Den Code für die beiden Aufgaben auf diesem Übungsblatt finden Sie im Archiv `bs1-ue09.zip`, das Sie nach dem Start eines Ubuntu-Containers (`hgesser/ubuntu-dev`) wie folgt herunterladen und entpacken. Danach wechseln Sie in den Ordner `ue09`:

```
cd /realworld
wget swf.hgesser.de/bs-b1/prakt/bs1-ue09.zip
unzip bs1-ue09.zip
cd ue09
```

1. Inter-Prozess-Mutex

Die POSIX-Mutexe (`pthread_mutex_t`) erlauben die Synchronisation von Threads (innerhalb eines Prozesses), aber sie lassen sich nicht zur Synchronisation von Prozessen nutzen.

Das Programm `lockfiles.c` initialisiert über `initnum()` eine Textdatei mit der Zeile

```
42
```

und liest diese dann zur Kontrolle mit `readnum()` wieder ein und gibt den Wert aus. Dann wird ein Kindprozess erzeugt, und Vater und Kind rufen in einer Schleife 1000 x die Funktion `decnum()` bzw. `incnum()` auf, welche jeweils die Datei öffnet, die Zeile mit der Zahl liest und die um 1 dekrementierte bzw. inkrementierte Zahl zurück schreibt. `incnum()` sieht so aus:

```
void incnum () {
  int num;
  FILE *handle = fopen (FILENAME, "r+");
  fscanf (handle, "%d", &num);
  rewind (handle);
  fprintf (handle, "%d\n", ++num);
  fclose (handle);
}
```

Informationen zu `fopen`, `fscanf`, `rewind`, `fprintf` und `fclose` finden Sie bei Bedarf in den *man pages*. Die Zeile

```
fprintf (handle, "%d\n", ++num);
```

bewirkt übrigens, dass *zuerst* der Wert von `num` um 1 erhöht wird (++) und *dann* der aktualisierte Wert mit `fprintf()` in die Datei geschrieben wird.

Vater und Sohn arbeiten parallel; das erhoffte Ergebnis nach 1000 x „++“ und 1000 x „--“ ist, dass in der Datei wieder der Wert 42 steht.

a) Übersetzen Sie das Programm mit `gcc -o lockfiles lockfiles.c` und starten Sie es durch Eingabe von `./lockfiles` . Sie sollten eine Ausgabe der folgenden Form sehen:

```
student@swfdebian:~$ ./lockfiles
Parallele Zaehler mit Dateizugriff
num = 42
1000 x decnum: fertig.
1000 x incnum: fertig.
num = -260 != 42 !!
```

wobei der von 42 abweichende Wert in der letzten Zeile bei jedem Aufruf ein anderer ist. Es besteht offensichtlich ein Synchronisationsproblem durch den gleichzeitigen Dateizugriff von Vater und Sohn.

b) Identifizieren Sie im Programm die kritischen Bereiche. Wenn es sich um zwei Threads handelte, könnten Sie einen Mutex deklarieren und die kritischen Bereiche mit `pthread_mutex_lock` schützen.

c) Unix bietet über die Funktion `lockf()` einen dateibasierten Lock-Mechanismus, der sich wie ein Mutex nutzen lässt und der prozess-übergreifend funktioniert. In den Zeilen 15, 19 und 28 der Programmdatei wird der Name einer Lock-Datei festgelegt, ein Dateideskriptor `lockfd` für die Lockdatei deklariert und schließlich bei der Initialisierung die Datei geöffnet:

```
15    #define LOCKFNAME "./countme.lck"
[...]
19    int lockfd;  // File Descriptor fuer Lock-Datei
[...]
28      lockfd = open (LOCKFNAME, O_RDWR | O_TRUNC | O_CREAT, 0644);
```

Dieser Deskriptor steht nach dem Aufruf von `fork()` auch im Kindprozess bereit – dadurch lässt sich die Lockdatei zur Synchronisation zwischen Vater und Kind nutzen.

Lesen Sie die *man page* zu `lockf` und finden Sie heraus, wie Sie `lockf()` aufrufen müssen, um exklusiven Zugriff auf die Lockdatei anzufordern bzw. wieder aufzugeben. Die Hinweise auf *specified sections of the file* erläutern, dass die Sperre sich auf Teile der Lockdatei beschränken kann; wählen Sie einen der `lockf()`-Parameter so, dass die ganze Datei gesperrt wird.

Setzen Sie an den Stellen, an denen Sie in einem POSIX-Thread-Programm `pthread_mutex_lock()` und `pthread_mutex_unlock()` verwendet hätten, die Aufrufe von `lockf()` ein und prüfen Sie, ob das Programm nun fehlerfrei arbeitet.

(*Hinweis* #1: Es gibt auch eine Funktion `flock()`, welche ähnliche Features wie `lockf()` bereitstellt. Im Test im Docker-Container funktionierte das Locking mit `flock()` aber nicht.)

(*Hinweis* #2: Beachten Sie, dass das Programm unterschiedliche Funktionen zum Öffnen der Datendatei und der Lock-Datei verwendet – erinnern Sie sich daran, welche der beiden Funktionen `fopen()` und `open()` ein Syscall-Wrapper und welche eine „höhere“ Bibliotheksfunktion ist?)

d) Der beschriebene Mechanismus stellt *genau einen* Mutex zur Verfügung. Benötigen Sie mehrere, könnten Sie mehrere Lockdateien verwenden – oder eine einzige Lockdatei so einsetzen, dass sie mehrere Mutexe unterstützt. Schauen Sie erneut in die *man page* zu `lockf` und überlegen Sie, wie Sie eine beliebige Zahl von Mutexen mit `lockf()` implementieren könnten.

2. Synchronisation mit Barrieren

Neben den Mutexen und Semaphoren, die Sie in der Vorlesung kennengelernt haben, gibt es noch weitere *Synchronisationsprimitive* – darunter die so genannte *Barriere*, die in ihrer Funktion am deutlichsten eine „synchronisierende“ Wirkung hat.

Eine Barriere kann das folgende gewünschte Verhalten einer Menge von Threads erzwingen:

- Mehrere Threads arbeiten gemeinsam an einer Aufgabenstellung.
- Jeder Thread arbeitet prinzipiell unahängig von den übrigen Threads, aber die Arbeiten sind in jedem Thread in mehrere Abschnitte unterteilt.
- Ein Thread darf erst dann von einem Abschnitt in den nächsten wechseln, wenn auch alle übrigen Threads den Abschnitt komplettiert haben.

Dazu wird eine Barriere zwischen den Abschnitten errichtet. Hat ein Thread einen Abschnitt komplettiert, meldet er das durch Aufruf einer Barrieren-Funktion. Er wird an dieser Stelle blockieren und seine Arbeit erst dann fortsetzen, wenn auch alle übrigen Threads den Abschnitt komplettiert und ebenfalls die Barrieren-Funktion aufgerufen haben. Der letzte Thread aus der Gruppe kann seine Arbeit also ohne Unterbrechung fortsetzen und in den nächsten Abschnitt wechseln, und nach ihm können auch alle übrigen Threads in den nächsten Abschnitt wechselnd

Betrachten Sie das Programm `synchronized-workers.c` (siehe Listing auf der folgenden Seite; es liegt ebenfalls im Ordner `ue09`).

a) Lesen Sie das Programm. Zur Erklärung einiger interessanter Code-Abschnitte:

Es gibt zwar mit `getpid()` eine Funktion, die die Prozess-ID zurückgibt, die entsprechende Funktion `gettid()` für die Thread-ID fehlt in einigen älteren Linux-Versionen aber in der Standardbibliothek. Linux stellt einen System Call für diese Abfrage bereit, der sich über `syscall(__NR_gettid)` aufrufen lässt. Das passiert in Zeile 13.[1]

1 Im Ubuntu-Container funktioniert auch der Aufruf `gettid()`, wenn Sie die Anweisung `#define _GNU_SOURCE` am Anfang der C-Datei ergänzen, siehe Manpage zu `gettid`.

```
1   // synchronized-workers.c
2
3   #include <pthread.h>
4   #include <sys/types.h>
5   #include <stdio.h>
6   #include <unistd.h>
7   #include <sys/syscall.h>
8   pthread_t         t1, t2, t3;
9   pthread_barrier_t barrier;   // Deklaration einer Barrieren-Variable
10
11  void *worker (void *args) {
12    int i, ret = 0;
13    int tid = syscall (__NR_gettid);  // Get thread ID
14    for (i=0; i<10; i++) {
15      printf ("[%d]: i=%d\n", tid, i);
16      if ((i==3) || (i==6)) {
17        printf ("Thread %d erreicht Barriere\n", tid);
18        // hier Platz für Barrieren-Code; setze ret
19        printf ("Thread %d hinter Barriere (ret=%d)\n", tid, ret);
20      }
21    }
22  }
23
24  int main () {
25    // hier Platz für Barrieren-Initialisierung
26    printf ("Start\n");
27    pthread_create (&t1, NULL, worker, NULL);
28    pthread_create (&t2, NULL, worker, NULL);
29    pthread_create (&t3, NULL, worker, NULL);
30    pthread_join (t1, NULL);
31    pthread_join (t2, NULL);
32    pthread_join (t3, NULL);
33    printf ("Fertig\n");
34  }
```


Das Hauptprogramm erzeugt in Zeile 27–29 drei Worker-Threads, welche alle dieselbe Funktion `worker()` ausführen.

Jeder Worker-Thread gibt im wesentlichen zehn Zeilen Text aus, in denen jeweils die Thread-ID und die Schleifenvariable `i` stehen. Die Arbeit soll in drei Abschnitte (0..3, 4..6, 7..9) unterteilt werden, und die Übergänge von einem Abschnitt in den nächsten sollen über eine Barriere synchronisiert werden. Der nötige Code für die Synchronisation fehlt noch.

b) Übersetzen Sie das Programm mit dem folgenden Befehl:

```
gcc -pthread -o synchronized-workers synchronized-workers.c
```

Führen Sie das Programm dann mehrfach aus. (Dem Programmnamen stellen Sie immer `./` voran, damit die Shell das Programm im aktuellen Ordner findet.)

Sie erkennen dann, dass die Barriere (wie zu erwarten) noch nicht in Betrieb ist, denn die Ausgaben „… erreicht Barriere“ und „… hinter Barriere“ erscheinen immer direkt hintereinander, z. B. wie im nebenstehenden Block: Die Threads arbeiten einfach durch. Das gilt es nun zu beheben.

```
Start
[64]: i=0
[64]: i=1
[64]: i=2
[64]: i=3
Thread 64 erreicht Barriere
Thread 64 hinter Barriere (ret=0)
[64]: i=4
[64]: i=5
...
```

c) Installieren Sie im Ubuntu-Container mit dem Kommando

```
sudo apt install manpages-posix-dev
```

eine umfangreichere Sammlung von Manpages nach. Darunter befinden sich nun auch die Manpages für `pthread_barrier_init()` und `pthread_barrier_wait()`. Lesen Sie diese mit

```
man pthread_barrier_init
man pthread_barrier_wait
```

durch. Falls die Nachinstallation nicht gelingt, finden Sie die Manpages auch online:

https://linux.die.net/man/3/pthread_barrier_init
https://linux.die.net/man/3/pthread_barrier_wait

d) Bauen Sie sinnvolle Aufrufe der beiden Funktionen in das Programm ein und testen Sie, ob es jetzt die Barrieren-Funktionalität umsetzt.

e) Passen Sie das Programm so an, dass es mit einer beliebigen Zahl von Worker-Threads arbeitet, die vor Zeile 8 über

```
#define NUMBER_OF_WORKERS 5
```

definiert (im Beispiel als 5) wird. Verwenden Sie dazu ein `pthread_t`-Array. Was ist bei der Initialisierung der Barriere zu beachten?