Betriebssysteme

# Übung 5 Multithreading


**Gruppenmitglieder:** Johannes Theiner
**Semester:** Sommersemester 2019
**letzte Änderung:** 17.04.2019

# 1 Level A

## 1.1 Verständnisfragen

### 1.1.1 Welche Gründe könnten dafür sprechen, bei einer Programmimplementierung Threads zu verwenden, statt mehrere Prozesse zu verwenden?

- Daten zwischen Threads zu synchronisieren ist einfacher
- Schnellerer Kontextwechsel
- Schnellere Erzeugung
- mehrere Kerne können besser ausgenutzt werden

### 1.1.2 Was unterscheidet einen Thread von einem Prozess?

| **Thread** | **Prozess** |
|---|---|
| Nutzt Speicher von Eltern | Eigener Speicher |
| Billiger Kontextwechsel | teurer Kontextwechsel |
| abhängig von anderen | nicht abhängig |

# 2 Level B

## 2.1 Aufgabe

### 2.1.1 Welches Problem beobachten Sie?

Der Wert von sharedInteger entspricht nicht der Anzahl der ausgeführten Threads.

### 2.1.2 Was ist die Ursache?

Es können sich andere Threads zwischen die Inkrementierung und das zurückschreiben schieben und deren Änderungen werden dann später vom älteren Thread überschrieben.

### 2.1.3 Sorgen Sie in obigem Programm für wechselseitigen Ausschluss mit Mutexen aus der pthreads-Bibilothek oder FreeBSD-Semaphoren

```c
// build: clang -lpthread thread3.c
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int doPrintf = 1;

pthread_mutex_t mutex;
int sharedInteger = 0;
static int numIncrements = 2000;
static int sleepTime_micros = -1;

void maySleep() {
    if (sleepTime_micros >= 0) {
        usleep(sleepTime_micros); // 0 => just yield CPU
    }
}

void *thread_printdots() {
    int i, j;
    for (i = 0; i < numIncrements; i++) {
        pthread_mutex_lock(&mutex);
        j = sharedInteger;
        j = j + 1;
        if (doPrintf) {
            printf(".");
            fflush(stdout);
        }
        maySleep();
        sharedInteger = j;
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}

int main(int argc, char **argv, char **envp) {
    pthread_t thread_1;
    pthread_mutex_init(&mutex, NULL);
    printf("usage: a.out numIncrements(=2000) sleepTime_micros(=-1)\n");
```

```
    char * end;
    if (argc >= 2) {
        numIncrements = strtol(argv[1], &end, 10); // number of increments of sharedIntege
    }
    if (argc == 3) {
        sleepTime_micros = strtol(argv[2], &end, 10); // time to sleep between loops
    }
    sharedInteger = 0;
    if (pthread_create(&thread_1, NULL, thread_printdots, NULL)) {
        perror("error creating thread.");
        abort();
    }
    for (int i = 0; i < numIncrements; i++) {
        pthread_mutex_lock(&mutex);
        sharedInteger++;
        pthread_mutex_unlock(&mutex);
        if (doPrintf) {
            printf("o");
            fflush(stdout);
        }
        maySleep();
    }
    if (pthread_join(thread_1, NULL)) {
        perror("error joining thread.");
        abort();
    }
    printf("\nsharedInteger = %d\n", sharedInteger);
    pthread_mutex_destroy(&mutex);
    return (sharedInteger == 2 * numIncrements) ? 0 : 1;
}
```

## 2.2 Verständnisfragen

### 2.2.1 Führen Sie Gründe auf warum ein Wechsel zwischen zwei Threads günstiger sein könnte als zwischen zwei Prozessen.

- Es müssen nur Register gesichert/wiederhergestellt werden
- Braucht keine zusätzlichen Ressourcen

### 2.2.2 Führen Sie Vor- und Nachteile von Kernel-Level Threads im Vergleich zu User-Level-Threads auf.

Vorteile:

- Scheduler kann Prozesse mit vielen Threads bevorzugen
- Wenn viel blockiert wird können andere Threads die Wartezeit verwenden

Nachteile:

- großer Overhead
- viel langsamer als User-Level-Threads

### 2.2.3 An welchen Stellen eines C-Programms kann vom Betriebssystem ein Kontextwechsel zu einem anderen Thread oder Prozess durchgeführt werden?

Bei Threads kann dieser Wechsel zwischen einzelnen Anweisungen ausgeführt werden, solange der Thread nicht gesperrt ist. Bei Prozessen kann der Kontextwechsel zwischen einzelnen Anweisungen ausgeführt werden.

### 2.2.4 Wie stellt die Semaphorenimplementierung sicher, das keine Probleme auftauchen?

Wird eine Resource angefordert dekrementiert die Semaphore einen internen Zähler und vergibt bei einem Zähler von 0 keine Resourcen mehr, da keine mehr vorhanden sind. Wird eine Resource wieder frei gegeben und die Semaphore darüber benachrichtigt wird der Zähler wieder inkrementiert und die Resource kann wieder vergeben werden.