THREADS

Hinweise: Sämtliche Ausführungen gelten für unixoide Betriebssysteme. Lesen Sie bitte auch die zugehörige Praktikumsdokumentation durch.

1. GRUNDLAGEN

Bei Threads handelt es sich um eine besondere Form von leichtgewichtigen Prozessen. Anders als Prozesse haben Threads keinen eigenen Adressraum im Speicher. Sie teilen sich den Adressraum im Speicher mit dem übergeordneten Prozess. Threads können so sehr effizient miteinander kommunizieren. Für die moderne Programmierung werden Threads immer wichtiger: Damit die zahlreichen parallelen Prozessoren ausgelastet werden können, sind Threads unerlässlich.

In Linux-Systemen existieren zwei unterschiedliche Thread-IDs: Die POSIX-kompatible Thread-ID und die Thread-ID, die vom Linux-Kernel verwaltet wird. Die POSIX-kompatible Thread-ID wird immer in der pthread-Bilbiothek genutzt. `pthread_create` liefert diese Thread-ID zurück und `pthread_join` wartet auf diese Thread-ID. Mit dem Befehl `pthread_self()` kann ein Thread seine eigene Thread-ID ermitteln:

```
Pthread_t threadId = pthread_self();
printf („Meine Thread-ID ist %ld\n“, threadId);
```

Im Terminal können Sie die Threads mit dem Befehl `ps -eLf` überwachen. Die in der Spalte LWP angezeigten IDs der Threads unterscheiden sich von den POSIX-Thread-IDs. Seit dem Kernel 2.4 können unter Linux diese IDs mit dem Befehl `gettid()` abgerufen werden:T

```
pid_t threadId = gettid();
printf("Meine Thread-ID ist %i\n",threadId);
```

Hinweis:

Die C-Bibliothek `glibc` unterstützt erst seit dem Jahr 2019 den Befehl `gettid()`. Auch in aktuellen Varianten von `glibc` reicht es nicht aus, die Header-Datei `unistd.h` hinzuzufügen. Weiterhin muss `#define _GNU_SOURCE` im Kopfbereich angegeben werden.

2. MATRIXMULTIPLIKATION

Aufgabe 2.1: Schreiben Sie ein Programm zur Multiplikation von quadratischen Matrizen. Die Anzahl der Zeilen bzw. Spalten sollten Sie mit einer Konstanten festlegen. Die beiden zu multiplizierenden Matrizen sowie auch die Ergebnismatrix sollten Sie global deklarieren. Initialisieren Sie die beiden Ausgangsmatrizen mit Zufallszahlen. (Hinweis: Die Ergebnisse sind leichter zu verifizieren, wenn Sie zunächst die Zufallszahlen auf den Bereich 0-5 beschränken.)

In Abgrenzung zu den regulären Funktionen zur Matrixmultiplikation soll Ihre Matrixmultiplikation mit Threads arbeiten. Verwenden Sie zur Berechnung jeder Zeile der Ergebnismatrix einen eigenen Thread. Geben Sie die ID des Threads kurz nach der Erzeugung aus, sowie auch kurz nachdem der Thread mit seiner Arbeit fertig ist.

Die Threads sollen direkt mit den global deklarierten Matrizen arbeiten. Eine Übergabe der Matrizen als Parameter ist also nicht notwendig. Sie werden allerdings einen Parameter für das Zeilenende der Ergebnismatrix brauchen.

Aufgabe 2.2: Erhöhen Sie die Anzahl der Spalten und Zeilen der Matrizen (zum Beispiel auf 50). Sind die Ausgaben der Thread-IDs grundsätzlich strukturell gleich oder unterscheiden sich die Ausgaben? Warum?

Aufgabe 2.3: Die Anzahl der Rechenkerne wird in Zukunft beständig weiter steigen. Angenommen Sie hätten mehr Rechenkerne zur Verfügung als Zeilen in der Matrix zu berechnen sind. Wie könnten Sie das Programm weiter „parallelisieren“?

Aufgabe 2.4: Könnten Sie das geschriebene Programm auch in gleicher Weise mit `fork()`, also parallelen Prozessen umsetzen? Begründen Sie Ihre Aussage.

3. PRIMZAHLEN ZÄHLEN

Ohne Primzahlen sind zentrale Funktionen von asymmetrischen Verschlüsselungsfunktionen nicht denkbar. Allerdings werden sehr große Primzahlen gebraucht. Es hängt also sehr viel davon ab, sehr große Primzahlen in akzeptabler Zeit zu finden. Die einfache, aber auch aufwändige Division einer zu überprüfenden Zahl durch alle vorangegangenen Primzahlen ist hierfür zu langsam. Stattdessen nutzt man Testverfahren wie zum Beispiel den Fermatschen Primzahltest (mehr zu Primzahlentests finden Sie zum Beispiel hier: https://de.wikipedia.org/wiki/Primzahltest). Diese Tests sind nicht vollständig zuverlässig, aber sie können mit einer so hohen Wahrscheinlichkeit eine valide Aussage darüber treffen, ob die vorliegende Zahl eine Primzahl ist, dass für die Kryptographie fast ausschließlich solche Funktionen zum Einsatz kommen.

Aufgabe 3.1: Schreiben Sie ein Programm, welches einen definierten Zahlenbereich (z.B. 10 000 000 – 20 000 000) auf Primzahlen untersucht und ausgibt, wie viele Primzahlen gefunden wurden. Beginnen Sie zunächst mit einem Thread, der den gesamten Zahlenbereich untersucht. Sie müssen hierfür nicht selbst den Fermatschen Primzahlentest implementieren. Stattdessen können Sie die Funktion `is_prime` aus der gleichnamigen Headerdatei nutzen (vgl. Material zum Übungsblatt). Der Thread-Funktion sollten Sie sowohl die kleinste als auch die größte Zahl des Zahlenbereichs übergeben. Die Funktion überprüft dann den gesamten Zahlenbereich auf Primzahlen und gibt die Anzahl der gefundenen Primzahlen zurück. Verwenden Sie hierfür keine globalen Variablen. Nutzen Sie für die Übergabe der Parameter an die Funktion die `pthread_create` Methode. Für den Rückgabewert nutzen Sie die `pthread_join` Methode. Messen Sie auch die Zeit, die Ihr

Programm für die Ausführung benötigt. Hierfür können Sie `time_measurement.h/c` aus dem Material zum Übungsblatt verwenden.

Aufgabe 3.2: Erhöhen Sie schrittweise die Anzahl der parallelen Threads. Teilen Sie hierfür den gleichen Zahlenbereich wie in Aufgabe 2.1 in gleich große Teile auf, die Sie auf die unterschiedlichen Threads verteilen. Was passiert mit der Laufzeit des Programms, wenn Sie die Anzahl der Threads steigern? Ab welchem Zeitpunkt zeigt es keinen oder wenig Effekt, wenn die Anzahl der Threads weiter gesteigert wird?

Aufgabe 3.3: Beobachten Sie Ihr Programm mit mehreren parallelen Threads mit dem Programm `ps` (z.B. `ps aux -T`). Welche PIDs haben die Threads? Wie viele Threads werden für den übergeordneten Prozess ausgewiesen?