UNIVERSITÄT OSNABRÜCK

marcel.luetkedreimann [☺] uni-osnabrueck.de

Übung zur Vorlesung
Betriebssysteme
Wintersemester 2023/24

# Programmierblatt 2: POSIX-Threads

Programmierblatt 2: POSIX-Threads

# Primeserver

➪ **Abgabe der Lösungen bis Montag, 04. Dezember 16:00 im AsSESS**

Auf Programmierblatt 1 haben wir uns unter anderem mit UNIX-Prozessen beschäftigt. Für die folgenden Aufgaben sollen Sie POSIX-Threads (pthreads) verwenden. Denken Sie bei allen Aufgabenteilen an eine geeignete Fehlerbehandlung für pthreads!

Ein Server-Thread soll nach Primzahlen suchen und diese in einer Datenstruktur speichern. Eine beliebige Anzahl Client-Threads sollen die Primzahlen aus der Datenstruktur auslesen und ausgeben. Dabei müssen die kritischen Abschnitte des Programms geschützt werden. Außerdem sollen Server und Client synchronisiert werden, sodass ein Client nur eine Primzahl ausliest, wenn eine Zahl im Speicher verfügbar ist. Als Letztes soll sich um ein geordnetes Beenden des Programms gekümmert werden.

**Geben Sie Ihre Entwicklungsschritte jeweils in den angegebenen Dateien ab.**

## 1 Primeserver und Primeclient Grundgerüst (20 Punkte)

Wir beginnen zunächst mit einem vereinfachten Szenario:

- Es gibt neben dem Server-Thread vorerst nur einen Client-Thread.
- Die Primzahlen werden zunächst nur in einer einfachen Variable gespeichert.
- Auf das Schützen von kritischen Bereichen und das Synchronisieren von Threads wird verzichtet.
- Server und Client Thread beenden sich nie.

Erstellen Sie also einen Server-Thread mithilfe von **pthread_create(3)**, der in einer Endlosschleife ein Zähler hochzählt und prüft, ob der Zähler eine Primzahl ist. Für die Überprüfung finden Sie die Funktion „isPrime“ in der Vorgabe. Wurde eine Primzahl gefunden, soll diese in einer globalen Variable speichert werden. Anschließend soll der Server mithilfe von **sleep(3)** eine Sekunde warten. Der Ablauf soll sich in einer Endlosschleife wiederholen. Der Client-Thread ließt in einer Endlosschleife die Zahl aus der globalen Variable aus, setzt die globale Variable auf 0 und gibt die Primzahl aus. Dann soll auch der Client eine Sekunde warten. Das Programm soll, auch wenn die Threads nie terminieren, auf die Fertigstellung der Threads mittels **pthread_join(3)** warten.

**Hinweise:**

- Lassen Sie auch den Server-Thread eine Ausgabe machen, wenn eine Primzahl gespeichert wurde, um den Ablauf nachzuvollziehen.
- Es ist sehr wahrscheinlich, dass das Programm auch ohne Schutz der kritischen Bereiche korrekt funktioniert.
- Sollte ein Funktionsparameter nicht verwendet werden, kann die Compilerwarnung verhindert werden, indem der Parameter zum void-Datentyp gecastet wird:

```c
void* foo(void *args)
{
  (void)args;
  return NULL;
}
```

→ prime_a1.c

## 2 Threadsynchronisation (35 Punkte)

Nun soll der Zugriff auf die globale Variable durch ein Mutex geschützt werden. Das Mutex kann mittels **pthread_mutex_init(3)** initialisiert werden. Mithilfe von **pthread_mutex_lock(3)** kann das Lock gesetzt werden und mit **pthread_mutex_unlock(3)** lässt sich das Lock wieder freigeben. Erstellen und initialisieren Sie außerdem zwei Bedingungsvariablen mithilfe von **pthread_cond_init(3)**. Der Server soll mittels **pthread_cond_wait(3)** warten, falls die Primzahl noch nicht ausgelesen wurde. Der Client soll hingegen warten, falls noch keine neue Primzahl gespeichert wurde. Server und Client können sich gegenseitig mit **pthread_cond_signal(3)** Bescheid geben, dass es weitergehen kann. Denken Sie auch an das Freigeben des Mutex mit **pthread_mutex_destroy(3)** und der Bedingungsvariablen mit **pthread_cond_destroy(3)** am Ende des Programms (auch wenn das Programm im aktuellen Zustand nie terminiert)!

Ändern Sie außerdem die Wartezeit des Servers auf 2 Sekunden, damit ein Warten des Clients sichtbar wird.

→ prime_a2.c

## 3 Größerer Zwischenspeicher (20 Punkte)

Jetzt soll ein größerer Zwischenspeicher implementiert werden. Zuvor konnte nur eine Primzahl in der globalen Variable gespeichert werden. Nun soll ein globales Array 5 Primzahlen speichern können. Damit sich Server und Client das Array teilen können, benötigen Server und Client eine eigene Zählvariable. Der Server speichert die Zahlen der Reihe nach in dem Array und beginnt wieder von vorne, falls das Array voll geworden ist. Der Client ließt das Array beginnend bei der ersten Position und setzt das Array an der gelesenen Postion auf 0. Auch der Client beginnt wieder am Anfang des Arrays, sollte er am Ende des Arrays angekommen sein.

Jetzt bietet es sich an, die Wartezeit des Servers wieder auf 1 Sekunde und die Wartezeit des Clients auf 2 Sekunden zu setzen, um zu prüfen, ob der Server wartet, falls das Array voll geworden sein sollte.

**Hinweise:**

- Um das Programm zu testen, können Sie auch andere Arraygrößen wählen.

→ prime_a3.c

## 4 Mehr Clients und Threadterminierung (25 Punkte)

Im letzten Aufgabenteil sollen beliebig viele Clients erlaubt werden. Nutzen Sie das Argument des Programms, um eine Anzahl Threads festlegen zu können. Außerdem soll sich um ein geordnetes Beenden gekümmert werden, sodass die Threads nicht mehr in einer Endlosschleife gefangen sind.

Dafür soll ein weiterer Thread erstellt werden, der eine Nutzereingabe abfragt. Hat der Nutzer die Entertaste gedrückt (also ein '\n' eingegeben) soll der Server und alle Clients beendet werden. Dafür bietet es sich an, eine globale Variable zu verwendet, die aussagt, ob die Threads beendet werden sollen. Da Threads in einem **pthread_cond_wait(3)** warten könnten, müsse alle Signale ausgelöst werden. Alle Clients können mit **pthread_cond_broadcast(3)** aufgeweckt werden. Für den Server reicht ein einfaches **pthread_cond_signal(3)**. Nach Drücken der Entertaste sollen keine Ausgaben mehr von Clients erfolgen!

→ `prime_a4.c`

**Tipps zu den Programmieraufgaben:**

- Kommentieren Sie Ihren Quellcode ausführlich, so dass wir auch bei Programmierfehlern im Zweifelsfall noch Punkte vergeben können!
- Denken Sie daran, dass viele Systemaufrufe fehlschlagen können! Fangen Sie diese Fehlerfälle ab (die Aufrufe melden dies über bestimmte Rückgabewerte, siehe die jeweiligen man-Pages), geben Sie geeignete Fehlermeldungen aus (z.B. unter Zuhilfenahme von **perror(3)**) und beenden Ihr Programm danach ordnungsgemäß.
- Die Programme sollen dem C11- und POSIX-Standard entsprechen sich mit dem gcc auf aktuellen Linux-Rechnern wie denen im CIP-Pool oder der BSVM übersetzen lassen.
  `gcc -std=c11 -Wall -D_GNU_SOURCE -pthread -o prime prime_ax.c`
  Weitere (nicht zwingend zu verwendende) nützliche Compilerflags sind: `-Wextra -Wpedantic -Werror -D_POSIX_SOURCE`
- Achten Sie darauf, dass sich der Programmcode ohne Warnungen übersetzen lässt, z.B. durch Nutzung von `-Werror`.