Fachhochschule Erfurt
Vorlesung: Betriebssysteme
Wintersemester 2024/25

Dr. Jürgen Hess

Aufgabenblatt 1

Thema: Betriebssystemmechanismen zur Synchronisation von Prozessen

Abgabe: 21.01.2025

Für die Lösung der Aufgaben empfehle ich einen Online Editor für C zu verwenden, da hier bereits alle Voraussetzungen und Bibliotheken vorhanden sind und dieser Platformunabhängig funktioniert: https://www.onlinegdb.com/ Sie können den Editor frei und ohne Anmeldung verwenden.

Die hier abgedruckten Programmiercode können Sie in den Editor kopieren oder diesen auch hier herunterladen: https://github.com/juehess/lecturematerial-os/uebung1

POSIX Thread- und Synchronisationsfunktionen

Für die Lösung der folgenden Aufgaben finden Sie unten eine kurze Dokumentation der wichtigsten POSIX-Thread-Funktionen. Weitere Details zum POSIX-Standard finden Sie bei Interesse hier: https://pubs.opengroup.org/onlinepubs/9699919799/

- **pthread_create**: Erstellt einen neuen Thread.

```
int pthread_create(
    pthread_t *thread,                  // Zeiger auf das
                                        // Thread-Objekt
    const pthread_attr_t *attr,         // Thread-Attribute
                                        // (NULL für Standard)
    void *(*start_routine)(void *),     // Funktion, die der
                                        // Thread ausführt
    void *arg                           // Argument für die
                                        // Funktion
);
```

- **pthread_join**: Wartet auf die Beendigung eines Threads.

```
int pthread_join(
    pthread_t thread,                   // Thread, auf den
                                        // gewartet wird
    void **retval                       // Rückgabewert des
                                        // Threads (NULL, wenn
                                        // nicht benötigt)
);
```

- **pthread_mutex_init**: Initialisiert ein Mutex-Objekt.

```
int pthread_mutex_init(
    pthread_mutex_t *mutex,             // Zeiger auf das
                                        // Mutex-Objekt
    const pthread_mutexattr_t *attr     // Mutex-Attribute
                                        // (NULL für Standard)
);
```

- **pthread_mutex_lock**: Sperrt das Mutex.

```
int pthread_mutex_lock(
    pthread_mutex_t *mutex              // Zeiger auf das
                                        // Mutex-Objekt
);
```

- **pthread_mutex_unlock**: Gibt das Mutex frei.

```
int pthread_mutex_unlock(
    pthread_mutex_t *mutex              // Zeiger auf das
                                        // Mutex-Objekt
);
```

- **sem_init**: Initialisiert ein Semaphore-Objekt.

```
int sem_init(
    sem_t *sem,                         // Zeiger auf das
                                        // Semaphore-Objekt
    int pshared,                        // 0, wenn Semaphore nur in
                                        // einem Prozess genutzt wird
                                        // 1, wenn Semaphore zwischen
                                        // Prozessen geteilt wird
    unsigned int value                  // Startwert des Semaphores
);
```

- **sem_wait**: Decrementiert den Zähler des Semaphores und blockiert ggf.

```
int sem_wait(
    sem_t *sem                          // Zeiger auf das
                                        // Semaphore-Objekt
);
```

- **sem_post**: Erhöht den Zähler des Semaphores.

```
int sem_post(
    sem_t *sem                          // Zeiger auf das
                                        // Semaphore-Objekt
);
```

Aufgabe 1: Counter synchronisieren

Vollziehen Sie das Beispiel aus der Vorlesung noch einmal nach und beantworten Sie dann folgende Fragen:

(a) Reparieren Sie den unsynchronisierten Zugriff auf den Counter reparieren, indem sie *pthread_mutex* hinzufügen.

(b) Warum ist es notwendig, den Mutex vor der Verwendung mit *pthread_mutex_init* zu initialisieren und am Ende mit *pthread_mutex_destroy* zu zerstören? Was könnte passieren, wenn dies vergessen wird?

(c) Was passiert, wenn ein Thread *pthread_mutex_lock* aufruft, aber *pthread_mutex_unlock* vergisst? Wie könnten solche Fehler erkannt und vermieden werden?

(d) Welche Rolle spielt die künstliche Verzögerung (usleep)? Warum wird dadurch das Risiko von Race Conditions in diesem Beispiel erhöht?

```c
#include <stdio.h>
#include <unistd.h> // Für usleep

int counter = 0; // Gemeinsame Ressource

void* increment(void* arg) {
    int thread_id = *(int*)arg;
    for (int i = 0; i < 100; i++) {

        printf("Thread %d arbeitet, counter: %d\n", thread_id, counter);
        int temp = counter; // Lese den aktuellen Wert
        usleep(0.2);
        temp = temp + 1;     // Erhöhe den Wert
        counter = temp;      // Schreibe den neuen Wert zurück
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;

    int id1 = 1, id2 = 2;
    pthread_create(&t1, NULL, increment, &id1);
    pthread_create(&t2, NULL, increment, &id2);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Finaler Counter-Wert: %d\n", counter);
    return 0;
}
```

Aufgabe 2: Gleichzeitiger Zugriff

Implementieren Sie einen Semaphore-Mechanismus, um maximal zwei gleichzeitige Zugriffe auf eine Ressource zuzulassen.

(a) Fügen Sie einen Semaphore sem_t hinzu, um den gleichzeitigen Zugriff zu regeln.

(b) Wie initialisieren Sie einen Semaphore, um maximal zwei gleichzeitige Zugriffe zu erlauben? Welche Parameter sind bei der Initialisierung entscheidend

(c) Beschreiben Sie, wie sich der Zustand der Semaphore verändert, wenn Threads auf die Ressource zugreifen oder diese verlassen. Welche Werte nimmt die Semaphore an und was bedeuten sie?

(d) Kann es in diesem Beispiel noch zu Race Conditions kommen? Falls ja, unter welchen Bedingungen und wie könnte das verhindert werden?

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


void* car(void* arg) {
    int car_id = *(int*)arg;

    printf("Auto %d wartet auf einen Parkplatz...\n", car_id);
    printf("Auto %d parkt.\n", car_id);
    sleep(1); // Simuliert das Parken
    printf("Auto %d verlässt den Parkplatz.\n", car_id);

    return NULL;
}

int main() {
    pthread_t cars[5];
    int car_ids[5] = {1, 2, 3, 4, 5};

    for (int i = 0; i < 5; i++) {
        pthread_create(&cars[i], NULL, car, &car_ids[i]);
    }

    for (int i = 0; i < 5; i++) {
        pthread_join(cars[i], NULL);
    }

    return 0;
}
```

Aufgabe 3: Erzeuger-Verbraucher-Problem

Das Erzeuger-Verbraucher-Problem ist ein klassisches Synchronisationsproblem in der Informatik, das das Zusammenspiel zwischen mehreren Threads (Erzeugern und Verbrauchern) veranschaulicht, die auf eine gemeinsame Ressource (z. B. einen Puffer) zugreifen. Ziel ist es, sicherzustellen, dass Erzeuger Daten in den Puffer schreiben und Verbraucher Daten aus dem Puffer lesen, ohne dass es dabei zu Konflikten oder Dateninkonsistenzen kommt. Vollziehen Sie das Beispiel aus der Vorlesung noch einmal anhand des Source Codes nach. Beantworten Sie dann folgende Fragen:

(a) Was passiert, wenn der Mutex entfernt wird?

(b) Warum verwenden wir zwei Semaphoren (leer und voll)? Wäre auch die Verwendung einer einzelnen Semaphore möglich?

(c) Füge zusätzliche Threads wiefolgt hinzu:

```
pthread_t producers[2], consumers[2];
pthread_create(&producers[0], NULL, producer, NULL);
pthread_create(&producers[1], NULL, producer, NULL);
pthread_create(&consumers[0], NULL, consumer, NULL);
pthread_create(&consumers[1], NULL, consumer, NULL);
```

Funktioniert der Mechanismus weiterhin?

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define PUFFERGROESSE 3
int puffer[PUFFERGROESSE];
int zaehler = 0;

sem_t frei, belegt; // Semaphoren für freie und belegte Plätze
pthread_mutex_t lock; // Schutz des Puffers

void* erzeuger(void* arg) {
    for (int i = 0; i < 5; i++) {
        sem_wait(&frei); // Warte auf freien Platz
        pthread_mutex_lock(&lock);

        puffer[zaehler++] = i; // Element hinzufügen und Zähler erhöhen
        printf("Erzeuger fügt %d hinzu. Puffergröße: %d\n", i, zaehler);

        pthread_mutex_unlock(&lock);
        sem_post(&belegt); // Signalisiere belegten Platz
        sleep(1);
    }
    return NULL;
}

void* verbraucher(void* arg) {
    for (int i = 0; i < 5; i++) {
        sem_wait(&belegt); // Warte auf ein belegtes Element
        pthread_mutex_lock(&lock);

        int element = puffer[--zaehler]; // Zähler verringern und Element
    entfernen
        printf("Verbraucher entnimmt %d. Puffergröße: %d\n", element,
    zaehler);
```

```
        pthread_mutex_unlock(&lock);
        sem_post(&frei); // Signalisiere freien Platz
        sleep(1);
    }
    return NULL;
}

int main() {
    pthread_t prod, cons;

    sem_init(&frei, 0, PUFFERGROESSE); // Puffer startet mit allen freien
    Plätzen
    sem_init(&belegt, 0, 0); // Keine belegten Plätze
    pthread_mutex_init(&lock, NULL);

    pthread_create(&prod, NULL, erzeuger, NULL);
    pthread_create(&cons, NULL, verbraucher, NULL);

    pthread_join(prod, NULL);
    pthread_join(cons, NULL);

    sem_destroy(&frei);
    sem_destroy(&belegt);
    pthread_mutex_destroy(&lock);

    return 0;
}
```

Aufgabe 4: Deadlocks

(a) Wir haben in der Vorlesung eine Lösung des vorliegenden Deadlock-Problems mit der Funktion *pthread_mutex_trylock* besprochen. Welche andere Lösung wäre hier ebenfalls möglich? Implementieren sie diese und überprüfen Sie das Resultat.

(b) Was ist der Unterschied zwischen einer Race Condition und einem Deadlock? Kann es in diesem Beispiel zusätzlich zu Deadlocks auch zu Race Conditions kommen?

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

pthread_mutex_t lock1, lock2;

void* thread1_func(void* arg) {
    pthread_mutex_lock(&lock1);
    printf("Thread 1: locked lock1\n");

    sleep(1); // Simuliert Verzögerung

    printf("Thread 1: waiting for lock2\n");
    pthread_mutex_lock(&lock2);
    printf("Thread 1: locked lock2\n");

    pthread_mutex_unlock(&lock2);
    pthread_mutex_unlock(&lock1);
    printf("Thread 1: released locks\n");

    return NULL;
}

void* thread2_func(void* arg) {
    pthread_mutex_lock(&lock2);
    printf("Thread 2: locked lock2\n");

    sleep(1); // Simuliert Verzögerung

    printf("Thread 2: waiting for lock1\n");
    pthread_mutex_lock(&lock1);
    printf("Thread 2: locked lock1\n");

    pthread_mutex_unlock(&lock1);
    pthread_mutex_unlock(&lock2);
    printf("Thread 2: released locks\n");

    return NULL;
}

int main() {
    pthread_t t1, t2;

    pthread_mutex_init(&lock1, NULL);
    pthread_mutex_init(&lock2, NULL);

    pthread_create(&t1, NULL, thread1_func, NULL);
    pthread_create(&t2, NULL, thread2_func, NULL);

    pthread_join(t1, NULL);
```

```
    pthread_join(t2, NULL);

    pthread_mutex_destroy(&lock1);
    pthread_mutex_destroy(&lock2);

    return 0;
}
```