Gruppe 1 Lösungen Prozesse & Threads

Augabe 1

> fork

src: https://www.man7.org/linux/man-pages/man2/fork.2.html

Das Kommando `fork` erstellt einen `Child` Process, der zum Zeitpunkt des Erstellens mit seinem `Parrent` bis auf die `Process ID` und eine Reihe anderer Faktoren identisch ist.

> execl

src: https://man7.org/linux/man-pages/man3/exec.3.html

Die `exec()` Funktions-Familie erstellt ein neues `Process Image` im laufenden Prozess, wobei `execl` von `execve` erbt und `const char *arg` als Argumente für die aufzurufende Funktion annimt, die `pointer` zu `null-terminated strings` sein müssen.

> waitpid

src: https://man7.org/linux/man-pages/man2/wait.2.html

Das `waitpid` Kommando pausiert die Ausführung des aktuellen Threads bis der/die durch die `pid` spezifizierte `child` Thread/s seinen Zustand ändern.

> clone

src: https://man7.org/linux/man-pages/man2/clone.2.html

`clone` erstellt ähnlich zu `fork` einen `Child` Thread, hat dabei aber präzisere Kontrolle über den Vorgang, so dass zum Beispiel Threads erstellt werden können, die sich den `virtual adress` Raum teilen.

> system

src: https://man7.org/linux/man-pages/man3/system.3.html

`system` nutzt `fork` um einen `child` Prozess zu erstellen, der ein `shell` Kommando mithilfe von `execl` ausführt und dannach zum aktuellen Prozess zurückkehrt.

Augabe 2

Ein Python Programm das seine Terminal-Argumente alle 10 Sekunden ausgiebt.

```
# https://docs.python.org/3/tutorial/stdlib.html#command-line-arguments

import sys
import time

arguments = sys.argv
arguments.pop(0)
```

```
while True:
    time.sleep(10)
    print(arguments)
```

src: https://linuxg.net/how-to-kill-processes-in-linux-and-unix/

1. Dieses Programm wird zweimal aufgerufen mit `python3 Ex_2a.py Argument1 Argument2 &`.
2. Mit `jobs` kann man sich die Liste der laufenden Prozesse ausgeben lassen.
3. Mit `kill -STOP %x` den Prozess stopen, wobei `x` gleich der `jobID`.#
4. Mit `ps x` wobei `x` die `jobID` ist, lässt sich der Status abfagen, `S` für laufende Prozesse, `Ss` für gestoppte.
5. Mit `kill -CONT %x` den Prozess wieder starten, wobei `x` gleich der `jobID`.
6. Mit `kill -TERM %x` lässt sich ein Prozess beenden. Das kann mit `jobs` kontrolliert werden.

- Mit `ps` die `PID` der laufenden bash finden.
- Mit `kill -9 %x` wobei x die `PID` der bash ist, die bash beenden.

src: https://docs.python.org/3/library/multiprocessing.html

```
from multiprocessing import Process
import os


def child_function():
    pid = os.getpid()
    for i in range(200):
        print('child process id: ', pid)


if __name__ == '__main__':
    childs = []
    for i in range(3):
        p = Process(target=child_function)
        childs.append(p)
        p.start()
    for i in range(200):
        string_to_print = ''
        for i in childs:
            string_to_print = string_to_print + str(i.pid) + ',  '
        print(string_to_print)
```

Aufgabe 3

> blocked -> running

Wenn ein Thread vom OS geblocked wird, wählt das OS einen anderen Thread, der ausgeführt wird. Das bedeutet, ein `blocked` Thread wird von Definiton aus nach Auflösung des `blocked` Grundes in den `ready` Zustand gesetzt um dem neuen laufenden Thread nicht zu schaden.

> ready -> blocked

Das OS blockiert einen Thread, wenn während der Laufzeit etwas passiert, auf das der Thread warten muss. Da ein Thread im `ready` Zustand auch inaktiv ist und blockert werden kann wenn das OS den Thread auswählt, braucht es eine Zustandsänderung `ready -> blocked` nicht, das ein System ordungsgemäß funktioniert.

Aufgabe 4

src: https://www-tutorialspoint-com.translate.goog/python3/python_multithreading.htm?_x_tr_sl=en&_x_tr_tl=fr&_x_tr_hl=fr&_x_tr_pto=sc

Die Python-Entsprechung von pthread_create() besteht darin, die _thread-Bibliothek zu importieren und dann einen Thread wie folgt zu erstellen: _thread.start_new_thread(Argumente) Oder man importiert die Bibliothek threading und erstellen eine Klasse Thread, in der Sie die Methode **init**(self[args]) austauschen müssen, um zusätzliche Argumente hinzuzufügen, sowie die Methode run(self[args]), um zu implementieren, was der Thread beim Start tun soll. Um den Thread zu erstellen, schreiben Sie: Thread=class_name(arguments) und um ihn zu starten: thread.start()

src: https://superfastpython.com/join-a-thread-in-python/#How_to_Join_a_Thread

Die Python-Entsprechung von pthread_join() ist die Methode join(). Sie ermöglicht einen Thread zu blockieren, bis ein anderer Thread beendet ist. Der Ziel-Thread kann von mehrere Gründen beendet werden, z.B. : Er beendet die Ausführung seiner Zielfunktion. Er beendet die Ausführung seiner Methode run(), wenn er die Klasse Thread erweitert. Hat einen Fehler oder eine Ausnahme erzeugt. Nachdem der Ziel-Thread seine Arbeit beendet hat, kehrt die join()-Methode zurück und der aktuelle Thread kann weiter ausgeführt werden. Die join()-Methode setzt voraus, dass man eine threading.Thread-Instanz für den Thread hat, den wir verbinden möchten.

Aufgabe 5

```
// Kompilation : gcc -o Aufgabe5 -pthread thread.c
//Ausführung : ./Aufgabe5 100
//sourcen : https://franckh.developpez.com/tutoriels/posix/pthreads/#LIV-A

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/types.h>
#include <unistd.h>

int sum=0;                          //Die gloabale Variable
pthread_mutex_t mutex;

void * helloworld(void * arg) {         //Die Funktion, dass jeder thread
1.000.000 mal eine Schleife durchlaufen
    int argument = * (int*)arg;
    for (int i = 0; i < 1000000; i++)
    {
        if (argument%2==0){
```

```
            pthread_mutex_lock(&mutex);
            for (int j = 0; j < 5; j++)             //die Hälfte addieren
in einer Schleife 5 Mal jeweils eine 1
            {
                sum+=1;
            }
            pthread_mutex_unlock(&mutex);           //die Hälfte
subtrahieren in einer Schleife 5 Mal jeweils eine 1
        }

        else{
            pthread_mutex_lock(&mutex);
            for (int j = 0; j < 5; j++)
            {
                sum-=1;
            }
            pthread_mutex_unlock(&mutex);
        }
    }
    return NULL;
}

int main(int argc, char** argv){
    int i, zh;
    int * args;
    pthread_t * threads;
    zh = atoi(argv[1]);
    threads = malloc(zh * sizeof(pthread_t));
    args = malloc(zh * sizeof(int));
    pthread_mutex_init(&mutex, NULL);           //initalisierung der mutex
    for (i = 0; i < zh; i++) {
        args[i] = i;
        pthread_create(&threads[i], NULL, helloworld, &args[i]);
//Erzeugung den Threads
    }

    for (i = 0; i < zh; i++) {
        pthread_join(threads[i], NULL);                 //Sammeleung alle
Threads
        printf("thread Nr. %d joined \n", i);
    }
    printf("Summe=%d\n",sum);
    return 0;
}
```

Wir beobachten, dass die globale Variable 0 ist, was logisch ist, da die Threads denselben Wert so oft addieren wie subtrahieren. Wir beobachten auch, dass die Threads in der Reihenfolge (0, 1, 2, ..., 98, 99), was auf den Mutex zurückzuführen ist, der dafür sorgt, dass nicht alle gleichzeitig auf die globale Variable zugreifen können, sondern einer nach dem anderen.

Abgaben von Anton Stimmer, Oscar Röth