Technische Hochschule Ingolstadt
Fakultät Informatik

Flug- und Fahrzeuginformatik & Cybersicherheit

Einführung in die Informatik 2

Praktikum Betriebssysteme

Georg Seifert, Ulrich Margull

Sommersemester 2025

Letzte Änderung vom 12. Mai 2025

Inhaltsverzeichnis

1 Überblick

In den nachfolgenden Kapiteln werden die verschiedenen Übungen zur Vorlesung Einführung in die Informatik 2 besprochen. Zu jedem Hauptkapitel (Einführung Linux, Threads, Synchronisierung, Kooperation und Konkurrenz, Shared Memory) gibt es je ein Testat.

Neben diesen gibt es noch zwei weitere Aufgaben, die bis zum vorgegebenen Datum abgegeben werden müssen, um die entsprechenden Bonuspunkte zu erhalten.

1.1 Generelle Anmerkungen

Grundsätzlich sollen die von Ihnen erstellten Übungen/Lösungen für jeden gut verständlich geschrieben sein. Dies setzt einen ordentlichen Programmierstil voraus. Hierunter fallen neben der übersichtlichen Strukturierung des Codes auch das korrekte Einrücken und eine vernünftige Benennung der Variablen.

Diese Konventionen sind insbesondere für die Bonusaufgaben von Bedeutung, da hier neben der reinen Funktionalität auch die Nachvollziehbarkeit des Codes für den Korrektor gegeben sein muss.

1.2 Testate

Übung	**Kurzbeschreibung**	**Kapitel**
Einführung Linux	Einrichten der Linux-Umgebung und Übergabe von mehreren Parameter an eine Threadfunktion	Kapitel 2.3.1, Kapitel 2.3.2, Kapitel 2.3.3, Kapitel 2.5
Threads	Einfache Nebenläufigkeit (Collatz) wird implementiert und die Ausführungszeit der sequenziellen und parallelen Implementierung wird verglichen.	Kapitel 3.3.1, Kapitel 3.3.2, Kapitel 3.3.3, Kapitel 3.4
Synchronisierung	Eine einfache Beispielfunktion, die auf einer gemeinsamen Ressource arbeitet wird implementiert. Diese Ressource wird in einem darauffolgenden Schritt geschützt. Als weitere Übung wird die Collatz-Funktion um Synchronisierungsmechanismen erweitert.	Kapitel 4.5.1, Kapitel 4.5.2, Kapitel 4.5.3 Kapitel 4.6
Kooperation und Konkurrenz	Ein einfaches Erzeuger/Verbraucher System muss implementiert werden, das Daten basierend auf einer verketteten Liste Daten austauscht.	Kapitel 5.4.1, Kapitel 5.5
Shared Memory	Ein einfaches Erzeuger/Verbraucher System muss implementiert werden, die Daten basierend auf einer einfachen Struktur über Prozessgrenzen austauschen kann.	Kapitel 6.3, Kapitel 6.4

1.3 Bonuspunkte im Praktikum Betriebssysteme

Die folgenden Bonuspunkte werden für Zusatz-Aufgaben im Praktikum Betriebssysteme vergeben:

Kurzbeschreibung	**BP**	**Kapitel**	**Abgabe**
Für die **Implementierung eines Programms mit OMP** werden Bonuspunkte vergeben. Dazu müssen Sie das Collatz-Programm so umschreiben, dass die Parallelisierung mit OMP erfolgt. OMP-Beispiele dazu finden Sie in der Vorlesung im Abschnitt "Prozesse und Threads"; die Details müssen Sie sich selbst aneignen. Zur Abgabe müssen Sie a) das Programm dem Betreuer vorführen und b) eine kurze Beschreibung Ihres Programms und der Laufzeitmessung in Moodle abgeben (1-2 Seiten PDF).	0,5	Kapitel 8.1	12. Juni 2025
Für die **Implementierung des Spiels Schere, Stein, Papier** werden Bonuspunkte vergeben. Dabei spielen zwei Threads gleichzeitig gegeneinander. Ein dritter Schiedsrichter-Thread startet jeweils eine neue Runde und ermittelt den Gewinner. Hat ein Thread drei Runden gewonnen ist das Spiel zu Ende.	0,5	Kapitel 8.2	26. Juni 2025

2 Einführung Linux

2.1 Einführung

Die Betriebssysteme-Übungen der Vorlesung Einführung in die Informatik 2 dienen dazu, einige der theoretisch dargestellten Konzepte aus der Vorlesung anhand von Beispielen praktisch zu verdeutlichen. Hierzu werden die grundsätzlichen Konzepte zu Parallelisierung basierend auf Threads, deren Synchronisierung, Datenaustausch und die daraus resultierenden Probleme anhand von Beispielen dargestellt.

Dabei werden sukzessive die benötigten Bibliotheksfunktionen beschrieben und anhand von Übungen das Herangehen an die jeweilige Problematik dargestellt.

In der Übung wird dazu auf die POSIX Bibliotheken zurückgegriffen, die u.a. von Linux implementiert sind.

2.1.1 Die POSIX Bibliotheken

Die POSIX Threads oder auch pthreads ist eine portable Thread-Bibliothek, deren API in POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995) standardisiert ist. Diese wird von vielen Unix-artigen Betriebssystemen wie Linux, MacOS, FreeBSD, NetBSD, OpenBSD aber auch Android und AUTOSAR implementiert. Zudem gibt es für Windows Dritt-Bibliotheken (bsplw. winpthreads), die basierend auf der Windows API einen Wrapper für die phtreads API bereitstellt.

Die pthreads definieren für die Programmiersprache C verschiedene Datentypen, Funktionen und Konstanten. Diese werden in der `pthread.h` Header-Datei definiert und in der zugrundeliegenden Bibliothek implementiert. Darin sind knapp 100 Funktionen definiert, die unter anderem die Funktionalität für Thread-Management und Synchronisierung bieten. Um diese direkt zu identifizieren, tragen alle Funktionen das Präfix `pthread_`.

Neben den in pthread definierten Synchronisierungsmechanismen, gibt es zusätzlich die POSIX Semaphore API, die in POSIX.1b, Real-time extensions (IEEE Std 1003.1b-1993) standardisiert ist. Diese sind mit der pthread kompatibel, tragen jedoch das Präfix `sem_`.

Nachfolgend wird hauptsächlich auf die Linux-Implementierung der GNU C Library eingegangen, generell sind die Konzepte auf pthread kompatiblen Bibliotheken, aber auch eine vielzahl weiterer Thread-Bibliotheken, adaptierbar. Eine Unterstützung für andere C-Bibliotheken oder Betriebssysteme kann jedoch nur bedingt gegeben werden.

2.1.2 Linux Distribution

Als verwendete Linux Distribution kann entweder auf die virtuelle Maschine von Prof. Regensburger oder alternativ auf ein beliebiges Linux-System zurückgegriffen werden. Letzteres kann sowohl als virtuelle Maschine oder als native Installation eingesetzt werden.

Als weitere Option bietet Ihnen Windows 10[1] seit dem Build 16215 das Windows-Subsystem für Linux. Dies ermöglicht es, eine Linux UserLand-Umgebung unter Windows zu installieren. Eine Ubuntu 20.04 LTS oder Ubuntu 22.04 LTS Installation unter dem Windows-Subsystem für Linux bietet nach alle benötigten Voraussetzungen für die nachfolgenden Übungen der Betriebssystem Vorlesung. Für die Übungen sollte die WSL 2-Architektur verwendet werden;. Beachten Sie hierzu die vom Microsoft unter [**microsoft_wsl**] oder Ubuntu unter [**ubuntu_wsl**] bereitgestellten Dokumentationen.

Achten Sie jedoch darauf, dass die GNU C Library als C-Bibliothek und der GCC als Compiler angeboten werden. Beides wird in nahezu allen Desktop-Distributionen, wie z.B. Debian, Ubuntu, openSUSE, Fedora und Arch Linux und deren Derivate mitgeliefert. Grundsätzlich sollte auf ihrem System der GCC als C-Compiler sowie die Entwicklungsbibliotheken und Header-Dateien der C-Bibliotheken vorhanden sein.

Bei der Verwendung der virtuellen Maschine von Prof. Regensburger sind diese Voraussetzungen gegeben. Zudem sollten Sie mit dieser VM vertraut sein. Beachten Sie dabei auch die bereitgestellten Informationen zu diesem System.

Unter den meisten Debian-basierten Systemen (hierzu zählt auch Ubuntu) lässt sich diese Voraussetzungen für die nachfolgenden Übungen mit der Installation folgender beider Pakete und deren Abhängigkeiten erfüllen:

- `libc-dev`
- `gcc`

Diese Pakete sind generell unter den verschiedenen Distributionen verfügbar, jedoch können diese sich in der genauen Benennung unterscheiden.

2.2 Erstellen, Übersetzen und Ausführen der Übungsaufgaben

Im nachfolgenden wird ein kurzer Überblick gegeben, mithilfe welcher Tools die Übungen realisiert werden können. Zudem wird auf die Parameter des GCCs eingegangen, die für den erfolgreichen Übungsbetrieb nötig sind.

Hierbei wird nur auf Linux Systeme mit GCC und GNU C Library eingegangen. Eine Übertragbarkeit auf andere Systeme ist zwar möglich, darauf wird aber nicht weiter eingegangen und es kann keine Unterstützung angeboten werden.

[1] mit Ausnahme von Windows 10 S

2.2.1 Erstellen und Editieren des Quellcodes

Zum Erstellen der Übungsaufgaben eignet sich nahezu jeder Editor. Hier können Sie nach Ihren eigenen Vorlieben wählen.

Als grafische Editoren kann man auf die von der Desktopumgebung bereitgestellten Editoren zurückgreifen. Diese bieten meist eine Syntaxhervorhebung an, welches die Strukturen im Text leichter erkennen lassen und wodurch auch Tippfehler schneller auffallen. Zu den üblichen Editoren gehören kate (KDE), gedit (GNOME), Pluma (Mate) sowie Notepad++[2] unter Windows 10, die einen ähnlichen Umfang bieten. Für die WSL2 unter Windows 11 steht durch einen integrierten X11-Server auch eine graphische Oberfläche zur Verfügung.

Neben den grafischen Editoren können auch auf Editoren für die Kommandozeile zurückgegriffen werden. Zu den gängigsten gehören hier nano, vim oder emacs, die auf verschiedene Paradigmen aufsetzten und daher unterschiedlich steil in der Lernkurve sind. Bei der Verwendung der Kommandozeilen-Editoren könnte auf eine grafische Oberfläche verzichtet werden, welches sich zudem bei einem Windows-Subsystem für Linux anbietet.

Die Nutzung von integrierten Entwicklungsumgebung, wie Visual Studio Code, Eclipse, CLion, XCode usw. ist grundsätzlich möglich, wobei das Aufwand-Nutzen Verhältnis in keinem Verhältnis steht. In den Übungen werden i.d.R. keine aufwendigen bzw. umfangreichen Projekte behandeln, die eine Verwaltung der Projektstruktur mit sich zieht, sondern es werden kurze Fallbeispiele implementiert. Daher kann auch während dem Übungsbetrieb keine Unterstützung für die jeweiligen IDEs gegeben werden.

2.2.2 Übersetzen des Quellcodes

Zu Übersetzen des zuvor erstellten Quellcodes wird in den nachfolgenden Übungen der GCC verwendet. Die genaue Verwendung und Beschreibung der für die Übung nötigen Parameter werden in den folgenden Kapiteln bereitgestellt.

Überprüfen der Voraussetzung (Linux)

Um sicher zu stellen, dass die Voraussetzungen gegeben sind und dadurch bei den späteren Übungen zu keinen unerwarteten Problemen kommt, empfiehlt es sich den Compiler und die entsprechenden Bibliotheken zu testen.

Um zu verifizieren, dass sowohl der GGC installiert ist und das POSIX Thread Modell unterstützt wird, lässt sich wie folgt vorgehen.

Rufen Sie den GCC mit dem Parameter `gcc -v` auf. Dieser Aufruf listet die Konfiguration der installierten Version auf. In der Beispielausgabe (Debian Bullseye, März 2020) in Listing 2.1 wird in Zeile 21 wie gewünscht als Thread Model posix ausgegeben. Zusätzlich lässt sich auch über die Konfigurationsparameter (Listing 2.1, Zeile 9) erkennen, dass die Threads mit der Option posix konfiguriert wurden.

Sollte bei Ihnen hier nicht posix sondern beispielsweise single stehen, benötigen sie einen anderen GCC, der mit entsprechender Multi-Threading-Erweiterung erstellt ist.

Zudem können Sie überprüfen, ob die pthread Header-Dateien unter `/usr/include/pthread.h` verfügbar sind. Sollte dies nicht der Fall sein, müssen die Entwicklungsbibliotheken und Header-Dateien der GNU C Library nachinstalliert werden.

[2] Notepad++: `https://notepad-plus-plus.org/`

```
user@debian:~$ gcc -v
...
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Debian 9.2.1-30'
 --with-bugurl=file:///usr/share/doc/gcc-9/README.Bugs
 --enable-languages=c,ada,c++,go,brig,d,fortran,objc,obj-c++,gm2 --prefix=/usr
 --with-gcc-major-version-only --program-suffix=-9
 --program-prefix=x86_64-linux-gnu- --enable-shared --enable-linker-build-id
 --libexecdir=/usr/lib --without-included-gettext --enable-threads=posix
 --libdir=/usr/lib --enable-nls --enable-bootstrap --enable-clocale=gnu
 --enable-libstdcxx-debug --enable-libstdcxx-time=yes
 --with-default-libstdcxx-abi=new --enable-gnu-unique-object
 --disable-vtable-verify --enable-plugin --enable-default-pie --with-system-zlib
 --with-target-system-zlib=auto --enable-objc-gc=auto --enable-multiarch
 --disable-werror --with-arch-32=i686 --with-abi=m64
 --with-multilib-list=m32,m64,mx32 --enable-multilib --with-tune=generic
 --enable-offload-targets=nvptx-none,hsa --without-cuda-driver
 --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu
 --target=x86_64-linux-gnu --with-build-config=bootstrap-lto-lean
 --enable-link-mutex
Thread model: posix
gcc version 9.2.1 20200224 (Debian 9.2.1-30)
```

Listing 2.1: Informationen über GCC

Übersetzten und Linken des Quellcodes

Bei der Übersetzung des Quellcodes in den Objektcode (vgl. Listing 2.2) muss zusätzlich angegeben werden, dass Multitasking mithilfe der pthread-Bibliothek verwendet wird. Weitere Zusatzoptionen werden nicht benötigt, da die Header-Datei `pthread.h` unter einem Standard-Pfad, i.d.R. unter `/usr/include/`, abgelegt ist.

```
user@debian:~$ gcc -Wall -c -pthread -o example.o example.c
```

Listing 2.2: Übersetzen in Objektcode

Dabei haben die Parameter folgende Bedeutung:

`-Wall` Gibt möglichst viele und sinnvolle Warnungen aus.
Dieser Parameter ist nicht zwingend notwendig, weist aber auf potenzielle Fehler hin. Diese erleichtern oftmals die Fehlersuche und es werden Anmerkungen zu potenziellen Fehlerquellen oder falschen Parameterwerten angegeben.

`-c` Quellcode wird nur kompiliert und noch nicht gelinkt.

`-pthread` Explizite Option für pthread-Bibliothek. Setzt sowohl die internen Flags für den Präprozessor als für auch den Linker.

`-o` Legt den Namen der Ausgabedatei fest.
Falls nicht angegeben, wird standardmäßig beim alleinigen Kompilieren der Quellcode-Name genommen und die Erweiterung `.o` gesetzt.

Da die pthread eine Zusatzbibliothek, wie u.a. auch die Mathematik-Bibliothek, muss dies erneut auch beim Binden (vgl. Listing 2.3) mit angegeben werden.

```
user@debian:~$ gcc -Wall -pthread -o example example.o
```

Listing 2.3: Linken des Objektcode mit der pthread Bibliothek

Dazu muss ein der nachfolgenden Parameter angegeben werden:

`-l<lib>` Bindet an eine Zusatzbibliothek (hier -lpthread für libpthread), für jede beliebige Bibliothek möglich.

`-pthread` Explizite Option für pthread-Bibliothek. Setzt sowohl die internen Flags für den Präprozessor als für auch den Linker.

Es empfiehlt sich daher, dass mit der `-pthread` Option gearbeitet wird, da hier neben der zu bindenden Bibliothek weitere Flags gesetzt werden. Dies ist vor allem dann interessant, wenn weitere GNU C Library Funktionen genutzt werden, die ihrerseits Multitasking-Flags erwarten.

Neben der getrennten Übersetzung und dem anschließenden Linken einer Applikation, lassen sich beide Schritte gemeinsam durchführen. Dies lässt sich ähnlich wie Listing 2.3 durchführen, nur muss der Parameter `-c` weggelassen werden. Zudem muss die Dateiendung `.o` weggelassen werden. Wird bei diesem Übersetzungsvorgang die Dateiendung `.o` gesetzt, wird anstelle einer ausführbaren Datei einer Programmbibliothek erstellt.

Mögliche Fehler und deren Lösung

Beim Erstellen von Programmen treten immer wieder Fehler auf. Dies kann entweder an einem fehlerhaften Programmfluss liegen oder an einer nicht korrekten Nutzung des GCC. Die nachfolgenden Fehlermeldungen zeigen übliche Fehler und deren mögliche Lösung in Bezug auf pthread.

Sollte die Fehlermeldung aus Listing 4 auftreten, so ist davon auszugehen, dass die GNU C Library nicht (richtig) installiert ist; vergleiche hierzu Kapitel 2.2.2.

```
main.c:1:21: error: no include path in which to search for pthread.h
    1 | #include <pthread.h>
      |                     ^
```

Listing 2.4: pthread Header wurde nicht gefunden

Sollte hingegen ein Fehler auftreten, dass die Referenz zu einem oder mehreren `pthread_` Funktion nicht gefunden wird, wie in Listing 2.5 dargestellt, kann davon ausgegangen werden, dass bei dem Linking-Prozess die pthread Bibliothek nicht explizit mit angegeben wurde. Dadurch kann der Funktionsaufruf nicht aufgelöst werden; vergleichen Sie hierzu Kapitel 2.2.2.

```
/usr/bin/ld: /tmp/cc2Gzsli.o: in function `main':
main.c:(.text+0x19): undefined reference to `pthread_create'
collect2: error: ld returned 1 exit status
```

Listing 2.5: pthread Linking Fehler

2.3 Erste Schritte

In diesem Kapitel sollen Sie sich mit Ihrem System vertraut machen und dabei erste Beispielapplikation übersetzten und ausführen. Auf eine detaillierte Beschreibung der verwendeten Funktionen und deren Parameter wird in den nächsten Übungen detailliert dargestellt.

2.3.1 Einrichten des Entwicklungs-Systems

In einem ersten Schritt richten Sie ihr gewünschtes System so weit ein, dass die damit arbeiten können. Sollten Sie auf einer nicht-persistenten VM arbeiten, denken Sie zudem daran, die jeweiligen Änderungen an Ihrem Quelltext auf einem Speichermedium oder Filehosting-Dienst wie Bitbucket zu sichern.

Sollten Sie eine virtuelle Maschine nutzen überprüfen Sie zudem wie viele Prozessoren Sie der vorher eingerichteten Maschine zur Verfügung stellen. Um die relevanten Probleme und daraus resultierende Effekte darzustellen, sollten Sie ihrem System mindestens zwei, besser vier Prozessoren zur Verfügung stellen.

Bei der Verwendung von Oracle VM VirtualBox 6.0.x und 6.1.x lässt sich dies über die Einstellungen der jeweiligen Instanz realisieren. Klicken Sie hierzu bei ausgeschaltetem Zustand den Button Ändern der gewünschten Instanz an. In dem erscheinenden Einstellungs-Menü wechseln Sie auf System und wählen unter dem Tab Prozessor mithilfe des Schiebereglers die gewünschte Anzahl an Prozessoren aus, vgl. Abbildung 2.1.

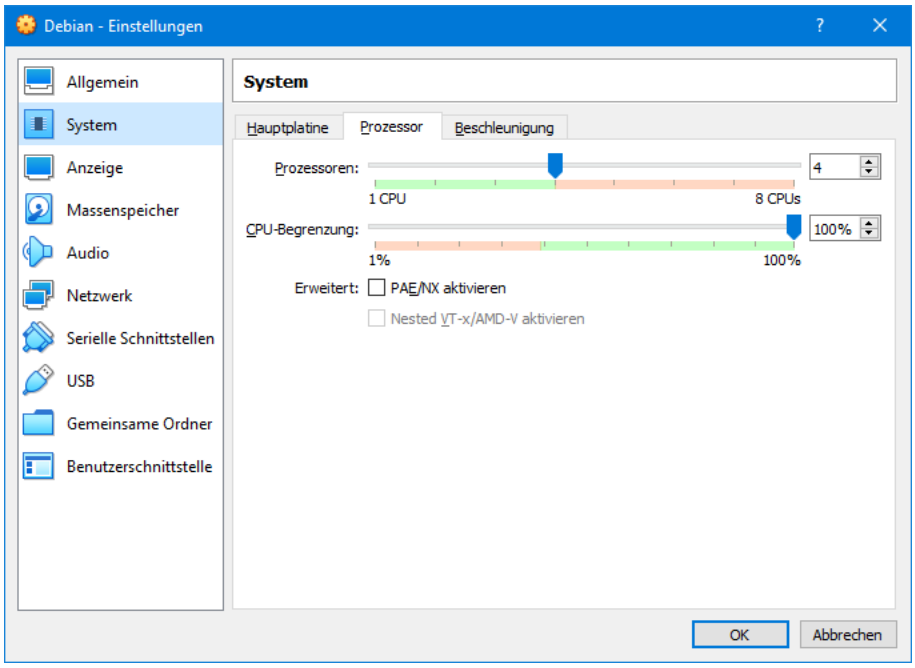

Abbildung 2.1: Mehrere Prozessoren zulassen

Prüfen Sie final, inwieweit die Voraussetzungen aus Kapitel 2.2.2 gegeben sind und installieren Sie gegebenenfalls die fehlenden Pakete, wie in Kapitel 2.1.2 beschrieben, nach.

2.3.2 Kompilieren, Linken und Ausführen einer Beispielanwendung

Als erste Übung und um den Umgang mit GCC zu wiederholen, kompilieren und linken Sie das in Listing 7.8 beschriebene Programm. Halten Sie sich hierzu an die Beschreibung in Kapitel 2.2.2. Probieren Sie hier sowohl das getrennte Kompilieren und Linken als auch beide Arbeitsschritte in einem aus.

```c
#include <stdio.h>
#include <pthread.h>

/* Thread-Funktion */
void *example_fct(void *args){
  size_t i;
  struct timespec sleep = { 0, 1000 };

  for(i = 0; i < 100; i++) {
    /* Schleife, die einen "zufälligen" Buchstaben ausgibt.
       Dieser wird anhand der Thread ID bestimmt. */
    putchar('a' + pthread_self() % 26);
    /* Warte ein wenig; 1 microsecunde */
    nanosleep(&sleep, NULL);
  }
  return NULL;
}

int main(){
  /* Lege zwei Thread-handle an */
  pthread_t threadA, threadB;

  /* Starte zwei Thread mit der auszuführenden Funktion example_fct.
     Konfigurations- und Übergabe-Parameter werden nicht gesetzt, daher NULL. */
  pthread_create(&threadA, NULL, &example_fct, NULL);
  pthread_create(&threadB, NULL, &example_fct, NULL);

  /* Warte auf Beendigung der beiden Threads */
  pthread_join(threadA, NULL);
  pthread_join(threadB, NULL);

  return 0;
}
```

Listing 2.6: Erste Beispielanwendung

Versuchen Sie dabei den Quellcode in den groben Zügen nachzuvollziehen und den Ablauf und die damit verbundene zeitliche Reihenfolge zu verstehen. Ein detailliertes Verständnis der einzelnen Funktionen sowie deren Parameter ist noch nicht nötig. Diese Details werden in einem der nächsten Übungen dargestellt. Haben Sie jetzt schon Interesse an den Details, können Sie die unter [**wolf2006**] oder den Linux Manpages [**man_pthread_create**, **man_pthread_join**, **man_pthread_self**] nachschlagen.

Führen Sie die vorher übersetzte Applikation mehrfach aus (Beispielausgabe in Listing 2.7).

- Verhält sich diese Funktion, wie Sie es erwartet haben?
- Was machen Sie für Beobachtungen?
- Können Sie sich die Änderungen in dem Mustern (Folge der Buchstabenausgabe) erklären?

```
user@debian:~$ ./first_example
iiiiiiissiissiissisiissiiisissisissiisisiissiissii↩
ssiissiisisisiisissiisisiisisiisisisisiisisiisisiisi↩
siisisiissiissiissiissiisissiissiisissisiisisiisis↩
siissiissiissiissiissiisisiisisissssssssssssssssss

user@debian:~$ ./first_example
ukuukukkuukkukuukkuukukukkukuukuukukukukukuukkuukk↩
uukkuukkuukkuukkukukkuukkuukukkukukkuukkuukkukuuku↩
kkukukukkuukkuukkuukukkukuukukukukukukkuukkuukkuuk↩
kuukkukuukukuukkuukkuukkuukkuukkuukkuukkuukkuukkuk
```

Listing 2.7: Beispielausgaben der ersten Appliaktion

2.3.3 Übergabe von Informationen an Thread-Funktion

Die Übergabe von Informationen ist (nicht nur) bei Multi-Threading Funktionen einer der essenziellen Anforderungen um parametrisierbare Aufgabenstellungen zu implementieren. Gegenüber der generellen Funktionsentwicklung, bei der die Übergabeparameter nahezu beliebig gewählt werden können, muss sie bei pthreads der Funktions-Zeiger der vorgegebenen Definitionen entsprechen. Dieser ist wie folgt definiert:

```
void *(*funktion)(void *)
```

Die einzelnen Teiler dieser Funktionsdefinition lassen sich wie folgt interpretieren:

`void *` Der Rückgabewert der Funktion ist ein Zeiger vom Typ `void *`. Dadurch kann ein beliebiger Zeigertyp verwendet werden, der jedoch durchgängig bekannt sein muss.

`(*funktion)` Der Funktionszeiger auf die Funktion. Dabei handelt es sich um den Namen der Funktion, den Sie entwickeln.

`(void *)` Der Übergabeparameter der Funktion, auch hier wieder vom Typ `void *`. Dies ermöglicht wiederum, jeden beliebigen Zeiger an die Funktion zu übergeben, wodurch auch hier wieder bekannt sein muss, welchen Typ er besitzen soll.

Anhand des Beispiels in Listing 2.8 wird dies noch einmal veranschaulicht erklärt. Das Ziel hier ist es, der Funktion einen `int` als Zeiger zu übergeben und einen in der Funktion „errechneten“ `int` zurückzuerhalten.

Um dies zu implementieren, wird in Listing 2.8, Zeile 24 der Zeiger der Variable `int threadParam` übergeben.

In der eigentlichen Thread-Funktion wird in Listing 2.8, Zeile 7 die `void *args` Variable wieder explizit auf einen `int *` gecastet und auf die Variable `int *inParam` geschrieben. In Zeile 12 soll nun ein `int` zurückgegeben werden. Dazu wird die Ganzzahl in einen `void *` gecastet und mit `return` zurückgegeben.

In der Hauptanwendung wird währenddessen so lange mit `pthread_join` gewartet, bis die Thread-Funktion terminiert. Um den Return-Wert als `int` zu erhalten, muss dieser wieder entsprechend gecastet werden. Hierzu muss man folgendes im Hinterkopf behalten:

geforderter Thread Datentyp	eigener Datentyp
`void *`	`int`
`void **`	`int *`

```c
#include <stdio.h>
#include <pthread.h>
#include <stdint.h>


/* Thread-Funktion */
void *example_fct(void *args){
  /* Die Übergabe wird zurück auf einen int-Pointer gecastet*/
  uintptr_t *inParam = (uintptr_t *)args;

  /* Der Inhalt des Pointers wird ausgegeben */
  printf("Infos von Main: %llu\n", *inParam);

  return (void *)101010;
}

int main(){
  /* Lege ein Thread-Handle, einen Übergabe- und einen Rückgabeparameter an */
  pthread_t thread;
  uintptr_t threadParam = 42;
  uintptr_t threadRetParam = -1;

  /* Starte einen Thread mit der auszuführenden Funktion example_fct.
     Zudem wir ein Parameter übrgeben.
     Konfigurations-Parameter werden nicht gesetzt, daher NULL. */
  pthread_create(&thread, NULL, &example_fct, &threadParam);

  /* Warte auf Beendigung des Threads */
  pthread_join(thread, (void **)(&threadRetParam));

  /* Inhalt des Rückgabeparameters ausgeben */
  printf("Rueckgabe von Thread: %llu\n", threadRetParam);

  return 0;
}
```

Listing 2.8: Beispielanwendung mit Parameterübergabe

Daher wird als zweiter Parameter der Zeiger auf die `int threadRetParam` übergeben, welches dem `void **` Zeiger entspricht.

2.4 Ihre Aufgabe

Erweitern Sie das Listing 2.8 so, dass sie nicht nur primitive Datentypen übergeben können.Als Beispiel soll hierzu eine Datenstruktur erstellt werden, bei der Sie einen Namen als Zeichenkette mit fester maximaler Länge und die tatsächliche Länge des Strings als vorzeichenlose Ganzzahl ausgetauschen können.

Der Name des Studenten soll vorab in der `main`-Funktion als C-String[3] gesetzt werden, die Länge des Namens soll innerhalb der Thread-Funktion berechnet werden. Greifen Sie hier beispielsweise auf die Funktion `size_t strlen(const char *s)` [**man_strlen**] aus der String-Bibliothek in `#include <string.h>` zurück.

Geben Sie nach Terminierung der Thread-Funktion die Inhalte der Datenstruktur mithilfe von `printf` aus. Dies soll zeigen, dass die Werte korrekt gesetzt sind und auch nach Terminierung des Threads vorhanden sind.

Der Rückgabewert des Threads wird nicht benötigt und kann ignoriert werden.

2.5 Ergebnis und Abnahme

Um das Testat zu bekommen, müssen folgende Ergebnisse vorgezeigt werden bzw. folgende Fragen beantwortet werden:

1. Das System ist soweit eingerichtet, dass die Beispiele übersetzten werden können.
2. Das Programm ist in der Lage, an eine Thread-Funktion eine Struktur mit Namen zu übergeben und nach Beendigung des Threads den Inhalt auszugeben.
3. Die Thread-Funktion setzt die Länge der übergebenen Zeichenkette korrekt.

[3] Ein C-String ist ein `char`-Array, dass mit einem null-Byte (`'\0'`) terminiert.

3 Threads

3.1 Parallelität von Applikationen

Sollen mehrere Aufgaben gleichzeitig abgearbeitet werden, können verschiedene Konzepte verwendet werden. Je nachdem, welches Ziel man mit der Parallelisierung anstrebt, setzt man entweder auf Prozesse oder auf Threads. In der Vorlesung haben Sie dabei folgende mögliche Unterscheidung kennengelernt:

Prozesse Paralleles Ausführen von unterschiedlichen Applikationen, die keine enge Interaktion untereinander haben.
Hierunter fallen individuelle Applikationen wie Browser, Messanger, Mediaplayer, usw.

Threads Parallele Arbeiten, die innerhalb einer Applikation ausgeführt werden und oft eine Interaktion miteinander benötigen.
Hierunter fallen Applikationen wie bspw. Webserver, die hunderte Anfragen pro Sekunde parallel abarbeiten.

Wie schon in der in der Einführung angedeutet, werden die Übungen hauptsächlich den zweiten Punkt, die Threads, behandeln.

Dazu wird in der Übung auf die POSIX Bibliotheken zurückgegriffen, die u.a. von Linux implementiert werden. Als Übungsplattform wir dazu entweder WSL2, die von Prof. Regensburger bereitgestellte virtuelle Maschine oder eine beliebige Desktop-Distribution, die GNU C Library als C-Bibliothek und dem GCC als Compiler bereitstellt, genutzt. Die genauen Voraussetzungen werden in „Übung Betriebssysteme – Einführung“ beschrieben.

3.2 Die POSIX Threads

Bei den POSIX Threads oder auch pthreads handelt es sich um eine portable Thread-Bibliothek, deren API in POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995) definiert wurde.

Unter Linux werden die pthreads als Native POSIX Thread Library (NPTL) innerhalb der GNU C Library implementiert. Die als NPTL laufenden Threads eines Prozesses werden innerhalb der gleichen Thread Gruppe abgebildet. Alle Mitglieder dieser Gruppe besitzen somit auch die gleiche PID aber unterschiedliche TID, vgl. Abbildung 3.1.

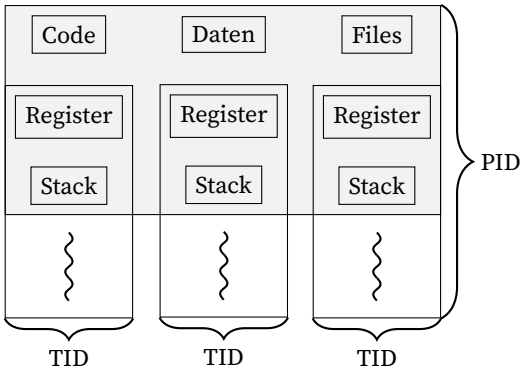

Abbildung 3.1: PID und TIDs eines pthread

Bei der Implementierung der Threads wird auf den `clone()` System Calls zurückgegriffen. Dazu wird jeder Thread auf einen eigen Kernel Scheduling Eintrag abgebildet.

3.2.1 Starten eines Threads

Um neben dem Haupt-Thread (i.d.R. die `main()`) weitere Threads zu starten, bietet die pthread-Bibliothek die Funktion in Listing 3.1 . Diese Funktion startet einen neuen Thread innerhalb des aktuellen Prozesses. Der Einsprungpunkt des neu erstellten Threads wird dabei als Funktionszeiger übergeben. Zudem lassen sich die Attribute des Threads anpassen, falls die Default-Einstellungen nicht genügen.

Bevor die Funktion erfolgreich terminiert, wird die ID des erstellten Threads in dem Puffer von Typ `pthread_t` gespeichert. Dieser dient im weiteren den Thread zu identifizieren und weitere Interaktionen mit diesem durchzuführen.

```
1 int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
2                    void *(*start_routine) (void *), void *arg);
```

Listing 3.1: Starten eines Threads [**man_pthread_create**]

Die zu übergebende Parameter haben folgende Bedeutung:

`thread` Puffer, in die ID des neuen Threads gespeichert wird.

`attr` Das Argument zeigt auf eine `pthread_attr_t` Struktur (Kapitel 3.2.1), mit welchen Eigenschaft der Thread zur Startzeit initialisiert werden soll. Wird als Parameter `NULL` übergeben, werden die Standardeigenschaften verwendet.

`start_routine` Funktionspointer auf den Einsprungpunkt des neuen Threads

arg Argumente für den übergebenen Funktionspointer

Der **Rückgabewert** der Funktion ist im erfolgreichen Fall 0. Sollte ein Fehler auftreten, wird ein Fehlercode zurückgegeben, zudem ist der Wert von `thread` nicht definiert.

Konfiguration der Thread-Attributen

Sind die Standard-Einstellungen nicht ausreichend, oder werden andere Einstellungen benötigt, so kann dies über die `pthread_attr_t` Struktur konfiguriert werden.

Um die Einstellungen anzupassen, muss das Struktur-Objekt erst mit der Init-Funktion aus Listing 3.2 initialisiert werden. Dabei werden die internen Werte auf die Default-Einstellungen gesetzt, die auch bei der Übergab von `NULL` in Listing 3.1 gesetzt werden. Wird das Attributobjekt nicht mehr benötigt, so muss diese mit der Destroy-Funktion aus Listing 3.2 gelöscht werden.

```
1 int pthread_attr_init(pthread_attr_t *attr);
2 int pthread_attr_destroy(pthread_attr_t *attr);
```

Listing 3.2: (De-)Initialisieren von Thread-Attributen [**man_pthread_attr_init**]

Zu den Parametern, die eingestellt werden können, sind Optionen für den Stack, die der Thread verwendet. Normalerweise verwendet ein Thread auf Intel-Architekturen (sowohl i386 als auch x86_64) eine Stack-Größe von 2 MB [**man_pthread_create**], welcher automatisch generiert wird.

Sollen hier benutzerdefinierte Größen oder Speicherbereiche verwendet werden, so lässt sich dies mit entsprechenden Funktionen anpassen. Wird nur ein andere Stack-Größe gewünscht, da beispielsweise lokal große Strukturen angelegt werden sollen, lässt sich das mit der Set-Funktion aus Listing 3.3 realisieren. Hierzu wird der Funktion neben dem vorher initialisierten Attributobjekt zusätzlich die gewünschte Stack-Größe übergeben. Entsprechend der Set-Funktion, lässt sich mithilfe der Get-Funktion aus Listing 3.3 die aktuell konfigurierte Größe auslesen.

```
1 int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
2 int pthread_attr_getstacksize(const pthread_attr_t *attr, size_t *stacksize);
```

Listing 3.3: Manipulation der Stack-Größe mithilfe von Thread-Attributen [**man_pthread_attr_setstacksize**]

Soll nicht nur die Stack-Größe, sondern der gesamte Stack vom Nutzer verwaltet werden, so kann dies mit den Funktionen in Listing 3.4 realisiert werden. Hierbei muss vom Nutzer jedoch selbst die Speicherverwaltung übernommen werden.

```
1 int pthread_attr_setstack(pthread_attr_t *attr,
2                           void *stackaddr, size_t stacksize);
3 int pthread_attr_getstack(const pthread_attr_t *attr,
4                           void **stackaddr, size_t *stacksize);
```

Listing 3.4: Manipulation des Stacks mithilfe von Thread-Attributen [**man_pthread_attr_setstack**]

Neben den hier beispielhaft genannten Möglichkeiten zur Manipulation der Thread-Attribute lassen sich noch einige mehr Parameter manipulieren. Hierzu zählen:

Scheduling Hiermit lassen sich sowohl die Scheduling-Algorithmen als auch die Prioritäten auswählen.

Detach-Status Hierüber lässt sich festlegen, ob die der Thread im Joinable- oder im Detached-Modus ist. Ein Thread kann jederzeit nach der Initialisierung in den Detached-Modus gesetzt werden, siehe Kapitel 3.2.4.

CPU-Zugehörigkeit Hier lässt sich festlegen, auf welchen CPUs der Thread laufen darf. Hierbei handelt es sich um eine nicht standardisierte Erweiterung der GNU C Library unter Linux. Diese Erweiterungen tragen das Suffix `_np` für „nonportable“ im Namen.

Da diese Optionen in der Übung nicht gebraucht werden, wird hier nicht im Detail eingegangen. Der Vollständigkeit halber werden sie erwähnt, da einige u.a. das Echtzeitverhalten bestimmen können.

3.2.2 Beenden eines Threads

Das Terminieren eines Threads kann mithilfe von zwei unterschiedlichen Möglichkeiten durchgeführt werden. Hierbei kann entweder mithilfe von der Funktion in Listing 3.5 beendet werden. Dabei hat die Funktion keinen Rückgabewert, da sie per Definition immer gelingt.

```
1 void pthread_exit(void *retval);
```

Listing 3.5: Terminieren eines Threads [**man_pthread_exit**]

Der zu übergebende Parameter hat folgende Bedeutung:

`retval` Rückgabewert der Funktion an den wartenden Thread (meistens den Haupt-Thread) des gleichen Prozesses.

Neben dieser expliziten Funktion kann ein Thread mithilfe von `return` beendet werden. Dies initiiert dann implizit den Aufruf von `pthread_exit()`, falls es sich nicht um den Haupt-Thread handelt.

Bei dem Rückgabewert bzw. Exit-Status der Thread-Funktion muss natürlich beachtet werden, dass keine lokalen Objekte (Zeiger auf lokale Variablen) zurückgegeben werden. Diese sind, wie bei jeder C-Funktion, nach Terminierung nicht mehr gültig und können daher auch nicht mehr nach erfolgreichen `pthread_join` verwendet werden.

3.2.3 Warten auf das Beenden eines anderen Threads

Die Funktion in Listing 3.6 wartet auf das Ende des ausgewählten Threads. Kehrt die Funktion erfolgreich zurück, so wurde der Thread beendet und die Thread-internen Ressourcen wurden freigegeben. Falls der ausgewählte Thread schon beendet wurde, kehrt die Funktion sofort zurück. Im Anschluss steht der Exit-Status, der bei Beendigung gesetzt wurde (vgl. Kapitel 3.2.2), zur Verfügung.

Zu beachten ist, dass der Thread, auf den gewartet werden soll, im Joinable-Modus ist.

Die zu übergebenden Parameter haben folgende Bedeutung:

`thread` ID des Threads, auf dessen Terminierung gewartet werden soll.

```c
int pthread_join(pthread_t thread, void **retval);
```

Listing 3.6: Warten auf Terminierung eines anderen Threads [**man_pthread_join**]

`retval` Zeiger auf das Rückgabeobjekt, dass der Thread als Exit-Code zurückgegeben hat. Wird NULL übergeben, so wird der Rückgabewert nicht gespeichert.

Generell sind alle Threads innerhalb eines Prozesses gleichberechtigt, somit kann jeder Thread auf einen anderen warten. Häufig wird zwar die Funktion in der `main()` verwendet, dies ist aber nicht zwingend nötig.

3.2.4 Unabhängig machen eines Threads

Neben dem Warten auf einen Thread, ist es auch möglich, einen Thread unabhängig zu machen. Dazu wird die Funktion in Listing 3.7 und die Thread-ID des zu detachenden Threads übergeben.

Das Detaching von Threads kann mit einem Daemon-Prozess[**wolf2006**] unter Linux verglichen werden, der normalerWeise im Hintergrund läuft, und mit dem nicht direkt interagiert wird.

```c
int pthread_detach(pthread_t thread);
```

Listing 3.7: Warten auf Terminierung eines anderen Threads [**man_pthread_detach**]

Neben der Funktion in Listing 3.7 kann bei der Generierung von Threads die entsprechende Attribute mithilfe von `pthread_attr_setdetachstate(pthread_attr_t *attr, PTHREAD_CREATE_DETACHED);` [**man_pthread_attr_setdetachstate**] gesetzt werden.

3.3 Übungsaufgabe

Generell ist die Idee hinter jeder Parallelisierung die Beschleunigung der gesamten Aufgabe. Dabei wird die Aufgabe i.d.R. in mehrere kleine Arbeitspakete aufgeteilt und auf verschiedenen Prozessorkerne verteilt, wo sie dann parallel bearbeitet werden.

Diese Beschleunigung wird als Speedup bezeichnet[1] und ist im Idealfall für n Prozessorkerne genau n-mal so schnell. Dies wird nahezu nie erreicht, da durch das Verteilen zusätzlicher Verwaltungs-Overhead entsteht. Dieser kann im Extremfall sogar dazu führen, dass eine Verlangsamung gegenüber einer reinen sequenziellen Ausführung entsteht.

3.3.1 Berechnung des Collatz-Problems

Als rechenintensive Aufgabe wird in dieser Übung das Collatz-Problem aus Algorithmus 1 verwendet, in Nachfolgenden auch Lastfunktion genannt. Hier soll nicht weiter auf den Algorithmus und etwaige Optimierungen eingegangen werden, da dieser nur als Referenzlast dienen soll.

Das Collatz-Problem, auch als (3n+1)-Vermutung bezeichnet, ist ein ungelöstes mathematisches Problem, das 1937 von Lothar Collatz gestellt wurde. Das Problem ist zwar einfach zu formulieren, aber notorisch schwierig. Weitere Informationen zu diesem interessanten Problem lassen sich in dem zugehörigen Wikipedia-Artikel oder in einschlägiger Literatur nachlesen.

Algorithmus 1 : Algorithmus des Collatz-Problems

Input : Zu prüfende Zahl x
Result : Anzahl der Iterationen

$cnt \leftarrow 0$;
while $x > 1$ **do**
  if x is even **then**
    $x \leftarrow \frac{x}{2}$
  else
    $x \leftarrow 3 * x + 1$
  $cnt \leftarrow cnt + 1$
return cnt

Für diese Aufgabe ist folgendes Szenarien umzusetzen: Berechnen Sie für den Wertebereich von 1 bis $100\,000\,000$ die jeweilige Länge der Collatz-Folge.

3.3.2 Parallelisieren der Berechnung

Um eine performante Parallelisierung zu erreichen, ist es wichtig, die zu berechnende Funktion möglichst gleichmäßig auf die einzelnen Threads zu verteilen.

Der Algorithmus des Collatz-Problems lässt sich hierzu sehr gut parallelisieren, indem einfach eine gleichmäßige Verteilung durchgeführt wird: für n Threads teilen Sie den Bereich in n (fast) gleich große Teile. Jeder Thread bekommt einen Teilbereich und durchsucht diesen.

[1] Ist t_{seq} die Laufzeit auf einem Kern und t_{par} die Laufzeit auf mehreren Kernen, so ergibt sich der Speedup $n = \frac{t_{seq}}{t_{par}}$

Hinweise zur Implementierung

Im Hauptprogramm berechnen Sie für jeden Thread den zu berechnenden Bereich. Am besten übergeben Sie Start- und Endwert des Bereiches als Parameter an den Thread. Da in C nur einen Parameter übergeben werden kann, müssen Sie eine Struktur erstellen, die Start- und Endpunkt des Bereiches enthält. Außerdem müssen Sie den Startwert, der zu längsten Folge führt, und die Anzahl der Iterationen von jedem Thread zurückgeben (in C kann man dazu den Rückgabewert in der Struktur ablegen).

Halten Sie zudem die Anzahl der Threads möglichst flexibel. Legen Sie die Anzahl der Threads entweder als Konstante an oder übergeben Sie diese als Argument an die `main` Funktion.

3.3.3 Ausführungszeit messen und Speedup errechnen

Um die Verkürzung der Laufzeit zu messen, messen Sie die Ausführungszeit ihrer Applikation. Führen Sie dazu die Applikation mit verschiedenen Anzahl an Threads aus und berechnen Sie den jeweiligen Speedup Faktor. Führen Sie diese Messung für beide Fälle durch und tragen Sie den Speedup in dasselbe Diagramm ein.

Zur Zeitmessung stehen ihnen zwei verschiedene Möglichkeiten zur Verfügung. Entweder Sie implementieren die Zeitmessung in ihre Applikation oder Sie messen die Ausführungszeit mit externen Tools.

Implementieren sie ihre Zeitmessung selbst, so können sie beispielsweise die POSIX-Funktion in Listing 3.8 verwenden. Diese befüllt die übergebene Struktur vom Typ `struct timespec` mit der aktuellen Zeit in Sekunden- und Nanosekunden-Anteil.

```
1 struct timespec {
2   time_t   tv_sec;        /* Sekunden */
3   long     tv_nsec;       /* Nanosekunden */
4 };
5
6 int clock_gettime(clockid_t clk_id, struct timespec *tp);
```

Listing 3.8: Warten auf Terminierung eines anderen Threads [**man_clock_gettime**]

Als zweite Option steht ihnen eine externe Messung mithilfe von Tools wie `time` zur Verfügung. Diese gibt die Ausführungszeit als drei verschiedene Werte mit folgender Bedeutung zurück:

real Die gesamte Laufzeit der Applikation. Wird oft als wall clock time bezeichnet.

user Die benötigte, akkumulierte Laufzeit auf allen CPUs.

sys Die Laufzeit, die innerhalb des Kernels abgearbeitet wurde.

Für die Zeitmessung ist hier die **real**-Zeit die interessante Zeit. Zudem kann man an der **user**-Zeit sehen, wie lang die Applikation auf allen Prozessoren gebraucht hat. Anhand dieses Werts kann man den Overhead berechnen, den man im Vergleich zu der sequenziellen Applikation benötigt.

3.4 Ergebnis und Abnahme

Um das Testat zu bekommen, müssen folgende Ergebnisse vorgezeigt werden bzw. folgende Fragen beantwortet werden:

1. Sie haben die Lastfunktion implementiert (sequenziellen Ausführung) und die Laufzeit vermessen. Geben Sie die Ausführungszeit für die Implementierung an.
2. Sie haben eine parallele Implementierung erstellt, bei der sich die Anzahl der Threads und der Bereich leicht anpassen lässt. Messen Sie die Laufzeit jedes einzelnen Threads, damit sichergestellt ist, dass alle Threads etwa die gleiche Laufzeit haben ($\pm 10\%$)
3. Berechnen Sie den SpeedUp-Faktor für 2 bis 16 Threads und stellen Sie beide Bereiche in einer Grafik dar.

4 Synchronisierung

4.1 Synchronisierung von Threads

In vielen Fällen arbeiten mehrere Threads auf denselben Daten oder tauschen Ergebnisse miteinander aus. Damit hier keine inkonsistenten Datensätze entstehen, muss sichergestellt werden, dass Operation auf den Daten vollständig durchgeführt wurde, bevor dieser erneut verwendet wird. Der Abschnitt, bei denen Inkonsistenzen auftreten können, wird in der Literatur mit dem Problem des kritischen Abschnitts beschrieben.

Um diesem Problem entgegenzuwirken, werden häufig kritische Abschnitte als nicht unterbrechbare Bereiche behandelt, wodurch sie nur sequentialisiert betreten werden. Dies lässt sich mithilfe des wechselseitiger Ausschluss (Mutex, Abk. für **mu**tual **ex**clusion) realisieren.

Dazu bieten die Betriebssysteme und darauf aufbauend verschiedene Bibliotheken weitreichende Möglichkeiten an, die zur Absicherung von kritischen Bereichen dienen.

4.2 Mutex

Für die Betreten- und Verlassen-Operationen bietet die pthreads-Bibliothek eine Mutex-Implementierung an, über die sich die Operationen zum wechselseitigen Ausschluss realisieren lassen. Dabei handelt es sich bei den Mutexen um einen binären Synchronisierungsmechanismen, der intern nur die beiden Zustände Besetzt oder Frei halten kann.

Dabei kann entweder auf statische oder dynamische initialisierte Mutex gesetzt werden, die sich durch die jeweiligen Initialisierungsroutinen unterscheiden werden. Um mit den Mutexen zu arbeiten, stehen drei verschiedene Funktionen (Listing 4.1) zur Verfügung, die jeweils mit dem Präfix `pthread_mutex_` gekennzeichnet sind:

```
1 int pthread_mutex_lock(pthread_mutex_t *mutex);
2 int pthread_mutex_trylock(pthread_mutex_t *mutex);
3 int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Listing 4.1: Betreten- und Verlassen-Operationen der pthread-Mutex

Mithilfe von `pthread_mutex_lock` oder `pthread_mutex_trylock` wird der kritische Abschnitt betreten und die übergeben Mutex-Datenstruktur gesperrt. Dabei blockiert die Funktion `pthread_mutex_lock` so lange, bis die Mutex von einem anderen Thread wieder freigegeben wird. Soll der Thread nicht auf unbestimmte Zeit blockiert werden, da neben dem kritischen Abschnitt weitere Tätigkeiten möglich sind, so kann mit `pthread_mutex_trylock` versucht werden, den kritischen Abschnitt zu betreten. Diese Funktion blockiert nicht und es muss anhand des Rückgabewertes geprüft werden, ob die Mutex durch den eigenen Thread gesperrt wurde.

Mithilfe von `pthread_mutex_unlock` wird nach Verlassen des kritischen Abschnitts die Mutex freigegeben, wodurch weitere Threads den kritischen Abschnitt betreten können.

Diese drei Funktionen geben jeweils bei fehlerfreier Ausführung eine Null zurück. Andernfalls wird ein Wert ungleich Null zurückgegeben, der sich als Fehlercode interpretieren lässt.

Wird Beispielsweise versucht, den kritischen Abschnitt mit `pthread_mutex_trylock` zu betreten und die Mutex ist schon von einem anderen Thread belegt, so wird `EBUSY` zurückgegeben.

4.2.1 Statische Mutexe

Statische Mutexe werden erstellt, indem ein globales oder statisches Handle (vgl. Exkurs zur Gültigkeit von Variablen) von Typ `pthread_mutex_t` mit der Konstanten `PTHREAD_MUTEX_INITIALIZER` initialisiert wird, vgl. Listing 4.2. Diese werden zu Programmstart erstellt und sind dadurch während der gesamten Laufzeit der Applikation gültig und müssen somit nicht weiter vom Nutzer verwaltet werden.

```
1 pthread_mutex_t globale_mutex = PTHREAD_MUTEX_INITIALIZER;
2
3 void fct() {
4   static pthread_mutex_t local_static_mutex = PTHREAD_MUTEX_INITIALIZER;
5 }
```

Listing 4.2: Möglichkeiten der Initialisierung statischer Mutexe

4.2.2 Dynamische Mutexe

Neben den statischen Mutexen, bietet die pthread-Bibliotek auch die Möglichkeit nutzergesteuert Mutexe zu verwalten. Diese Methode muss einerseits verwendet werden, wenn die Standard-Einstellungen nicht ausreichen und dadurch zusätzliche Attribute übergeben werden müssen (vgl. Unterabschnitt 4.2.3). Auf der anderen Seite kann es vorkommen, das dynamisch zur Laufzeit neue Mutexe benötigt werden oder die Anzahl erst im Laufe der Programmausführung feststeht.

Hierbei muss jedoch der Nutzer selbst sicherstellen, dass eine Mutex, nachdem sie nicht mehr gebraucht wird, wieder gelöscht bzw. zerstört wird.

Um neue Mutexe während der Laufzeit anzulegen oder zu löschen, stehen die Instruktionen aus Listing 4.3 zur Verfügung:

```
1 int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
2 int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Listing 4.3: (De-) Initialisierungsfunktionen für Mutexe

Bei der Initialisierungsfunktion `pthread_mutex_init` wird ein Zeiger auf eine neu erstellte Datenstruktur des Typs `pthread_mutex_t` übergeben und die gewünschten Attribute der Mutex. Mit der Übergabe von `NULL` wird eine Mutex erstellt, die die Default-Einstellungen wie bei den statischen Mutex enthält.

Beide Funktionen geben bei fehlerfreier Ausführung eine Null zurück. Andernfalls wird ein Wert ungleich Null zurückgegeben, der als Fehlercode interpretiert wird.

4.2.3 Mutex-Attribute

Mithilfe des Attributs `pthread_mutexattr_t` lässt sich das Verhalten der Mutexe in verschiedenen Situationen konfigurieren.

Dazu muss initial ein neu erstelltes Attribut-Handle initialisiert und nach Beendigung wieder freigegeben werden. Hierzu werden die beiden Funktionen aus Listing 4.4 verwendet.

```
1 int pthread_mutexattr_init(pthread_mutexattr_t *attr);
2 int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

Listing 4.4: (De-) Initialisierungsfunktionen für Mutex-Attribute

Dabei lässt sich beispielsweise einstellen, dass die Mutex nicht nur prozessintern, sondern prozessübergreifend funktionstüchtig ist. Hierzu muss natürlich vorab sichergestellt werden, dass das Mutex-Handle auch in einem Speicherbereich liegt, auf den von mehreren Prozessen zugegriffen werden kann. Dazu wird die Funktion aus Listing 4.5 verwendet.

```
1 int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared);
```

Listing 4.5: Gültigkeitsbereich von Mutex setzten.

Als Parameter wird der Funktion neben dem Attribut-Handle noch eines der folgenden `pshared` Attribute übergeben:

`PTHREAD_PROCESS_PRIVATE` Die Mutex wird als prozessinternes Objekt initialisiert, welches nur durch lokale Threads verwendet werden darf. Dies ist die Default-Einstellung und muss daher i.d.R. nicht gesetzt werden.

`PTHREAD_PROCESS_SHARED` Die Mutex wird als prozessübergreifendes Objekt initialisiert und kann dadurch auch von Threads anderer Prozesse verwendet werden.

Als weiteres Beispiel kann das Sperr-Verhalten der Mutex anpassen werden, in dem man ihren Typ ändert. Dies hat vor allem Einfluss darauf, wie sich eine Mutex verhält, falls der gleiche Thread mehrfach versucht die Mutex zu sperren. Hierzu wird die Funktion aus Listing 4.6 verwendet.

```
1 int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

Listing 4.6: Gültigkeitsbereich von Mutex setzten.

Dazu lassen sich vier verschiedene Modi einstellen, die folgen Eigenschaften mit sich bringen:

`PTHREAD_MUTEX_DEFAULT` Das generelle Verhalten bei mehrfachen Sperren oder Freigeben einer Mutex ist nicht definiert und führt zu einem undefinierten Verhalten. Zudem ist auch das Verhalten bei Freigabe nicht definiert, sollte die Mutex durch einen anderen Thread gesperrt worden sein. Die Implementierung erlaubt es seitens des Standards, die Funktionsweise auf eines der nachfolgenden Typen zu mappen.

`PTHREAD_MUTEX_NORMAL` Dieser Typ erkennt keine Deadlocks. Ein erneutes Sperren durch den gleichen Thread kann nicht mehr freigegeben werden, wodurch ein klassischer Deadlock induziert wurde. Zudem ist auch hier das Verhalten bei Freigabe nicht definiert, sollte die Mutex durch einen anderen Thread gesperrt worden sein.

PTHREAD_MUTEX_ERRORCHECK Dieser Typ bietet eine Fehlerprüfung. Versucht ein Thread das erneute sperren einer Mutex, bevor sie freigegeben wurde, kehrt sie mit einem Fehlercode zurück, anstelle in einen Deadlock zu geraten. Auch bei dem Freigaben einer nicht gesperrten Mutex oder bei einer von einem anderen Thread gesperrten Mutex wird ein Fehlercode zurückgegeben.

PTHREAD_MUTEX_RECURSIVE Dieser Typ erlaubt das mehrfache Sperren einer Mutex durch den gleichen Thread, ohne einer vorherigen Freigabe. Dadurch werden Deadlocks bei Mehrfachaufruf vermieden. Eine mehrfach gesperrte Mutex muss von einem Thread mit der gleichen Zahl an Freigaben entsperrt werden, bevor ein anderer Thread die Mutex übernehmen kann. Die Freigebe einer nicht gesperrten Mutex oder bei einer von einem anderen Thread gesperrten Mutex wird mit einem Fehlercode signalisiert.

Zu jeder der set-Funnktionen gibt es die entsprechende get-Funktion, mithilfe der sich der aktuelle Konfigurations-Zustand einer Mutex abfragen lässt.

Neben den hier beispielhaft aufgezählten Funktionalitäten gibt es noch einige weitere Funktionen, mit denen die Mutex-Eigenschaften angepasst werden können. Diese beginnen alle mit dem Präfix `pthread_mutexattr_`.

4.3 Semaphore

Neben dem Synchronisieren von Threads mithilfe von Mutexen lassen sich auch Semaphoren einsetzten. Diese sind im Gegenzug zu den Mutex nicht Teil der pthread-Bibliothek, sondern sind als POSIX Semaphore API in POSIX.1b, Real-time extensions (IEEE Std 1003.1b-1993) standardisiert. Dadurch sind diese Semaphoren auch unter `semaphore.h` definiert.

Semaphoren sind zählende Strukturen und können daher nicht nur Besetzt und Frei halten. Hierzu wird beispielsweise bei dem Eintritt in einen kritischen Bereich die interne Zählervariable dekrementiert, bei Verlassen des kritischen Abschnittes wieder inkrementiert. Im Vergleich zu den Mutexen ermöglicht dies, eine durch den Programmfluss definierte Anzahl an Threads in einem kritischen Abschnitt eintreten zu lassen. Dies kann nützlich sein, falls mehrere Betriebssystemressourcen zur Verfügung stehen, diese aber nicht überschritten werden dürfen.

Ein weiterer Vorteil der Semaphoren ist, dass eine Freigabe nicht von dem selben Thread durchgeführt werden muss (siehe hierzu Unterabschnitt 4.2.3), wodurch sich auch einfache Benachrichtigungssysteme implementiert lassen. Hierzu folgen weitere Details in einer der nächsten Übungen.

Diese POSIX-Semaphoren gibt es in zwei Formen, die sich nur in der Art der Erzeugung unterscheiden: benannte Semaphoren und unbenannte Semaphoren.

4.3.1 unbenannte Semaphoren (memory-based Semaphoren)

Die unbenannten Semaphore[1] erzeugt ihre internen Informationen in einem Speicherbereich platziert, der von mehreren Threads (oder Prozessen) gemeinsam genutzt wird.

[1] Das Betriebssystem macOS des Unternehmens Apple ist zwar ein POSIX-kompatibles Betriebssystem, dennoch werden die unbenannten Semaphoren nicht unterstützt. Die Instruktion `sem_open` wird vom Compiler als 'deprecated' klassifiziert und die Ausführung führt immer zu einer fehlerhaften Rückgabe (Stand Mai 2023). Daher muss unter macOS auf die benannten Semaphoren zurückgegriffen werden.

Zum Erstellen und Löschen dieser unbenannte Semaphore werden die beiden Funktionen in Listing 4.7 verwendet. Hierzu wird der Zeiger auf ein Handle des Typs `sem_t` übergeben. Bei der Initialisierung kann zudem noch über die variable pshared angegeben werden, ob die Semaphore nur zwischen lokalen Threads (Null) oder prozessübergreifend verwendet wird (beliebig, ungleich Null). Der letzte Parameter `value` gibt den initialen Wert der Semaphore an. Da es sich bei Semaphoren um zählende, dekrementierende Systeme handelt, muss der initiale Wert für die binäre Synchronisation von kritischen Abschnitten Eins betragen.

```
1 int sem_init(sem_t *sem, int pshared, unsigned int value);
2 int sem_destroy(sem_t *sem);
```

Listing 4.7: (De-)Initialiseren von unbenannten Semaphoren

4.3.2 Benannte Semaphoren

Ein benannter Semaphor wird durch einen Namen der Form `/beliebigername` identifiziert, d. h. eine nullterminierte Zeichenkette, die aus einem Schrägstrich, gefolgt von einem oder mehreren Zeichen besteht. Weitere Schrägstriche dürfen nicht enthalten sein. Zwei unabhängige Threads oder Prozesse können auf dieselbe benannten Semaphor zugreifen, indem sie den gleichen Namen übergeben und müssen das Handle nicht über einen gemeinsamen Speicherbereich austauschen. Zum Erstellen, Schließen und Löschen der benannten Semaphore werden die Funktionen in Listing 4.8 verwendet.

Mithilfe von `sem_open` werden die benannten Semaphoren erzeugt oder eine bestehende bestehende Semaphore geöffnet. Dabei wird die Semaphore anhand ihres eindeutigen Namen identifiziert. Als zweiter Parameter werden die sogenannten `oflags` aus `<fcntl.h>`, die bestimmen, ob eine neue Semaphore erstellt wird, oder nur geöffnet. Mit `O_CREAT` wird angegeben, dass eine neue Semaphore mit dem übergebenen Namen erstellt wird, falls sie noch nicht vorhanden ist. Soll diese exklusiv erstellt werden und den Nutzer informieren, falls diese schon existiert, muss zusätzlich das Flag `O_EXCL` übergeben werden. Über `mode` werden noch die Nutzer- und Gruppenrechte der Semaphore vergeben, die identisch mit den Dateiberechtigungs-Bits jeglicher Dateien unter UNIX sind. Der werd `value` gibt hier wieder den Initialwert der Semaphore an.

Wird die Semaphore von dem aktuellen Thread oder Prozess nicht mehr benötigt, muss diese mit `sem_close` geschlossen werden, um die lokalen Ressourcen wieder frei zu geben. Die Semaphore bleibt nach dem schließen noch im Betriebssystem vorhanden und kann weiterhin über den zuvor vergebenen Namen angesprochen werden. Um die Semaphore endgültig aus dem Betriebssystem zu entfernen müssen auch diese Ressourcen freigegeben werden. Dazu muss `sem_unlink` mit dem Namen der Semaphore aufgerufen werden. Die Semaphore wird nun endgültig gelöscht, sobald alle weiteren Threads oder Prozesse, die die Semaphore geöffnet haben, diese schließen.

```
1 sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
2 int sem_close(sem_t *sem);
3 int sem_unlink(const char *name);
```

Listing 4.8: (De-)Initialiseren von benannten Semaphoren

4.3.3 Sperren und Freigeben

Das Freigeben bzw. das Inkrementieren der internen Zählervariable wird mithilfe der Funktion in Listing 4.9 realisiert. Dieser wird dazu ein Zeiger auf ein Semaphor-Handle übergeben, das zuvor initialisiert wurde.

```
1 int sem_post(sem_t *sem);
```

Listing 4.9: Inkrementieren einer Semaphore

Für das Blockieren bzw. Dekrementieren der Semaphore, werden die drei Funktionen aus Listing 4.10 angeboten. Diese unterscheiden sich jedoch in ihrem Blockierverhalten, falls die interne Zählervariable Null ist. `sem_wait` blockiert so lange, bis ein anderer Thread die Semaphore wieder frei gibt, wodurch es beispielsweise auch zu Deadlocks führen kann. Soll nur überprüft werden, ob die Semaphore dekrementiert werden kann, andernfalls aber nicht blockiert werden muss, da beispielsweise andere Tätlichkeiten ausgeführt werden können, so bietet sich die Funktion `sem_trywait` an. Eine weitere Möglichkeit, die eine Mischung aus der ersten un zweiten Funktion, bietet die Funktion `sem_timedwait`. Diese wartet im Regelfall auf die Freigabe der Semaphore, dennoch bietet sie die Möglichkeit nach einem abgelaufenen Timeout abzubrechen. Hierdurch wird ermöglicht, bei Fehlern die Sperrung abzubrechen und somit eine Fehlerbehandlung zu realisieren. Die maximale Zeitspanne wird dazu als `struct timespec` Struktur übergeben, die Sie schon bei der `clock_gettime` Funktion kennengelernt haben.

```
1 int sem_wait(sem_t *sem);
2 int sem_trywait(sem_t *sem);
3 int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
```

Listing 4.10: Dekrementieren einer Semaphore

Alle hier genannten Funktionen der Semaphoren geben bei fehlerfreier Ausführung eine `0` zurück und im Fehlerfall `-1`. Der tatsächlich aufgetretene Fehler wird in der globalen Fehlervariable `errno` abgespeichert. Dieses Verfahren ist unter Linux/POSIX die übliche Art, Fehler einheitlich zurückzugeben. Dieses Verfahren ist generell bei nahezu allen System-Funktionen, aber auch vielen Bibliotheksfunktionen implementiert.

4.4 Weitere Methoden

Neben den oben gezeigten Funktionalitäten zur Synchronisierung bietet die pthreads-Bibliothek weitere Möglichkeiten, um parallele Arbeiten zu synchronisieren.

4.4.1 Condition-Variablen

Mithilfe der Condition-Variablen lässt sich ein Programmablauf synchronisieren. Dazu wartet ein Thread auf das Eintreten einer vorher bestimmten Bedingung (Event), die durch einen weiteren Thread initiiert wird. Die Funktionen der Condition-Variablen beginnen mit dem Präfix `pthread_cond_`

4.4.2 Barrier

Unter Barrieren versteht man ein Synchronisierungsmittel, die erst überwunden werden kann, wenn eine zuvor definierte Anzahl an Threads auf die Freigabe dieser Barriere warten. Die Funktionen der Barriers beginnen mit dem Präfix `pthread_barrier_`.

4.4.3 Spinlocks

Spinlocks bieten neben den Mutexen eine weitere Möglichkeit zum Absichern des gegenseitigen Ausschluss. Im Gegenzug zu den Mutexen eignen sich Spinlocks nur auf Multiprozessorsystemen, da sie die CPU nicht freigeben, sondern mithilfe von so genannte busy loop auf die Freigabe der Ressource warten. Hierdurch erspart man sich einen Kontextwechsel und dadurch auch das Auslagern der Thread-internen Zustände.

Die Funktionen der Spinlocks beginnen mit dem Präfix `pthread_spin_`.

4.4.4 RW-Locks

Die Read-Write-Locks bieten eine ähnliche Funktionalität wie Mutexe, jedoch ist das mehrfache Lesen zugelassen. Hierdurch können beliebige Threads gleichzeitig eine geteilte Datenstruktur lesen, aber nur ein einziger Thread zur selben Zeit beschreiben.

Die Funktionen der RW-Locks beginnen mit dem Präfix `pthread_rwlock_`.

4.4.5 Atomare Funktionen

Atomare Funktionen erlauben einfache Manipulationen von Daten, die auch bei paralleler Ausführung korrekt funktionieren. In C werden diese durch `#include <stdatomic.h>` bereitgestellt, in C++ mittels `#include <atomic>`. Beide Bibliotheken bieten ähnliche Funktionalitäten.

Ein Beispiel ist die C-Funktion `atomic_fetch_add(int * ptr, int value)`, welche zu der Variablen, auf die ptr zeigt, den value hinzuzählt und das Ergebnis wieder an derselben Stelle ablegt. Dies geschieht atomar, d.h. wird nicht korrupiert wenn mehrere Threads gleichzeitig auf die Variable zugreifen.

Folgendes Beispiel verdeutlicht das:

```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include <stdatomic.h>
4
5  // globale Variablen
6  int counter = 0, counter2 = 0;
7
8  void* worker_func(void*)
9  {
10   for( int i = 0; i < 1000000; i++ ) {
11     // unprotected increment
12     counter++;
13     // protected increment
14     atomic_fetch_add( &counter2, 1);
15   }
16   return 0;
17 }
18
19 int main()
20 {
21   pthread_t threads[8];
```

```
22
23    for( int i=0; i<8; i++ ) {
24      pthread_create( &threads[i], 0, &worker_func, 0 );
25    }
26    // Warte bis alle Threads fertig sind
27    for( int i=0; i<8; i++ ) {
28      pthread_join( threads[i], 0 );
29    }
30
31    printf("Unprotected counter: %d\n", counter );
32    printf("Protected counter: %d\n", counter2 );
33  }
```

Listing 4.11: Beispiel für atomare Funktionen

Die Ausgabe ist dann zum Beispiel (die erste Zahl ist bei jedem Durchlauf anders):

```
1     user@debian:~$ ./a.out
2     Unprotected counter: 2207283
3     Protected counter: 8000000
```

Listing 4.12: Beispielausgabe ohne bzw. mit atomarer Funktion)

4.5 Übungsaufgaben

Die nachfolgende Übungsaufgabe zeigt beispielhaft die Problematik, die durch einen unsynchronisierten Zugriff auf eine geteilte Ressource entsteht. Zudem sollen Möglichkeiten gefunden werden, die mit den zuvor eingeführten Methoden den Zugriff abzusichern.

4.5.1 Primitive Implementierung

Als Beispielfunktionen sollen zwei individuelle Funktionen dienen, die auf einer gemeinsamen globalen Variable Operationen durchführen. Die erste Funktion inkrementiert eine zuvor definierte globale Variable eine vordefinierte, zuvor übergebene Anzahl an Iterationen. Die zweite Funktion ist nahezu identisch jedoch dekrementiert sie die Variable.

Als obere Grenze der Iterationen können Sie beispielsweise folgenden Wert annehmen, bei dem Sie die Effekte beobachten können:

```
const size_t MAX_CNT = 10000000;
```

Starten Sie die beiden Funktionen je mit mindestens einem Thread, um die globale Variable parallel zu inkrementieren und dekrementieren.

Bevor Sie die Threads starten, initialisieren Sie die globale Variable mit Null. Nach erfolgreicher Beendigung der zählenden Threads überprüfen Sie den Wert der globalen Variable.

Was fällt Ihnen bei der Überprüfung auf? Fällt der Wert bei wiederholtem Ausführen der Applikation immer gleich aus?

4.5.2 Schützen des kritischen Abschnittes

Korrigieren Sie die primitive Implementierung, indem Sie den minimal nötigen Abschnitt mithilfe der vorher besprochenen Synchronisierungsmechanismen absichern. Als Übungszweck implementieren Sie sowohl die Semaphoren als auch die Variante mit den Mutexen. (Wenn Sie in C mit POSIX arbeiten, können Sie die statische und die dynamische Variante der Mutex ausprobieren).

4.5.3 Erweiterung des Collatz-Problem

Da die Übungsaufgabe keine wirkliche Berechnung mit den Werten aus dem kritischen Bereich durchführt, verursacht die Sicherungsmethoden einen vielfachen an Overhead und ist in solchen Fällen in Realanwenungen auch nicht sinnvoll.

Um einen wirklichen Vorteil der Mutex zu erhalten, kann auf die Aufgabe der letzten Übung (Übung Betriebssysteme – Threads) zurückgegriffen werden. Erweitern Sie die Übung erneut und bestimmen Sie anhand einer globalen Variable, welchen Startwert der Collatz-Folge der jeweilige Thread als Nächstes zu berechnen hat.

Alternativ implementieren Sie das Collatz-Problem, indem Sie atomare Funktionen zum Hochzählen des aktuellen Startwertes verwenden.

Errechnen Sie erneut den Speedup-Faktor (für Mutex und atomare Funktion) und vergleichen Sie ihn mit den vorherigen Werten.

4.6 Ergebnis und Abnahme

Um das Testat zu bekommen, müssen folgende Ergebnisse vorgezeigt werden bzw. folgende Fragen beantwortet werden:

1. Erklären sie die unterschiedlichen Ergebnisse bei mehrfacher Ausführung.
2. Was fällt bei der implementierung mit Synchronisierungsmechanismen auf?
3. Berechnen Sie den SpeedUp-Faktor für 2 bis 16 Threads mit Synchronisierungsmechanismen (Mutex und atomare Funktion).

Exkurs zur Gültigkeit von Variablen

Globale Variablen

Bei einer globalen Variable handelt es sich um eine Variable, auf die von jeder Funktion zugegriffen werden kann und nicht in einem Block definiert wird.

Dadurch behalten sie ihre Gültigkeit über die gesamte Laufzeit der Applikation und werden somit auch nicht nach Verlassen eines Blocks ungültig.

Der Nachteil hierbei ist, dass die Variable im Gegenzug zu lokalen Variablen keine Schutzmechanismen besitzt und jeder Thread innerhalb des gleichen Prozesses Änderungen vornehmen kann.

```
1  int counter = 0;
2
3  void cntOne() {
4    counter += 1;
5  }
6
7  void cntTwo() {
8    counter += 2
9  }
10
11 int main() {
12   counter += 5;
13   printf("Aktueller Wert: %d\n", counter);
14   cntOne();
15   printf("Aktueller Wert: %d\n", counter);
16   cntTwo();
17   printf("Aktueller Wert: %d\n", counter);
18   return 0;
19 }
```

Statische Variablen

Neben den globalen variablen, die über Blockgrenzen ihre Werte behalten gibt es zudem statische Variablen, die innerhalb einer Funktion die Werte auch nach Verlassen des Blocks behalten. Diese Variablen werden neben ihrem Datentyp mit dem Schlüsselwort static gekennzeichnet, wodurch dieser ein fester Speicherbereich reserviert wird.

Eine Initialisierung der Variable wird nur einmalig bei Programmstart vorgenommen und behält während der Laufzeit den zuletzt zugewiesenen Wert. Die Variable kann aber, im Gegenzug zu globalen Variablen, nur durch die Funktion verändert genutzt werden. Dennoch muss bei der Nutzung von statischen Variablen darauf geachtet werden, dass es sich hier um einen kritischen Abschnitt handelt. Diese muss bei der Verwendung von mehreren Threads entsprechen abgesichert werden, um entsprechend Inkonsistenzen zu verhindern.

```
1  void cnt() {
2    static int counter = 0;
3    counter += 1;
4    return counter;
5  }
6
7  int main() {
8    printf("Aktueller Wert: %d\n", cnt());
9    printf("Aktueller Wert: %d\n", cnt());
10   printf("Aktueller Wert: %d\n", cnt());
11   return 0;
12 }
```

5 Kooperation und Konkurrenz

5.1 Einführung

In der letzten Übung wurden die Grundlagen zur Synchronisierung von Threads dargestellt und das Konzept des kritischen Abschnittes besprochen. Das Problem war hier, dass durch den konkurrierenden Zugriff auf eine gemeinsame Ressource Inkonsistenzen entstehen können und daher die Benutzung der Betriebsmittel synchronisiert werden musste.

Neben dieser Konkurrenzsituation können Threads auch kooperativ miteinander arbeiten. Hierzu werden Ressourcen von einem (oder auch mehreren) Thread zur Verfügung gestellt, die von weiteren benötigt werden, ohne die sie nicht weiterarbeiten können. Hierzu muss nicht (nur) das Betriebsmittel synchronisiert werden, sondern auch der Programmfluss.

Ein typisches Beispiel sind hier Erzeuger-Verbraucher-Systeme, bei denen Daten vom Erzeuger in einen gemeinsamen Puffer schreiben und von den Verbrauchern gelesen werden, vgl. Abbildung 5.1. Damit die Verbraucher nicht unnötig prüfen müssen, ob noch Daten in dem Puffer zur Verfügung stehen, signalisiert der Erzeuger nach Bereitstellung neuer Elemente die Verfügbarkeit. Sollte der gemeinsame Puffer zusätzlich in der Größe[1] beschränkt sein, so ist eine Signalisierung in beide Richtungen erstrebenswert, um nicht unnötig Ressourcen zu verbrauchen.

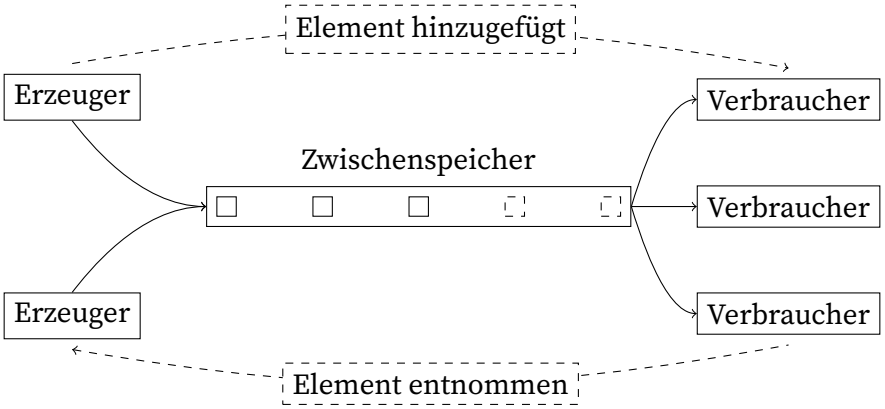

Abbildung 5.1: Erzeuger-Verbraucher Systeme

Häufig treten Konkurrenz und Kooperation kombiniert auf, da sowohl ein Informations-Kanal sowie der Schutz von kritischen Abschnitten verwenden werden muss, um miteinander zu kommunizieren.

Für diese Benachrichtigungssysteme stellen die Betriebssysteme verschiedene Lösungen zur Verfügung, die entsprechend der Anforderung verwendet werden können.

[1] Hier ist eine künstliche Größenlimitierung gemeint, die nicht unbedingt den physikalischen Grenzen, wie beispielsweise den maximalen Hauptspeicher, beruht.

5.2 Condition-Variablen

Mithilfe der Condition-Variablen der pthread-Bibliothek lässt sich Programmablauf synchronisieren, indem Tasks auf ein Ereignis warten, das durch einen andere gesetzt wird. Zudem wird die Freigabe noch an eine Mutex geknüpft, mir der ein nachfolgender Bereich auch noch vor wechselseitigem Ausschluss gesichert werden kann.

Dazu werden wartende Threads bis zur Freigebe eines Ereignisses schlafen gelegt, um den Prozessor freizugeben. Wird eine Condition-Variable freigegeben, ohne das auf diese gewartet wird, bleibt deren Ereignis ohne Wirkung und wird auch nicht für zukünftige Abfragen gespeichert.

Wie schon auch bei den Mutex bieten die Condition-Variablen statische und dynamische Initialisierungsmöglichkeiten an. Statische Initialisierung wird mithilfe der Konstante, wie in Listing 5.1 Zeile 1 dargestellt, vorgenommen und muss auch nicht wieder gelöscht werden. Dynamische Initialisierung und Deinitalisierung wird mit den entsprechenden Funktionen, wie in Listing 5.1 Zeile 2 bis 4 dargestellt, durchgeführt. Bei der Initialisierung kann neben dem Zeiger auf ein Handle vom Typ `pthread_cond_t`, zusätzlich entweder `NULL` für Default-Einstellung oder ein Attributs-Objekt angegeben werden. Hierbei können wieder einige Einstellungen vorgenommen werden, wie z.B. die Möglichkeit des prozessübergreifenden Verwendens der Condition-Variable.

```
1 pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
2 int pthread_cond_init(pthread_cond_t *restrict cond, const pthread_condattr_t *attr);
3 int pthread_cond_destroy(pthread_cond_t *cond);
```

Listing 5.1: (De-)Initialiserungsmöglichkeiten von Condition-Variablen

Zur Generierung eines Ereignisses stehen zwei verschiedene Funktionen zur Verfügung: Die Signal-Funktion (vgl. Listing 5.2, Zeile 1) wir dazu verwendet, einen einzelnen der wartenden Threads aufzuwecken, wohingegen die Broadcast-Funktion (vgl. Listing 5.2, Zeile 2) alle an der Condition-Variable wartenden Threads aufweckt.

```
1 int pthread_cond_signal(pthread_cond_t *cond );
2 int pthread_cond_broadcast(pthread_cond_t *cond );
```

Listing 5.2: Generierung eines Ereignisses

Um als Thread auf eine Condition-Variable zu warten stehen die beiden Funktionen in Listing 5.3 zur Verfügung. Diese unterscheiden sich nur durch das eventuelle Timeout-Kriterium, das sich zusätzlich als `abstime` angeben lässt. Vor dem Aufruf der Funktion muss zudem die Mutex, die als zweiter Parameter übergeben wird, gesperrt werden. Der Aufruf der entsprechenden `wait`-Funktione wird die Mutex atomar, sprich unterbrechbar und ohne Deadlocks zu verursachen, freigegeben und in Kombination mit dem Schlafenlegen des Threads durchgeführt. Bei erfolgreicher Signalisierung und der Möglichkeit des Sperrens der Mutex wird der Thread wieder geweckt und das Event kann in dem zugehörigen kritischen Abschnitt abgearbeitet werden.

5.3 Semaphore

Die generelle Verwendung der Semaphoren wurde bereits in der Übung zur Synchronisierung vorgestellt, daher wird auf eine erneute Beschreibung der einzelnen Funktionen verzichtet.

```
1 int pthread_cond_wait( pthread_cond_t *cond, pthread_mutex_t *mutex );
2 int pthread_cond_timedwait(pthread_cond_t *cond,
3                            pthread_mutex_t *mutex, const struct timespec *abstime);
```

Listing 5.3: (De-)Initialiserungsmöglichkeiten von Condition-Variablen

Im Gegenzug zu den Mutexen, lassen sich Semaphoren auch dazu verwenden, eine Synchronisierung des Programmflusses durchzuführen. Dazu wartet ein Thread, ähnlich der Condition-Variable, mithilfe von `sem_wait()` auf die Freigabe der Semaphore. Des Signalisieren durch die Semaphore wird durch einen weiteren Thread initiiert, indem er diese mit `sem_post()` freigibt.

Bedingt durch den zählenden Charakter der Semaphoren, wird im Gegenzug zu den vorher eingeführten Condition-Variablen eine Freigaben zwischengespeichert, falls keiner auf die Semaphore wartet. Ein späteres `sem_wait()` wird dadurch sofort freigegeben und nur die interne Zählervariable dekrementiert.

5.4 Übungsaufgabe

5.4.1 Erzeuger-Verbraucher-Systeme (Testat)

Das in der Einführung genannten Beispiel mit dem gemeinsamen Puffer, der von Erzeugern befüllt und Verbrauchern geleert wird, lässt sich über eine einfach verkettete Liste darstelle, die als FIFO genutzt wird. Hierbei sind sowohl geteilte Ressourcen als auch der Programmfluss synchronisiert werden.

Einfach verkettete Listen

Implementieren Sie hierzu eine einfache verkettete Liste (Abbildung 5.2), die als Payload einen Integer einhält. Als Zugriffsfunktionen benötigen Sie die Möglichkeit, an das Anfang der Liste ein neues Element zu hängen, und am Ende der Liste ein Element zu entfernen. Als Wiederholung zu verketteten Listen können Sie beispielsweise in [**wolf2009**] nachlesen.

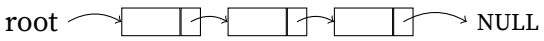

Abbildung 5.2: Einfach verkettete Liste

Verwalten der verketteten Liste

Um die verkettete Liste von mehreren Threads zu verwenden, überlegen Sie sich, wo Synchronisierungsmechanismen zwingend notwendig sind und wo sich der Einsatz zusätzlich empfiehlt.

Bedenken Sie hier auch, dass es eleganter ist, die Entnahme der Elemente nur vorzunehmen, solange die Liste gefüllt ist. Sollte die Liste leer sein, ist es Ressourcenverschwendung, pollend auf die Struktur zuzugreifen. Besser ist es hier, die aktuelle Anzahl an Elementen zu zwischenzuspeichern und sich darüber zu synchronisieren.

Producer- und Consumer-Threads

Implementieren Sie je eine Funktion, die zyklisch Daten in die Liste hinzufügt und aus der Liste entfernt. Neben dem Hinzufügen und Entfernen soll die Funktion auch weitere Arbeit simulieren. Dies können Sie entweder durch ein zufälliges Pausieren realisieren oder durch eine tatsächliche Berechnung. Hier können Sie wieder das Collatz-Problem verwenden, dem Sie eine zufällige Zahl übergeben. Als Payload übergeben Sie die zufällig erzeugte Zahl und gebe sie diese bei den Consumer-Funktionen aus.

Final instantiieren Sie mehre mehrere Threads mit den Producer und den Consumer-Funktionen.

Zufallszahlen

Als Zufallsvariable können Sie die Funktion `int rand(void)` verwende. Hierbei handelt es sich um einen einfachen linearen Kongruenzgenerator, der nur bedingt zufällig ist und eine kurze Periodenlänge hat. Für ernsthafte Anforderungen an Zufallszahlen, wie z.B. Kryptographie oder statistische Simulation, ist er daher nicht geeignet. Initialisieren Sie den Startewert (Seeds, `void srand(unsigned seed)`) für den Generator innerhalb jedes Threads mit einem individuellen Wert, da sonst beide Threads die gleiche Serie an Zufallsvariablen bilden. In der Literatur (und auch im Internet) liest man häufig, dass man den aktuellen Zeitpunkt als Seed verwenden soll und schlagen daher `srand(time(0))` vor. Dies ist für diese Implementierung ungeeignet, da time(0) den aktuellen Zeitstempel in Sekunden wiedergibt. Die Initialisierung der Threads wird jedoch meist innerhalb der gleichen Sekunde vorgenommen und daher ist der Seed identisch. Für unseren Fall können sie entweder die `pthread_self()` verwenden oder der Nanosekunden-Anteil von `clock_gettime`.

5.5 Ergebnis und Abnahme

Um das Testat zu bekommen, müssen folgende Ergebnisse vorgezeigt werden bzw. folgende Fragen beantwortet werden:

1. Die verkettete Liste ist implementiert und ein pollender Zugriff wird vermieden.
2. Die Producer-Funktion benötigt für verschiedene Durchläufe unterschiedlich lang.
3. Es lässt sich eine vorgegebene Anzahl an Producer und Consumer erzeugen.
4. Nachdem die Producer eine feste Anzahl von Elementen produziert haben und alle Elemente von den Consumern verarbeitet wurden, endet das Programm. Achten Sie darauf, dass alle Threads ordentlich beendet werden, keine Threads “hängen” bleiben und alle allokierten Elemente ordentlich freigegeben werden.
5. Verwenden Sie zur Synchronisierung ausschließlich Semaphoren. Lösungen mit Condition Variables werden NICHT akzeptiert!

6 Shared Memory

6.1 Einführung

In den bisherigen Übungen wurden ausführlich die Möglichkeiten der Parallelisierung und Synchronisierung mithilfe von Threads dargestellt. Dadurch, dass mehrere Threads sich den gemeinsamen Speicherbereich des zugrundeliegenden Prozesses teilen, ist der Austausch von Daten mithilfe von globalen Speicherbereichen, wie beispielsweise allokierten Strukturen auf dem Heap, einfach zu realisieren.

Mitunter kann es jedoch vorkommen, dass die Nutzung von Threads nicht erwünscht ist und die Parallelisierung mithilfe von mehreren Prozessen durchgeführt wird. Hier ist es nun nicht mehr direkt möglich, Daten über den gleichen Speicherbereich auszutauschen, da die einzelnen Prozesse in individuellen, virtuellen Speicherbereichen laufen. Dennoch ist es vielfach nötig, Daten über einen gemeinsamen Speicherbereich auszutauschen oder sich zu synchronisieren. Bei den Synchronisierungsmechanismen wurde schon mehrfach darauf hingewiesen, dass diese so konfiguriert werden, dass sie über Prozessgrenzen verwendet werden können. Hierzu muss beispielsweise bei der Initialisierung einer Semaphore der zweite Parameter (`pshared`) mit einem Wert ungleich `0` gesetzt werden. Um jedoch die zugrundeliegende Datenstruktur der Semaphore über die Prozessgrenzen zu verteilten, muss ein gemeinsamer Speicher verwendet werden. Dazu bieten nahezu alle Betriebssysteme sogenannte Shared Memory Mechanismen an, über die auf bestimmte Teile des Hauptspeichers von mehreren Prozessen zugegriffen werden, ohne jedoch die eigentliche Virtualisierung des Speichers zu umgehen.

Um dies zu realisieren, wird der physikalische Speicher für jeden Prozess, individuell in dessen virtuellen Speicher zuordnet (memory mapping). Dadurch kann die jeweilige Startadresse für die einzelnen Prozesse abweichen, wodurch eine Referenzierung innerhalb des Shared Memory Bereiches immer über den Offsets zu der individuellen Startadresse geschehen muss.

6.2 Shared Memory

Die Shared Memory Objekte unter POSIX/Linux werden mithilfe eines Namens angesprochen, wodurch eine einfache und prozessübergreifende Identifizierung mithilfe festgelegter Konstanten möglich ist.

6.2.1 Verwalten eines Shared Memory

Die grundlegende Verwaltung von Shared Memory Objekten mithilfe der POSIX-API ähnelt die der File-API. Sie lassen sich dadurch erstellen, öffnen, schließen und nach finalen Gebrauch auch wieder aus dem System entfernen, wodurch der Inhalt unwiderruflich gelöscht wird.

Zum Erstellen oder Öffnen eines Shared Memory Objektes wird der Befehl Listing 6.1, Zeile 1 verwendet. Dieser verhält sich analog zu `open()`, der für das Öffnen von Dateien verwendet wird.

```
1 int shm_open(const char *name, int oflag, mode_t mode);
2 int ftruncate(int fd, off_t length);
```

Listing 6.1: Erstellen von Shared Memory Objekten

Als erster Parameter wird hier der Name des Shared Memory Objektes übergeben. Aus Portabilitätsgründen sollte dieser durch die Form `/name` identifiziert werden. Diese null-terminierte Zeichenkette (C-String) darf aus bis zu 255 (`NAME_MAX`) Zeichen bestehen, die mit einem Schrägstrich beginnt, gefolgt von einem oder mehreren Zeichen, ungleich dem Schrägstrich.

Die `oflags` geben die Eigenschaften und Aktionen beim Erstellen des Shared Memory Objektes an. Dabei werden die Zugriffseigenschaften festgelegt, die entweder mit `O_RDONLY` für einen reinen lesenden Zugriff oder mit `O_RDWR` für zusätzlichen schreibenden Zugriff gesetzt wird. Zusätzlich zu den Zugriffsfreigaben kann mit dem Parameter `O_CREAT` definiert werden, dass ein neues Shared Memory Segment der Länge Null angelegt wird, falls es noch nicht vorhanden ist. Wird zusätzlich zu `O_CREAT` die Option `O_EXCL` übergeben, so wird das Segment angelegt oder bei vorherigem Vorhandensein wird der Funktionsaufruf mit einem Fehler quittiert. Als weiterer Zusatz zu `O_CREAT` kann `O_TRUNC` übergeben werden, wodurch ein vorher vorhandenes Segment auf die Länge Null gekürzt wird.

Über die `oflags` lassen sich Zugriffsrechte setzten, die Kompatible zu den vorher angegebenen seien müssen. Hierbei kann entweder `S_IRUSR` für reinen lesenden Zugriff oder in Kombination mit `S_IWUSR` für lesenden und schreibenden Zugriff angeben werden. Falls dies nicht explizit angegeben werden soll, kann auch `NULL` übergeben werden, wodurch Default-Rechte angenommen werden.

Der Rückgabewert ist ein File-Descriptor, der sich identisch mit einem File-Descriptor von `open` verhält. Sollte ein Fehler auftreten wird `-1` zurückgegeben und die `errno` gesetzt, worüber sich weitere Informationen zu dem Fehler auslesen lassen.

Nach dem Erstellen eines neuen Shared Memory Objektes wird dies mit der Größe von $0\,\mathrm{B}$ erstellt. Um den Inhalt an die gewünschte Größe anzupassen, muss der verfügbare Bereich vergrößert werden. Dies lässt sich mit der Funktion `ftruncate` aus Listing 6.1, Zeile 2 realisieren, die Größe der übergebenden Datei auf die gewünschte Länge setzt. Falls die Datei vorher größer war, geht ein Teil der Daten verloren. Andersherum wird sie vergrößert und der zusätzliche Teil wird mit Null-Bytes (`'\0'`) aufgefüllt.

```
1 int close(int fd);
2 int shm_unlink(const char *name);
```

Listing 6.2: Schließen und Löschen von Shared Memory Objekten

Wird die Verbindung zu dem File-Descriptor des Shared Memory Objektes nicht mehr benötigt, lässt sich diese mit der allgemeinen File-Operation `close()` aus Listing 6.2 beenden. Dazu wird der gewünschte File-Descriptor übergeben. Der Inhalt des Shared Memory Objektes bleibt nach Schließen des File-Descriptor jedoch erhalten.

Final lässt sich mit `shm_unlink` aus Listing 6.2 und dem Namen das Shared Memory Objekte löschen. Dadurch werden alle Informationen, die innerhalb des Speicherbereichs liegen gelöscht und sind nicht mehr nutzbar.

6.2.2 Zuordnung des Shared Memory

Mithilfe des File-Descriptor lässt sich aktuell nur sequenziell Daten aus dem Shared Memory zugreifen. Um adressorientiert auf dem Objekt arbeiten zu können, muss dieser in den virtuellen Speicher des jeweiligen Prozesses gemapped werden. Ein weiterer Vorteil dieser Methode ist, dass hierbei weniger Systemaufrufen durchgeführt werden müssen, wodurch auch der Wechsel zwischen User-Space und Kernel-Space vermieden werden.

```
1 void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
2 int munmap(void *addr, size_t length);
```

Listing 6.3: Speicherzuordnung eines File-Descriptor

Um einen File-Descriptor in den Speicher abzubilden, wird auf die Funktion `mmap()` aus Listing 6.3 zurückgegriffen. Diesem Befehl können neben dem hier verwendeten Shared Memory Objekts auch File-Descriptors für Dateien oder Geräte übergeben werden. Die zu übergebenden Parameter haben folgende Bedeutung:

`addr` Über die `addr` kann ein Vorschlag für die Adressregion angegeben werden, auf den der Speicher gemappt werden soll. Wird `NULL` übergeben, so wird ein zufälliges Segment zurückgegeben. Wird hier ein wert übergeben, muss dieser nicht zwingend übernommen werden, sondern das Betriebssystem wähle einen für ihn passendes Segment in der Nähe aus. Dies kann sich in unterschiedlichen Prozessen voneinander abweichen.

`length` Größe des Objektes, das in den Speicher abgebildet werden soll.

`prot` Mit `prot` werden die Schutzmechanismen festgelegt. Dazu lassen sich mit `PROT_READ` für Leserechte, `PROT_WRITE` für Schreibrechte, `PROT_EXEC` für ausführbar und `PROT_NONE` für nicht zugreifbare Bereiche angeben. Diese Schutzart muss mit den entsprechenden Optionen übereinstimmten, die beim Öffnen des File-Descriptor verwendet wurde.

`flags` Mithilfe von `flags` lassen sich Eigenschaften des geöffneten Segments setzten. Dabei kann mithilfe von `MAP_SHARED` für über Prozessgrenzen verwendeten Speicher oder `MAP_PRIVATE` lokale Adressbereiche definiert werden.

`fd` Der File-Descriptor, der in den Speicher abgebildet werden soll.

`offset` Über den `offset` kann der Beginn in dem Dateisegment angegeben werden.

Als Rückgabewert erhält man einen individuellen Zeiger auf den Speicher, auf dem die Datei gemappt wurde. Sollte ein Fehler auftreten wird `MAP_FAILED` (entspricht dem Wert `(void *)-1`) zurückgegeben und die Ursache des Fehlers wird in `errno` hinterlegt.

Wird das Segment nicht mehr benötigt, lässt sich das Speicher-Mapping mit `unmap()` aus Listing 6.3 löschen.

6.2.3 Verwendung der Shared Memory

Die Nutzung eines Shared Memory Bereiches kann generell gleichwertig mit dem eines Speicherbereiches durchgeführt werden, dass im lokalen Bereich eines Prozesses liegt. Es kann (je nach Schutzmechanismen) via Pointer-Arithmetik beschrieben und gelesen werden, wodurch keine Systemaufrufe nötig sind.

Besondere Aufmerksamkeit muss jedoch bei Verwendung von Verzeigerung gewidmet werden, wie beispielsweise bei verketteten Listen. Hier muss das Memory Management selbst übernommen werden, da nur auf Segmente innerhalb des Objektes über Prozessgrenzen zugegriffen werden kann. Hierdurch verbietet es sich, `malloc()` und `free()` zu verwenden, da hiermit Speicherbereiche innerhalb des lokalen Heap der individuellen Prozesse verwaltet werden.

Zudem muss bei einer Verzeigerung innerhalb des Shared Memory Bereiches darauf geachtet werden, dass die Startadresse sich zwischen den Prozessen unterscheiden kann. Grundsätzlich kann zwar eine Adressregion bei der Initialisierung über `mmap()` angegeben werden, jedoch wird nicht garantiert, dass diese auch gesetzt werden kann. Um hier dennoch eine Adressierung vornehmen zu können muss diese unabhängig der Startadresse durchgeführt werden. Um dies zu gewährleisten wird häufig der Offset zu dem Segmentbeginn angegeben und in jedem Prozess die eigene Startadresse auf den Offset addiert. Hierdurch kann wieder wie mit regulären Adressen gearbeitet werden. Beim Zurückschreiben einer entsprechenden Adresse muss wiederum die Startadresse abgezogen werden und nur das Offset zurückgeschrieben werden.

6.3 Übungsaufgabe

Implementieren Sie ein einfaches Erzeuger-Verbraucher-System, bei dem zwei Prozesse auf dem gleichen Shared Memory Objekt arbeiten. Erstellen Sie hierzu zwei Programme, die ihre Daten mit den Strukturen aus Listing 6.4 synchronisiert über ein Shared Memory austauschen. Dabei wird vom Produzenten ein Wert in die Payload-Struktur als `work` geschrieben, woraufhin der Verbraucher diesen ausließt, quadriert und in `result` zurückschreibt. Dieses Wechselspiel wird mehrfach durchgeführt, um eine korrekte Funktionalität und die damit verbundene Synchronisierung nachzuweisen.

```
1  struct sync {
2    sem_t newWork;
3    sem_t newResult;
4    void * payloadOffset;
5  };
6
7  struct payload {
8    int work;
9    int result;
10 };
```

Listing 6.4: Beispielstruktur für die Übungsaufgabe

Hierzu erstellt der Produzent zu Beginn ein neues Shared Memory, indem er die beiden Strukturen ablegt. Dabei wird das `struct sync` Objekt an den Ursprung des Segments gelegt. Um den Erzeuger mitzuteilen, wo sich der Payload (`struct payload`) befindet, wird der `void *` Zeiger entsprechend gesetzt. Achten Sie hierbei darauf, dass die Startadresse des Shared Memory Objektes bei den beiden Prozessen unterschiedlich sein kann. Als nächsten Schritt wird im Produzenten noch die beiden Semaphoren initialisiert. Achten Sie darauf, dass diese für eine prozessübergreifende Verwendung initialisiert werden und überlegen Sie, wie der Initialwert der beiden Semaphoren gesetzt werden muss.

Im Nachgang implementieren Sie die Applikation für den Verbraucher. Dieser muss das Shared Memory Objekt nur noch öffnen und die darin enthaltenen Datenstrukturen verwenden. Eine Initialisierung der Werte muss nicht vorgenommen werden, da dies zuvor durch den Produzenten durchgeführt worden ist.

Nach der Initialisierung der beiden Applikationen, legt der Produzent einen neuen Wert in den Datenstruktur `struct payload`, signalisiert dies dem Verbraucher und wartet auf dessen Ergebnis. Der Verbraucher wartet seinerseits auf einen neuen Wert des Produzenten, quadriert und schreibt diesen zurück. Im Anschluss signalisiert er dem Erzeuger das neue Ergebnis. Dieses Wechselspiel wir mehrfach wiederholt. Nach einer vorgegebenen Anzahl von Iterationen wird das Wechselspiel beendet und die zuvor initialisierten und geöffneten Strukturen wieder korrekt geschlossen und entfernt.

6.4 Ergebnis und Abnahme

Um das Testat zu bekommen, müssen folgende Ergebnisse vorgezeigt werden bzw. folgende Fragen beantwortet werden:

1. Eine adressunabhängig Verwaltung der benötigten Komponenten ist implementiert
2. Der Austausch von Informationen durch verschiedenen Prozessen ist möglich.

7 Anhang: C++

7.1 Vergleich von C++ und POSIX

Einige der Aufgaben können alternativ auch in C++ unter Verwendung der C++ Standardbibliothek verwendet werden. Diese bietet folgende Elemente:

Threads Die C++ Standardbibliothek enthält Threads. Diese können direkt im Code genutzt werden, oder in eine Klasse integriert werden (als Member-Funktion oder durch Ableitung). Das C++ Thread API bietet die Möglichkeit, Threads anzulegen, zu starten, zu joinen sowie eine Parameterübergabe. Die übergebenen Parameter sind dabei typ-sicher, was ein großer Vorteil gegenüber void* ist. Die C++ Threads haben keinen Rückgabewert, jedoch können einzelne Parameter als Referenz definiert werden; damit können dann Ergebnisse aus dem Thread zurückgegeben werden. Siehe Beispiele im folgenden Kapitel.
C++ nutzt im Allgemeinen die POSIX-Schnittstelle, um Threads auf dem Betriebssystem zu verwenden. Es werden jedoch nicht alle Einstellungsmöglichkeiten der POSIX Schnittstelle unterstützt, z.B. fehlen Prozessor-Affinität, Scheduling-Strategien, etc.

Mutex C++ bietet Mutex an. Diese können alternativ zu POSIX-Mutexe genutzt werden.

Atomic Operations C++ bietet eine Reihe von Atomic-Funktionen für den geschützten Zugriff auf Datenelemente an.

Zähl-Semaphore Seit C++ 21 enthält die Standardbibliothek auch Semaphore (counting_ semaphore und binary_ semaphore); allerdings werden diese von den aktuellen Compilern nicht unterstützt. Hier sind ggfs. die POSIX-Semaphore zu verwenden.

Shared Memory Hierzu bietet C++ keine Unterstützung. Es müssen die POSIX-Schnittstellen verwendet werden.

Die folgenden Beispiele sind mit einem GCC Compiler der folgenden Version kompiliert:

```
user@linux:/home/user/cpp$ c++ --version
c++ (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Listing 7.1: ÜBersetzen mit C++

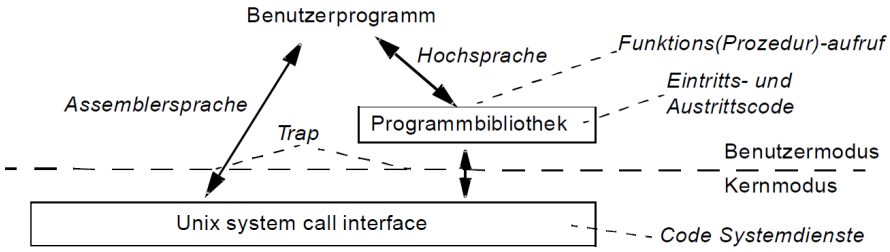

Abbildung 7.1: API vs Programmbibliothek

Funktion	POSIX	C++
Threads	pthread Threads	C++ Threads
Mutex/Lock	pthread Mutex	C++ Mutex
Condition Variables	pthread Condition Variables	C++ Condition Variables
(Zähl-)Semaphore	Linux / POSIX sem (nicht unter MacOS verfügbar)	(counting_semaphore erst ab C++20 verfügbar, wird noch nicht von den Compilern unterstützt)
Shared Memory	Linux shm API	(keine Unterstützung durch C++)

7.2 Erste C++ Beispielprogramme

7.2.1 Zufällige Buchstaben

Das folgende Programm erzeugt zwei Threads, die beide je 100mal einen zufällig ausgewählten Buchstaben ausgeben (siehe auch Kapitel 2.3.1):

```
#include <chrono>
#include <thread>
#include <iostream>

using namespace std::chrono_literals;

/* Thread-Funktion */
void example_fct(char c) {

  size_t i;
  /* Waehle einen zufaelligen Buchstaben aus */
  c = 'a' + (std::hash<std::thread::id>()(std::this_thread::get_id())) % 26;

  for(i = 0; i < 100; i++) {
    /* Gib Buchstaben aus */
    std::cout << c;
    /* Warte ein wenig; 1 microsecunde */
    std::this_thread::sleep_for(1us);
  }

  std::cout << std::endl;
  return;
}

int main() {

  /* Erzeuge 2 Threads */
  std::thread myThreadA(example_fct, 'a' );
  std::thread myThreadB( example_fct, 'b');

  /* Warte bis beide Threads fertig sind */
  myThreadA.join();
  myThreadB.join();

  return 0;
}
```

Listing 7.2: Erste C++ Beispielanwendung

Compilieren erfolgt dann mit dem C++ Compiler:

```
  user@linux:/home/user/cpp$ c++ beispiel001.cpp
```

Listing 7.3: Übersetzen mit C++

Die Ausgabe ist dann wie folgt:

```
user@linux:~/cpp$ ./a.out
tetetetetetetetetetetetetetetetetetetetetetetetetetetetetetetetetetetetetetetetetetetetetetetetetetetetete
teetetetetetetetetetetetetetetettteettetetetetetetetetetetetetetetetetetetetetetetetetetetetetetetetetetet
etetetetetetetetetetetetetetetetee
user@linux:~/cpp$ ./a.out
sspspspspspsspspppspspppsspspspsspspspspspppspspspspspsspspspspspsspspspspspspspspspspspsp
spspspspspspppspspspspspspspspppspspspspspspspsspspspspspspsspppspspspspspspspspspspspsps
pspspspspspspspspspspspspspspspspp
```

Listing 7.4: Output des Programms

7.2.2 Rückgabe-Parameter

Hier das zweite Beispielprogramm, das einen Rückgabewert mittels C++ Referenz zurückgibt (siehe auch Kapitel 2.3.3):

```
#include <iostream>
#include <thread>
#include <string>

/* Thread-Funktion: er Rückgabewert wird mittels Referenz zurückgegeben */
void example_fct(std::string theString, int& theStringLen){

  theStringLen = theString.length();

  return;
}

int main(){

  int sLen;
  std::string str = "Hallo Welt!";

  /* Erzeuge einen Thread mit 2 Übergabeparametern (typisiert) */
  /* die Referenz muss mittels std::ref in einen Referenz-Wrappter verpackt werden */
  std::thread myThread( example_fct, str, std::ref(sLen) );
  myThread.join();

  std::cout << "Länge: " << sLen << std::endl;

  return 0;
}
```

Listing 7.5: C++ Beispielanwendung mit Rückgabewert

Das Programm gibt dann aus:

```
  user@linux:/home/user/cpp$ c++ beispiel002.cpp
  user@linux:/home/user/cpp$ ./a.out
  Länge: 11
  user@linux:/home/user/cpp$
```

Listing 7.6: Übersetzen mit C++

Manchmal ist auch die Option -pthread notwendig, also

```
  user@linux:/home/user/cpp$ c++ -pthread beispiel002.cpp
```

Listing 7.7: Übersetzen mit C++

7.2.3 Liste von Threads (mittels std::vector)

Um eine Liste von Threads zu verwalten, kann man die erzeugten Threads in einem vector speichern. Allerdings kann ein Objekt vom Typ ”thread”nicht kopiert werden; hier müssen wir den move-Operator bemühen:

```
#include <iostream>
#include <thread>
#include <vector>

/* Worker-Funktion: gibt Zahl mit Verzögerung aus */
void print_i(int n ){

  std::cout << "Ich bims! Nr." << n << std::endl;
  std::this_thread::sleep_for(std::chrono::seconds(2));
  std::cout << "Ausgeschlafen! Nr." << n << std::endl;
  return;
}

int main(){

  const int anz_threads = 3;

  // Speicher für Threads
  std::vector < std::thread > threads;

  // Zeitmessung
  auto start = std::chrono::system_clock::now();

  for( int i = 0; i < anz_threads; i ++ ) {
    // Erzeuge einen Thread und speichere ihn im vector
    threads.push_back(std::thread( print_i, i ));
  }

  // Warte auf Ende der Threads
  for( auto & thr: threads )
    thr.join();

  auto stop = std::chrono::system_clock::now();
  auto elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(stop - start);
  std::cout << "Zeit: " << elapsed.count() << "ms" << std::endl;

  return 0;
}
```

Listing 7.8: C++ Programm mit mehreren Threads und std::vector (alle Probleme gelöst)

Das Programm gibt dann aus:

```
  user@linux:/home/user/cpp$
  Ich bims! Nr.0
  Ich bims! Nr.1
  Ich bims! Nr.2
  Ausgeschlafen! Nr.0
  Ausgeschlafen! Nr.2
  Ausgeschlafen! Nr.1
  Zeit: 2001ms
  user@linux:/home/user/cpp$
```

7.2.4 C++ Mutex

Eine Mutex (Mutual Exclusive) ist ein einfaches Element, welches den exklusiven Zugriff auf eine Ressource absichert. Vor dem Zugriff muss mittels der Memberfunktion `lock()` das Element gesperrt werden; die Freigabe erfolgt mittels `unlock()`. Möchte ein Thread eine Mutex sperren die schon gesperrt ist wird er angehalten, bis die Mutex wieder freigegeben wurde.

Das folgende Beispielprogramm funktioniert nicht richtig, wenn die Mutex deaktiviert ist. Schaltet man die Mutex ein, so ist das Ergebnis korrekt (jedoch dauert es dann deutlich länger).

```
#include <chrono>
#include <iostream>
#include <mutex>
#include <thread>
#include <vector>

// global mutex object
std::mutex m;
const bool use_mutex = false;
const int NUM = 8;

// global variable, used by all threads
int counter = 0;

// increases counter by max
void fun(int max)
{
  for( int i = 0; i < max; i++ ) {
    if( use_mutex )
      m.lock();
    counter += 1;
    if( use_mutex )
      m.unlock();
  }
}

int main()
{
  std::vector< std::thread > threads;

  if( use_mutex )
    std::cout << "Mutex aktiv" << std::endl;
  else
    std::cout << "Mutex NICHT aktiv" << std::endl;

  for( int i = 0; i < NUM; i++ )
    threads.push_back( std::thread( fun, 1000000 ) );

  for( auto &thr :  threads )
   thr.join();

  std::cout << counter << " (correct value: " << NUM*1000000 << ")" << std::endl;

}
```

Listing 7.9: Zählprogramm mit Mutex

Ist die Mutex aktiviert, gibt das Programm die korrekte Berechnung aus:

```
  user@linux:/home/user/cpp$ time ./a.out
  Mutex aktiv
    8000000 (correct value: 8000000)

    real    0m0.951s
    user    0m1.194s
    sys     0m5.440s
```

Ist die nicht Mutex aktiviert, ist die Berechnung fehlerhaft (aber schneller):

```
user@linux:/home/user/cpp$ time ./a.out
Mutex NICHT aktiv
1526367 (correct value: 8000000)

real    0m0.022s
user    0m0.089s
sys     0m0.000s
```

7.2.5 C++ Semaphore

Der C++20 Standard bietet zwei verschiedene Semaphore an: eine Zählsemaphore (`counting_semaphore`) und eine binäre Semaphore (`binary_semaphore`). Im Gnu-C++ Compiler sind diese Features experimentell, deshalb muss die Option `-std=c++20` oder `-std=c++2a` beim Kompilieren angegeben werden.

Ein einfaches Beispielprogramm sieht dann so aus:

```
#include <chrono>
#include <iostream>
#include <semaphore>
#include <thread>

// global binary semaphore instances
// object counts are set to zero
// objects are in non-signaled state
std::binary_semaphore<5>
    smphSignalMainToThread{0},
    smphSignalThreadToMain{0};

void ThreadProc()
{   // wait for a signal from the main proc
    // by attempting to decrement the semaphore
    smphSignalMainToThread.acquire();
    // this call blocks until the semaphore's count
    // is increased from the main proc

    std::cout << "[thread] Got the signal\n"; // response message

    // wait for 3 seconds to imitate some work
    // being done by the thread
    using namespace std::literals;
    std::this_thread::sleep_for(3s);
    std::cout << "[thread] Send the signal\n"; // message

    // signal the main proc back
    smphSignalThreadToMain.release();
}

int main()
{   // create some worker thread
    std::thread thrWorker(ThreadProc);

    std::cout << "[main] Send the signal\n"; // message

    // signal the worker thread to start working
    // by increasing the semaphore's count
    smphSignalMainToThread.release();

    // wait until the worker thread is done doing the work
    // by attempting to decrement the semaphore's count
    smphSignalThreadToMain.acquire();

    std::cout << "[main] Got the signal\n"; // response message
    thrWorker.join();
}
```

Listing 7.10: C++ Programm mit Semaphore

Das Programm gibt dann aus:

```
  user@linux:/home/user/cpp$
  [main] Send the signal
  [thread] Got the signal
  [thread] Send the signal
  [main] Got the signal
  user@linux:/home/user/cpp$
```

8 Anhang: Bonuspunkte

8.1 Parallele Berechnung des Collatz-Problems mit OpenMP (0,5 BP)

In dieser Aufgabe sollen Sie das Collatz-Problem (siehe zweite Aufgabe) mittels OpenMP parallelisieren. Dabei sollen Sie sich selbständig in das Thema OpenMP einarbeiten und eine Lösung finden.

8.1.1 Ergebnis und Abnahme

Um den Bonuspunkt zu bekommen, müssen folgende Ergebnisse vorgezeigt werden:

1. Sie haben ein Programm geschrieben, dass OpenMP Code enthält und die Berechnung mittels OpenMP auf verschiedene Threads verteilt.
2. Das Programm berechnet die längste Collatz-Folge zwischen 1 bis $100\,000\,000$ und gibt den Startwert und die Länge aus.
3. Innerhalb des Programms verwenden Sie OMP zur Parallelisierung. Geben Sie zusätzlich die Anzahl der OMP-Threads aus, die das Programm verwendet.
4. Die Abnahme erfolgt durch den Betreuer im Praktikum. Den Code laden Sie wie üblich in Moodle hoch.

8.1.2 Hinweise und Tipps

Hier einige Hinweise und Tipps, die Ihnen bei der Aufgabe helfen könnten:

1. Mehr über OpenMP können Sie z.B. hier erfahren: `https://learn.microsoft.com/de-de/cpp/parallel/openmp/reference/openmp-directives?view=msvc-170`. Das bezieht sich zwar auf den Microsoft Compiler, funktioniert aber auch mit dem GCC.
2. Beim Kompilieren mit GCC müssen Sie zusätzlich noch `-fopenmp` als Paramater angeben.
3. Verwenden Sie `#include <openmp.h>`, um OpenMP Befehle nutzen zu können (die pragmas können Sie auch ohne include nutzen).
4. Die Anzahl der Threads können Sie mit folgendem Code ermitteln (nThreads ist dabei eine int-Variable):

```
1    #pragma omp master
2    nThreads = omp_get_num_threads();
```

5. Folgende OpenMP Befehle könnten für Sie nützlich sein:

`#pragma omp parallel` Parallelisiert den folgenden Block (d.h. alles was danach in geschweifeten Klammern kommt). Hier müssen Sie ggfs. noch die Variablen als shared oder private deklarieren.
Hinweis: Threads brauchen Sie im Programm dann nicht mehr definieren, das übernimmt OpenMP für Sie.

`#pragma omp for` Damit wird die folgende for-Schleife parallelisiert. Der Schleifenbereich wird in gleich große Bereiche aufgeteilt und jeder Thread arbeitet einen Bereich ab.

`#pragma omp atomic` Damit wird der folgende Ausdruck vor parallelen Zugriffen geschützt. Das ist nötig, wenn wir den Startwert und die Länge updaten wollen.

8.2 Kooperation: Schere, Stein, Papier (0,5 BP)

Implementieren Sie das Spiel Schere, Stein, Papier (Echse, Spock)[1], bei dem zwei Threads gleichzeitig gegeneinander spielen. Um zu gewinnen müssen drei Spielen gewonnen werden. Zur Erinnerung: Schere schneidet Papier, Papier bedeckt Stein und Stein schleift Schere. Die Wahl der Geste kann wieder durch den oben eingeführten Zufallsgenerator bestimmt werden.

Ein weiterer Thread benachrichtigt die beiden Spieler über eine neue Runde und wertet im Anschluss das Ergebnis aus. Bei Erreichen der drei gewonnenen Spiele informiert der zudem über den Abbruch und gibt den Gesamtsieger bekannt. Sollte nach 50 Runden noch kein Sieger feststehen, wird das Spiel unentschieden beendet.

Überlegen Sie sich hierzu, wie Sie die beiden Spieler-Threads gleichzeitig starten und wie Sie anschließend dem Auswerte-Thread mitteilen, dass beide ihre Wahl übergeben haben.

8.2.1 Ergebnis und Abnahme

Um den (halben) Bonuspunkt zu bekommen, müssen folgende Ergebnisse vorgezeigt werden:

1. Sie haben ein Programm geschrieben, dass drei Threads verwendet: zwei Spieler und einen Schiedsrichter (das kann auch der main-Thread sein).
2. Der Schiedsrichter startet eine neue Runde und wartet auf die Ausgabe der Spieler. Dann entscheidet er, wer diese Runde gewonnen hat. Nach drei gewonnenen Runden endet das Spiel mit der Ausgabe des Gewinners. Hat nach 50 Runden noch kein Spieler gewonnen, so endet das Spiel unentschieden.
3. Die Spieler-Threads erzeugen nach Rundenstart jeweils zufällig eine neue Zahl.
4. Die Abnahme erfolgt durch den Betreuer im Praktikum. Den Code laden Sie wie üblich in Moodle hoch.

8.2.2 Mögliche Erweiterungen (Optional)

Alternativ zur Zufallsausgabe können Sie im Spieler auch verschiedene Strategien implementieren. Versuchen Sie, Spieler mit verschiedenen Strategien gegeneinander antreten zu lassen. Welche Strategie schneidet am besten ab? Ein Spieler könnte auch durch einen Menschen ausgeführt werden: der Thread wartet dann auf eine Eingabe.

[1] Die Erweiterung geht auf Sam Kass zurück und wurde durch die Serie The Big Bang Theory bekannt.