu tun haben, sitzen sie einfach durchgehend um einen kreisrunden Esstisch.
>
> Das Essen steht in einer Schüssel mitten auf dem Tisch. Jeder der Philosophen kann jederzeit darauf zugreifen, ohne von einem gleichzeitigen Zugriff anderer Philosophen beeinträchtigt zu werden.
>
> Es gibt Nudelsuppe, und die Philosophen essen mit Stäbchen. Da jedoch die Philosophen nicht so viel finanzielle Aufmerksamkeit seitens des Staats erhalten wie die wohl von den Politikerinnen und Politikern als wichtiger angesehen technischen Disziplinen, sind im Haushalt nur so viele Stäbchen vorhanden wie auch Philosophen darin leben.
>
> Die Stäbchen liegen zwischen den Philosophen. Somit liegt jeweils links und rechts neben jedem Philosophen ein Stäbchen. Dieses müssen sie sich jedoch mit ihrem rechten und linken Sitznachbarn teilen.

Das Problem ist in Abb. 1 bildlich dargestellt. Wir erkennen: Es geht um ein Problem mit gemeinsam genutzten Ressourcen. Die Philosophen stellen Threads oder Prozesse dar, die Stäbchen sind gemeinsame Ressourcen, und das Nehmen eines Utensils entspricht dem Sperren des dafür benötigten Mutex.

Sie finden eine naive Implementierung des Philosophenhaushalts im Beispielcode unter `examples/dining_philosophers_buggy.c`. Übersetzen Sie das Programm und führen Sie es aus. Sie werden beobachten, dass schon nach kurzer Zeit alle Threads anhalten. Gemeinsam ist ihnen, dass sie gerade ein Stäbchen aufnehmen wollen – also ein Mutex sperren wollen.

Woher kommt diese Verklemmung? Nehmen wir einmal an, Plato hat lange gedacht und bekommt nun Hunger. Er nimmt also nun das Stäbchen zu seiner Linken. Als er jedoch versucht, das Stäbchen auf seiner rechten Seite zu nehmen,

muss er feststellen, dass Voltaire wohl gerade auch dieselbe Idee hatte, und nun das begehrte Stäbchen seinerseits in seiner linken Hand hält.

Triumphierend greift nun Voltaire an seine rechte Seite – nur um festzustellen, dass auch Descartes dieselbe Idee hatte. Es stellt sich heraus, dass alle Philosophen gleichzeitig Hunger hatten, und sich jeweils das Stäbchen auf ihrer linken Seite gesichert haben. Rechts von ihnen liegt nun kein Stäbchen mehr – und so warten alle darauf, dass der jeweilige Nachbar sein linkes Stäbchen wieder hinlegt.

Das wird aber nur dann geschehen, wenn dieser Nachbar mit Essen fertig ist, und er kann erst mit dem Essen anfangen, wenn er sein rechtes Essstäbchen bekommen hat. So warten nun alle gegenseitig aufeinander und es kommt keiner zum Zuge. Sie befinden sich in einem *Deadlock*.

Leider ist das Problem nicht so einfach zu lösen. Es bleibt bestehen, wenn alle zuerst nach dem rechten und dann nach dem linken Stäbchen greifen. Und auch wenn man die Reihenfolge zufällig wählt, besteht immer noch eine nicht zu vernachlässigende Wahrscheinlichkeit, dass es doch zum Deadlock kommt.

6.1 Ein Lösungsansatz

Eine Lösung besteht darin, dass alle Philosophen zunächst nach dem Stäbchen auf ihrer linken Seite greifen, und dann *versuchen*, das Stäbchen zu ihrer Rechten zu ergattern. Ist das rechte Stäbchen nicht zu haben, müssen sie auch das linke Stäbchen wieder hinlegen und das Ganze von vorne versuchen.

Dieser Ansatz ist frei von Deadlocks. Allerdings kann es dennoch sein, dass ein oder mehrere Philosophen verhungern, denn sie könnten in einer Endlosschleife gefangen sein, solange nur ihr rechter Nachbar immer schneller ist als sie. Diesen Effekt nennt man tatsächlich „Starvation“.

6.2 Ihre Aufgabe

Im Beispiel `examples/mutex.c` haben Sie die Funktionen `pthread_mutex_lock` und `pthread_mutex_unlock` zum Sperren bzw. Freigeben eines Mutex kennengelernt. Zusätzlich gibt es die Funktion `pthread_mutex_trylock`, die versucht, einen Mutex zu sperren. Ist der Mutex frei, so funktioniert die Funktion wie `pthread_mutex_lock`.

Ist der Mutex jedoch belegt, so wartet `pthread_mutex_trylock` nicht, sondern kehrt sofort – mit einer Fehlermeldung – zum Aufrufer zurück. In diesem Fall gibt die Funktion einen Rückgabewert ungleich Null zurück.

Implementieren Sie die in Abschnitt 6.1 skizzierte Lösung. Nutzen Sie dabei die Datei `exercises/dining_philosophers.c` als Vorlage.