
Abgabe 2

```
1 /*
2  * This program calculates the time required to
3  * execute the program specified as its first argument.
4  * The time is printed in seconds, on standard out.
5  */
6 #include <stdio.h>
7 #include <unistd.h>
8 #include <stdlib.h>
9 #include <time.h>
10 #include <sys/neutrino.h>
11
12 #define MILLISECONDS_PER_SECOND 1E6
13 #define TIME_TO_WAIT 1000000 //1ms
14
15 struct timespec rqtp, start, stop;
16
17 /*
18  * Prints current frequency of system tick
19  * Changes system tick to nanosecs
20  * Prints new frequency of system tick
21  * System tick indicates the frequency how oft the cpu is interrupted from
22  * the clock
23  * Minimal value is 10 microseconds
24  */
25 int changeSystemTick(unsigned int nanosecs) {
26     struct _clockperiod new, old;
27
28     if (ClockPeriod(CLOCK_REALTIME, NULL, &old, 0) != 0) {
29         perror("clock period");
30         return EXIT_FAILURE;
31     }
32
33     printf("old fract: %ld, old nsec: %ld \n", old.fract, old.nsec);
34
35     new = old;
36     new.nsec = nanosecs;
37
38     if (ClockPeriod(CLOCK_REALTIME, &new, NULL, 0) != 0) {
39         perror("clock period");
40         return EXIT_FAILURE;
41     }
42
43     if (ClockPeriod(CLOCK_REALTIME, NULL, &old, 0) != 0) {
44         perror("clock period");
45         return EXIT_FAILURE;
46     }
47
48     printf("new fract: %ld, new nsec: %ld \n", old.fract, old.nsec);
49
50     return EXIT_SUCCESS;
51 }
52
```

```

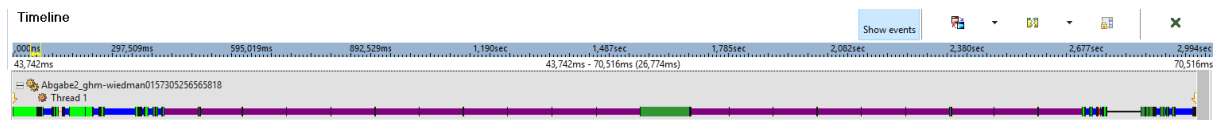
53 /*
54  * Simulates a cycle of TIME_TO_WAIT
55  * Get current time with clock_gettime
56  * For initialization setStart = 1
57  * To simulate an ongoing cycle setStart = 0 for every iteration
58  * Add TIME_TO_WAIT to the start time
59  * With clock_nanosleep the programm sleeps until TIME_TO_WAIT is over
60  */
61 int takt(int setStart) {
62     int s;
63     int ms;
64     int err;
65
66     if (clock_gettime(CLOCK_REALTIME, &start) == -1) {
67         perror("clock_gettime");
68         return EXIT_FAILURE;
69     }
70
71     if (setStart == 1) {
72         rntp = start;
73     }
74
75     rntp.tv_nsec += TIME_TO_WAIT;
76
77     err = clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &rntp, NULL);
78     if (err != 0) {
79         printf("clock_nanosleep: %d \n", strerror(err));
80     }
81
82     if (clock_gettime(CLOCK_REALTIME, &stop) == -1) {
83         perror("clock_gettime");
84         return EXIT_FAILURE;
85     }
86
87     return EXIT_SUCCESS;
88 }
89
90 /*
91  * Set system tick
92  * Simulate ongoing cycle of 1ms
93  * Print the waited miliseconds, check continuity of the results
94  */
95 int main(int argc, char** argv) {
96     changeSystemTick(1 * MILLISECONDS_PER_SECOND);
97     int i = 0;
98     takt(1);
99
100     while(i < 20){
101         takt(0);
102         i++;
103     }
104     int s,ms;
105     s = (stop.tv_sec - start.tv_sec) * 1000;
106     ms = (stop.tv_nsec - start.tv_nsec) / MILLISECONDS_PER_SECOND;
107

```

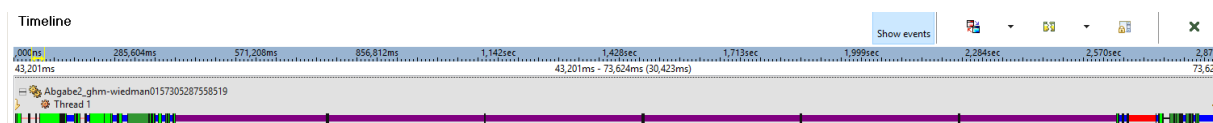
```

108 printf("start: %ld, n %ld \n", start.tv_sec, start.tv_nsec);
109 printf("stop: %ld, n %ld \n", stop.tv_sec, stop.tv_nsec);
110 printf("Waited miliseconds: %d\n", s + ms);
111 return EXIT_SUCCESS;
112 }

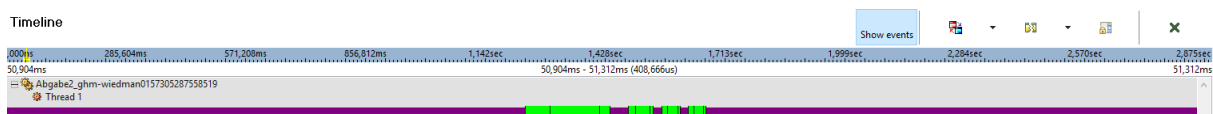
```



Bei Überprüfung des Codes mit dem Kernel Event Tracer ist zu sehen, dass bei einer Durchführung mit System Takt mit $\geq 1\text{ms}$ durch den simulierten Takt immer 1ms geschlafen wird.



Wenn der System Takt $< 1\text{ms}$ wird der simulierte Takt dem System Takt angepasst, da nur alle zb 4ms ein Interrupt durch den Sheduler passiert und das Programm nur bei einem Interrupt aufwachen kann.



Da nur eine ms geschlafen hätte werden sollen, liegt die Aufwachszeit der nächsten x Iterationen in der Vergangenheit. Das heißt das Programm geht nur kurz in den sleep, wo dies dann bemerkt wird und er wieder aufwacht.