1. Worker-Threads vs. Worker-Prozesse

Betrachten Sie die Programme `worker-threads.c` (Listing 1, links) und `worker-prozesse.c` (Listing 2, rechts), die Sie auch von der Kurswebseite herunterladen können (`bs1-ue08.zip`).

```
1   #include <pthread.h>  // pthread_*
2   #include <stdio.h>    // printf
3   #include <unistd.h>   // sleep
4
5
6
7   #define MAX_WORKERS 10
8
9   int i;
10
11  void *worker (void *args) {
12    int i;
13    sleep (1);
14    char c = (int)args;
15    for (i=0; i<10; i++) {
16      printf ("%c", c); fflush (0);
17      sched_yield ();
18    }
19    return NULL;
20  }
21
22  int main () {
23    printf ("--Start--\n");
24    pthread_t threads[MAX_WORKERS];
25    int i;
26
27    // Threads erzeugen
28    for (i=0; i<MAX_WORKERS; i++) {
29      pthread_create (&threads[i],
30                      NULL,
31                      worker,
32                      (void*)(i+'A'));
33    }
34
35
36
37    // auf Threads warten
38    for (i=0; i<MAX_WORKERS; i++)
39      pthread_join (threads[i], NULL);
40
41    printf ("\n--Fertig--\n");
42    return 0;
43  }
```

```
1   #include <stdio.h>      // printf
2   #include <unistd.h>     // sleep
3   #include <sys/types.h>  // waitpid
4   #include <sys/wait.h>   // waitpid
5   #include <sched.h>      // sched_yield
6
7   #define MAX_WORKERS 10
8
9   int i;
10
11  void *worker (void *args) {
12    int i;
13    sleep (1);
14    char c = (int)args;
15    for (i=0; i<10; i++) {
16      printf ("%c", c); fflush (0);
17      sched_yield ();
18    }
19    return NULL;
20  }
21
22  int main () {
23    printf ("--Start--\n");
24    int pids[MAX_WORKERS];
25    int i;
26
27    // Kindprozesse erzeugen
28    for (i=0; i<MAX_WORKERS; i++) {
29      pids[i] = fork ();
30      if (pids[i] == 0) {
31        // Sohn...
32        worker ((void*)(i+'A'));
33        return 0;
34      }
35    }
36
37    // auf Kindprozesse warten
38    for (i=0; i<MAX_WORKERS; i++)
39      waitpid (pids[i], NULL, 0);
40
41    printf ("\n--Fertig--\n");
42    return 0;
43  }
```

a) Lesen Sie zunächst die beiden Programme und überlegen Sie, welches Verhalten bei der Ausführung zu erwarten ist. (Zur Erklärung von zwei vermutlich unbekannten Funktionen:

- Die Funktion `sched_yield()` in Zeile 17 aktiviert den Scheduler, der aufrufende Prozess bzw. Thread gibt damit freiwillig die CPU ab.
- Die Funktion `fflush()` in Zeile 15 sorgt dafür, dass die mit `printf()` ausgegebenen Zeichen sofort im Terminal erscheinen und nicht gepuffert werden.)

b) Starten Sie einen Ubuntu-Container (`hgesser/ubuntu-dev`), laden Sie den Programm-Quellcode herunter, entpacken Sie das Archiv und wechseln Sie in den Ordner `ue08`:

```
cd /realworld
wget swf.hgesser.de/bs-b1/prakt/bs1-ue08.zip
unzip bs1-ue08.zip
cd ue08
```

Übersetzen Sie die Quellcodedateien mit den folgenden Befehlen:

```
gcc -pthread -o worker-threads worker-threads.c
gcc -o worker-prozesse worker-prozesse.c
```

(Sie können auch einfach `make` eingeben – dann werden die obigen Compiler-Aufrufe über das `Makefile` automatisch ausgeführt.)

Führen Sie dann beide Programme mehrfach aus. (Dem Programmnamen stellen Sie beim Aufruf immer `./` voran, damit die Shell das Programm im aktuellen Ordner findet.) Sie sollten Ausgaben erhalten, die den folgenden ähneln:

```
swf@e8421f6cf45e:~/host$ ./worker-threads
--Start--
DEFGHIJCBCBFIGHJECBIFHGJECBIFGHJEBCFIHJGEBCFIHJGEBCFIHJGEBCFIHJGEBCFIHJGEBCFIHJGEDDDDDDDDDAAAAAAAAAA
--Fertig--
swf@e8421f6cf45e:~/host$ ./worker-prozesse
--Start--
DEFGHIJCCIGJFHECIGFJHECIGFJHECIFGJHECIGFJHECFIGJHECFGIJHECFIGJHECFGIJHEDDDDDDDDDBBBBBBBBBBAAAAAAAAAA
--Fertig--
```

(Ziehen Sie das Terminalfenster am besten so breit, dass die Ausgabe zwischen `--Start--` und `--Fertig--` ohne Zeilenumbruch angezeigt wird.)

c) Kommentieren Sie in beiden Quellcode-Dateien die erste Zeile aus der Funktion `worker()` (Zeile 12) durch Voranstellen von `//` aus. Dann ist `i` nicht länger eine lokale Variable der Funktion, und die `for`-Schleife verwendet die globale Variable `i` (aus Zeile 9) als Schleifenvariable.

Übersetzen Sie beide Programme nach dieser Änderung neu und führen Sie mehrere Testläufe durch.

– Welche Veränderung können Sie beobachten,

– und was verursacht diese Veränderung?

d) Passen Sie in beiden Programmen die Funktion `worker()` an, indem Sie vor dem `return`-Befehl in Zeile 19 noch einen Aufruf `sleep(30);` einbauen (welcher die Funktion 30 Sekunden warten lässt). Kompilieren Sie dann beide Programme neu, starten Sie diese und betrachten Sie in einem zweiten Terminalfenster, in dem Sie denselben Docker-Container mit `docker exec … bash` „betreten“ (siehe letzte Übung), die Prozessliste.

Für die Prozess-Variante (`worker-prozesse`) verwenden Sie dazu die Kommandos

```
ps u -C worker-prozesse
```

und

```
pstree -p | grep "worker-"
```

Bei der Thread-Variante (`worker-threads`) geben Sie den Befehl

```
ps -Lf -C worker-threads
```

ein, um auch die Threads zu sehen. `pstree -p` zeigt auch Threads an – wie unterscheidet sich bei diesem Tool die Darstellung der Threads von der Darstellung der Kindprozesse?

Was bedeuten bei den Threads die Angaben in den Spalten `PID` und `LWP`? Mit `man ps` und `man pstree` rufen Sie die eingebaute Dokumentation zu `ps` und `pstree` auf.

2. Threads und fork

In dieser Aufgabe testen Sie, wie es sich auswirkt, wenn Sie `pthread_create()` und `fork()` gemeinsam verwenden.

a) Schreiben Sie ein kleines Testprogramm, das zunächst mit `fork()` den aktuellen Prozess verdoppelt. Vater- und Kindprozess erzeugen anschließend beide mit `pthread_create()` jeweils einen neuen Thread; der Vater führt darin die Funktion `vater()` aus, das Kind die Funktion `kind()`:

```
void *vater () {
  printf ("Neuer Thread im Vaterprozess\n");
  for (;;) ;
}
void *kind () {
  printf ("Neuer Thread im Kindprozess\n");
  for (;;) ;
}
```

Geben Sie auch vor und nach der Thread-Erzeugung eine kurze Meldung à la „Vater-Prozess vor pthread_create“ aus.

Starten Sie das Programm, betrachten Sie die Ausgabe und prüfen Sie (in einem anderen Terminalfenster) mit `ps -eLf`, welche Prozesse und Threads zum laufenden Programm gehören.

b) Jetzt ändern Sie die Reihenfolge: Das zweite Testprogramm erzeugt zunächst mit `pthread_create()` einen neuen Thread. Der neue Thread ruft dann `fork()` auf und beendet sich mit `return`, in der `main()`-Funktion warten Sie mit `pthread_join()` auf dieses Ereignis. Finden Sie auch hier heraus, was passiert – um die Prozesse und Threads mit `ps` beobachten zu können, bauen Sie an geeigneten Stellen wieder eine Endlosschleife `for (;;);` ein.

c) Hausaufgabe: Lesen Sie den Artikel „Threads and fork(): think twice before mixing them“ (über den Moodle-Kurs als PDF-Datei verfügbar) – er enthält eine Erklärung für das in Aufgabe **b)** beobachtete Verhalten.