Table of Contents

1. Zephyr threads communication and synchronization 2
1.1. FIFO queue 2
1.2. Semaphores 5
1.3. Mutexes 6
1.4. Functions for semaphores and mutexes 6
1.5. Tasks for the laboratory 7

1. Zephyr threads communication and synchronization

In this part, we will show methods of exchanging data between threads. In the example from the previous part, threads were defined using K_THREAD_DEFINE()macro.

```
K_THREAD_DEFINE(thread0_id, STACKSIZE, thread0, NULL, NULL, NULL,
        THREAD0_PRIORITY, 0, 0);
```

The arguments of the macro are:

- the name of the thread
- size of the stack that is created for individual use of the thread
- address of the function of the thread
- three initial arguments of the thread which can be passed at the start of the thread (here all NULL)
- constant that specifies the priority of the thread
- thread options (here 0)
- delay to start the thread (0 meaning no delay)

As we can see every thread has its own stack, which means that if there is a need to exchange data between threads we have to use some shared variable visible for all threads interested in using it. In multitasking environments, especially running on a multi-core processor, there is a need to synchronise the access to shared resources. Zephyr offers some mechanisms for inter-thread data exchange and synchronisation. They include three that we present in this laboratory:

- FIFO queue
- Semaphores
- Mutexes

1.1. FIFO queue

In the example available in the Zephyr framework the FIFO queue is used to pass data from two threads (blink0, blink1) which are used to blink two LEDs to the uart_out thread which displays received data in the terminal.

The FIFO queue can be defined with the macro K_FIFO_DEFINE. In our example, it defines the FIFO queue and gives it the name “printk_fifo”.

```
K_FIFO_DEFINE(printk_fifo);
```

The Blink0 thread as well as Blink1 calls the same function blink() passing three parameters:

1. address of the structure that represents LED (&led0 or &led1 respectively)
2. number of milliseconds the function goes to sleep between iterations
3. id of LED (0 or 1)

Notice that blink() is called only once from thread Blink0 and once from thread Blink1. After the initialisation of LED, it jumps into infinite while(1) loop. Every iteration toggles the LED using the least significant bit of the cnt variable

```
gpio_pin_set(spec->port, spec->pin, cnt % 2);
```

and inserts the content of the object tx_data into the printk_fifo using the dynamically allocated memory region of the size equal to printk_data_t size.

```
k_fifo_put(&printk_fifo, mem_ptr);
```

The uart_out thread gets the element from the thread with the function k_fifo_get

```
struct printk_data_t *rx_data = k_fifo_get(&printk_fifo, K_FOREVER);
```

After successful reading, prints the LED number and counter value (cnt) to the console, and releases the memory allocated by blink.

The full code looks as follows. If you face problems with compilation add the following line to the prj.conf file

```
CONFIG_HEAP_MEM_POOL_SIZE=1024
```

```
/*
 * Copyright (c) 2017 Linaro Limited
 *
 * SPDX-License-Identifier: Apache-2.0
 */

#include <zephyr.h>
#include <device.h>
#include <drivers/gpio.h>
#include <sys/printk.h>
#include <sys/__assert.h>
#include <string.h>

/* size of stack area used by each thread */
#define STACKSIZE 1024

/* scheduling priority used by each thread */
#define PRIORITY 7

#define LED0_NODE DT_ALIAS(led0)
#define LED1_NODE DT_ALIAS(led1)

#if !DT_NODE_HAS_STATUS(LED0_NODE, okay)
#error "Unsupported board: led0 devicetree alias is not defined"
#endif

#if !DT_NODE_HAS_STATUS(LED1_NODE, okay)
#error "Unsupported board: led1 devicetree alias is not defined"
```

```
#endif

struct printk_data_t {
    void *fifo_reserved; /* 1st word reserved for use by fifo */
    uint32_t led;
    uint32_t cnt;
};

K_FIFO_DEFINE(printk_fifo);

struct led {
    struct gpio_dt_spec spec;
    const char *gpio_pin_name;
};

static const struct led led0 = {
    .spec = GPIO_DT_SPEC_GET_OR(LED0_NODE, gpios, {0}),
    .gpio_pin_name = DT_PROP_OR(LED0_NODE, label, ""),
};

static const struct led led1 = {
    .spec = GPIO_DT_SPEC_GET_OR(LED1_NODE, gpios, {0}),
    .gpio_pin_name = DT_PROP_OR(LED1_NODE, label, ""),
};

void blink(const struct led *led, uint32_t sleep_ms, uint32_t id)
{
    const struct gpio_dt_spec *spec = &led->spec;
    int cnt = 0;
    int ret;

    if (!device_is_ready(spec->port)) {
        printk("Error: %s device is not ready\n", spec->port->name);
        return;
    }

    ret = gpio_pin_configure_dt(spec, GPIO_OUTPUT);
    if (ret != 0) {
        printk("Error %d: failed to configure pin %d (LED '%s')\n",
            ret, spec->pin, led->gpio_pin_name);
        return;
    }

    while (1) {
        gpio_pin_set(spec->port, spec->pin, cnt % 2);

        struct printk_data_t tx_data = { .led = id, .cnt = cnt };

        size_t size = sizeof(struct printk_data_t);
        char *mem_ptr = k_malloc(size);
        __ASSERT_NO_MSG(mem_ptr != 0);
```

```
        memcpy(mem_ptr, &tx_data, size);

        k_fifo_put(&printk_fifo, mem_ptr);

        k_msleep(sleep_ms);
        cnt++;
    }
}

void blink0(void)
{
    blink(&led0, 100, 0);
}

void blink1(void)
{
    blink(&led1, 1000, 1);
}

void uart_out(void)
{
    while (1) {
        struct printk_data_t *rx_data = k_fifo_get(&printk_fifo,
                                K_FOREVER);
        printk("Toggled led%d; counter=%d\n",
               rx_data->led, rx_data->cnt);
        k_free(rx_data);
    }
}

K_THREAD_DEFINE(blink0_id, STACKSIZE, blink0, NULL, NULL, NULL,
        PRIORITY, 0, 0);
K_THREAD_DEFINE(blink1_id, STACKSIZE, blink1, NULL, NULL, NULL,
        PRIORITY, 0, 0);
K_THREAD_DEFINE(uart_out_id, STACKSIZE, uart_out, NULL, NULL, NULL,
        PRIORITY, 0, 0);
```

Note that there is no main() function in this code, everything happens in thread functions.

1.2. Semaphores

Semaphore is the variable that presents the state of the shared resource. This variable must be available for all the threads interested in using the resource. The common way to implement the semaphore is the global variable. Still, Zephyr OS has the semaphore mechanism built in which has some additional features like controlling the limit value of the semaphore, preventing blocking the ISR handlers. At initialization, there is the possibility to specify an initial state of the semaphore (must be greater than 0) and a maximum limit. Function “Give” will increment the semaphore value up to the maximum limit, in which case the semaphore will not be modified. “Give” can be called from any thread or ISR because it does not stop program execution in any case. Function “Take” will decrement

the semaphore count down to zero (the semaphore is unavailable). “Take” can be called only from threads but not from ISR, because if a thread is trying to “Take” an unavailable semaphore must wait until some other thread makes it available (ISRs can't be blocked anytime). Thus such a thread waits until another thread calls “Give” for the same semaphore.

1.3. Mutexes

Mutexes are similar to the semaphores but present two values only: locked and unlocked. Despite this, they are implemented also as the counter in which zero value means that the resource is unlocked, and any other value means that the resource is locked. If the mutex is locked three times it also must be unlocked three times to make it unlocked. Another feature that differs between mutexes and semaphores is that the mutex can be unlocked only by the same thread which locks it. Mutexes are mainly used to synchronize the critical shared code that can be executed from many threads.

1.4. Functions for semaphores and mutexes

To create the semaphore with the name “my_semaphore”, initial value “0”, and limit “1”, you can use the following macro

```
K_SEM_DEFINE(my_semaphore, 0, 1);
```

Or create the k_sem structure and call the initialization function.

```
struct k_sem my_semaphore;
k_sem_init(&my_semaphore, 0, 1);
```

Giving semaphore can be done with the function:

```
k_sem_give(&my_semaphore);
```

and taking the semaphore with the code as follows:

```
if (k_sem_take(&my_semaphore, K_MSEC(50)) != 0) {
   printk("Semaphore not available!");
} else {
   /* semaphore availabe, execute the code here */
}
```

As you can see the k_sem_take() function has a timeout that can prevent the thread from hanging. This argument can also be K_NO_WAIT (returns immediately) and K_FOREVER (blocks the thread).
To create the mutex you can use the macro

```
K_MUTEX_DEFINE(my_mutex);
```

or the k_mutex structure definition and initialization as for the semaphores:

```
struct k_mutex my_mutex;
k_mutex_init(&my_mutex);
```

Locking the mutex is done with the function:

```
if (k_mutex_lock(&my_mutex, K_MSEC(100)) == 0) {
    /* mutex successfully locked */
} else {
    printk("Can't lock mutex\n");
}
```

Here the timeout can be also K_NO_WAIT and K_FOREVER.
Unlocking the mutex can be done with the function:

```
k_mutex_unlock(&my_mutex);
```

1.5. Tasks for the laboratory

1. Write the program that blinks the LED as many times as it is set in the semaphore (decrementing semaphore with every blink). The semaphore can be given (incremented) by pressing the button. Thus if you press the button three times the LED will blink three times.

From:
https://dokuwiki.zmitac.aei.polsl.pl/ - **RAU6 Dokuwiki**

Permanent link:
https://dokuwiki.zmitac.aei.polsl.pl/en:edu:zmitac:s2:aih:lab:nrf52threads2?rev=1701606280

Last update: **2023/12/03 13:24**