# Aufgabe 4:

Dokumentieren Sie die richtige Funktionalität der kooperierenden Tasks durch einen Screenshot des Kernel Event Tracers:


Anhand der Timeline lässt sich erkennen, dass Takt 1 wie in der Aufgabe gestellt alle 4ms mit einer Verarbeitungszeit von 2 ms (hellgrün) läuft.
Takt 2 läuft nach jedem dritten Durchlauf von Takt 1 3 ms.

Semaphoren sind nicht identisch mit Mutexen:

Eine Semaphore kann atomar hoch- und runtergezählt werden und bleibt aber immer über Null ist die Semaphore Null und würde runtergezählt werden so wird gewartet bis sich der Semaphorenwert geändert hat. Semaphoren können jederzeit von jedem Thread hoch- bzw. heruntergezählt werden.

Eine Mutex ist eine spezielle Form der Semaphore bei deren Wert nur Null oder Eins sein kann. Außerdem kann die Mutex nur von dem besitzenden Thread runtergezählt werden. Besitztender Thread ist derjenige Thread der sie zuvor hochgezählt hat.

**Sourcecode**

```c
#include <stdlib.h>
#include <stdio.h>

#include <pthread.h>
#include <errno.h>

#include <semaphore.h>

#define BILLION 1000000000L

void task1(void*);
void task2(void*);
void notBusyWait(int nanos);
void waste_msecs(unsigned int msecs);

sem_t semaphore;

int main(int argc, char *argv[]) {

	pthread_attr_t attr;
	pthread_attr_init(&attr);
	int ret;
	pthread_t task1ID, task2ID;

	if (sem_init(&semaphore, 0, 0)) {
		perror("semaphore");
		return EXIT_FAILURE;
	}

	pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);

	ret = pthread_create(&task1ID, &attr, &task1, NULL);
	if (ret != EOK) {
		fprintf(stderr, "pthread_create: %s\n", strerror(ret));
		return EXIT_FAILURE;
	}
	ret = pthread_create(&task2ID, &attr, &task2, NULL);
	if (ret != EOK) {
		fprintf(stderr, "pthread_create: %s\n", strerror(ret));
		return EXIT_FAILURE;
	}

	ret = pthread_join(task1ID, NULL);
	if (ret != EOK) {
		fprintf(stderr, "pthread_join: %s\n", strerror(ret));
		return EXIT_FAILURE;
	}
	ret = pthread_join(task2ID, NULL);
	if (ret != EOK) {
		fprintf(stderr, "pthread_join: %s\n", strerror(ret));
		return EXIT_FAILURE;
	}

	return EXIT_SUCCESS;
}

void notBusyWait(int nanos) {
	int ret;
```

```
	struct timespec timeOld;

	ret = clock_gettime(CLOCK_REALTIME, &timeOld);
	if (ret != 0) {
		fprintf(stderr, "error time: %s\n", strerror(ret));
		exit(EXIT_FAILURE);
	}

	timeOld.tv_nsec += nanos;
	if (timeOld.tv_nsec >= BILLION) {
		timeOld.tv_sec++;
		timeOld.tv_nsec = timeOld.tv_nsec - BILLION;
	}
	ret = clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &timeOld, NULL);
	if (ret != 0) {
		fprintf(stderr, "error: %s\n", strerror(ret));
		exit(EXIT_FAILURE);
	}
}
void waste_msecs(unsigned int msecs) {
	unsigned int i = 0;
	unsigned int max = msecs * 99843;
	int tmp;
	for (i = 0; i < max; i++) {
		tmp += 1;
	}
}

void task1(void* arg) {
	int i = 0;

	while (1) {
		notBusyWait(2e6);
		waste_msecs(2);
		i = (i + 1) % 3;

		if (i == 1) {
			if (sem_post(&semaphore))
			{
				perror("sempahore post");
				exit(EXIT_FAILURE);
			}
		}
	}
}

void task2(void* arg) {

	while (1) {
		if (sem_wait(&semaphore))
		{
			perror("semaphore wait");
			exit(EXIT_FAILURE);
		}
		waste_msecs(3);
	}

}
```