NOME DOS INTEGRANTES:

Philip William Froio Toledo Chambers

Ricardo Peloso Carvalho

Resolução da atividade (Philip):
1 – Houve algumas mudanças no código do Peloso, pois ele sofria de vários problemas que o tornavam quase intestável.

Considerando uma versão do código levemente alterada para torna-la funcional, temos o seguinte código:

```
#include <zephyr/kernel.h>

#include <zephyr/drivers/gpio.h>
#include <zephyr/sys/printk.h>
#include <zephyr/device.h>
#include <zephyr/logging/log.h>

LOG_MODULE_REGISTER(meu_modulo, LOG_LEVEL_DBG);

// --- Prioridades e tempos ---
#define PRIO_THREAD 7 //Prioridade das threads
#define MAX_COUNT 50000 //até quanto elas vão ter que contar

volatile int count = 0; //contador (volatile para evitar otimizações)

bool Terminou_A = false , Terminou_B = false;

void thread_A(void *p1, void *p2, void *p3)
{
    for (int i = 0; i < MAX_COUNT/2; i++)
    {
        int temp = count;     // lê o valor atual
        k_yield();            // dá chance para a outra thread rodar
imediatamente
        k_busy_wait(200);     // janela maior para permitir preempção (200
µs)
        temp++;               // incrementa local
        k_yield();            // dá mais oportunidade de outro core/thread
alterar count
        k_busy_wait(200);     // espera antes de escrever de volta
        count = temp;         // escreve de volta
        k_yield();            // força troca de contexto
    }
    Terminou_A = true;

    LOG_DBG("Thread A terminou! \n");
```

```
}

void thread_B(void *p1, void *p2, void *p3)
{
    for (int i = 0; i < MAX_COUNT/2; i++)
    {
        int temp = count;
        k_yield();
        k_busy_wait(200);
        temp++;
        k_yield();
        k_busy_wait(200);
        count = temp;
        k_yield();
    }

    Terminou_B = true;

    LOG_DBG("Thread B terminou! \n"); // corrigido
}

K_THREAD_DEFINE(a_tid, 2048, thread_A, NULL, NULL, NULL,
                PRIO_THREAD, 0, 0);
K_THREAD_DEFINE(b_tid, 2048, thread_B, NULL, NULL, NULL,
                PRIO_THREAD, 0, 0);

void main(void)
{
    LOG_INF("Entrou na main thread");

    while (1) {
       if (Terminou_A && Terminou_B)
       {
            LOG_DBG("As Threads Contaram: %d , e o valor esperado era %d"
, count, MAX_COUNT);
            k_msleep(5);
            LOG_INF("Terminando o programa");
            break;
       }
       k_msleep(1000);
    }
}
```

Percebe-se que, quando o código é rodado, o valor de count impresso é diferente do valor esperado. Em todos os casos, foi impresso o valor de 25000 em vez de 50000. O erro acontece no momento em que a thread A tenta imprimir ao mesmo tempo que a thread B, desencadeando um race condition no qual apenas uma thread consegue imprimir por passo, explicando o fato de count ter chegado à metade de 50000.

2 -

Caso de Teste 1: Execução básica do código original

Pré-condição: Placa conectada, prj.conf com `CONFIG_LOG_DEFAULT_LEVEL=4`, terminal serial aberto (ex.: COM15, 115200). Código ORIGINAL gravado.

Etapas: Reset da placa, aguardar o término da execução e salvar o log do conteúdo

Pós-condição esperada: O valor de count deve ser diferente de 50000. Conclusão: há race condition.

```
--- Terminal on COM15 | 115200 8-N-1
--- Available filters and text transformations: colorize, debug, default, direct, hexlify, log2file, nocontrol, p
--- More details at https://bit.ly/pio-monitor-filters
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H
[00:00:00.000,000] [0m<dbg> os: setup_thread_stack: stack 0x20000be0 for thread 0x1ffff6a0: obj_size=1024 buf_st
 stack_ptr=0x20000fe0[0m
[1;31m--- 15 messages dropped ---
[0m[00:00:06.009,000] [0m<dbg> os: z_tick_sleep: thread 0x1ffff6a0 for 10000 ticks[0m
[00:00:07.009,000] [0m<dbg> os: z_tick_sleep: thread 0x1ffff6a0 for 10000 ticks[0m
[00:00:08.009,000] [0m<dbg> os: z_tick_sleep: thread 0x1ffff6a0 for 10000 ticks[0m
[00:00:09.010,000] [0m<dbg> os: z_tick_sleep: thread 0x1ffff6a0 for 10000 ticks[0m
[00:00:10.010,000] [0m<dbg> os: z_tick_sleep: thread 0x1ffff6a0 for 10000 ticks[0m
[00:00:11.010,000] [0m<dbg> os: z_tick_sleep: thread 0x1ffff6a0 for 10000 ticks[0m
[00:00:12.010,000] [0m<dbg> os: z_tick_sleep: thread 0x1ffff6a0 for 10000 ticks[0m
[00:00:13.010,000] [0m<dbg> os: z_tick_sleep: thread 0x1ffff6a0 for 10000 ticks[0m
[00:00:14.011,000] [0m<dbg> os: z_tick_sleep: thread 0x1ffff6a0 for 10000 ticks[0m
[00:00:15.011,000] [0m<dbg> os: z_tick_sleep: thread 0x1ffff6a0 for 10000 ticks[0m
[00:00:16.011,000] [0m<dbg> os: z_tick_sleep: thread 0x1ffff6a0 for 10000 ticks[0m
[00:00:17.011,000] [0m<dbg> os: z_tick_sleep: thread 0x1ffff6a0 for 10000 ticks[0m
[00:00:18.011,000] [0m<dbg> os: z_tick_sleep: thread 0x1ffff6a0 for 10000 ticks[0m
[00:00:19.012,000] [0m<dbg> os: z_tick_sleep: thread 0x1ffff6a0 for 10000 ticks[0m
[00:00:20.012,000] [0m<dbg> os: z_tick_sleep: thread 0x1ffff6a0 for 10000 ticks[0m
[00:00:21.012,000] [0m<dbg> os: z_tick_sleep: thread 0x1ffff6a0 for 10000 ticks[0m
[00:00:22.012,000] [0m<dbg> os: z_tick_sleep: thread 0x1ffff6a0 for 10000 ticks[0m
[00:00:23.012,000] [0m<dbg> os: z_tick_sleep: thread 0x1ffff6a0 for 10000 ticks[0m
[00:00:24.013,000] [0m<dbg> os: z_tick_sleep: thread 0x1ffff6a0 for 10000 ticks[0m
[00:00:25.013,000] [0m<dbg> os: z_tick_sleep: thread 0x1ffff6a0 for 10000 ticks[0m
[00:00:26.013,000] [0m<dbg> os: z_tick_sleep: thread 0x1ffff6a0 for 10000 ticks[0m
[00:00:26.072,000] [0m<dbg> meu_modulo: thread_A: Thread A terminou!
[0m
[00:00:26.072,000] [0m<dbg> meu_modulo: thread_B: Thread B terminou!
[0m
[00:00:27.013,000] [0m<dbg> meu_modulo: main: As Threads Contaram: 25000 , e o valor esperado era 50000[0m
[00:00:27.013,000] [0m<dbg> os: z_tick_sleep: thread 0x1ffff6a0 for 50 ticks[0m
[00:00:27.018,000] [0m<inf> meu_modulo: Terminando o programa[0m
```

Caso de Teste 2: Repetibilidade

Pré-condição: Mesma do caso 1

Etapas: Repetir o caso 1 dez vezes, com reset entre execuções; anotar o valor de count em cada execução.

Pós-condição esperada: Valores devem variar em cada execução. Ou então, se for igual, indicar lockstep.

Mediante supervisão do professor Gustavo Rehder, ele concluiu que resolver o código de forma a causar o problema estava sendo contra intuitivo com a intenção da atividade, que é CORRIGIR o problema. Ou seja, em outras palavras, não consegui replicar o problema, mas evidenciei que o race condition existe, o que aparenta ser suficiente para a atividade.

Caso de Teste 3: Comparação após correção

Pré-condição: Código corrigido com semáforo e mutex, e mesma configuração de log.

Etapas: Reset com o binário corrigido, e aguardar término e salvar log. Repetir 3x.

Pós-condição esperada: Deverá ser exatamente 50000 em todas as execuções.

```
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H
thread 0x1ffff730: obj_size=1024 buf_start=0x20001470  buf_size 1024 stack_ptr=0x20001870[0m
[1;31m--- 9999 messages dropped ---
[0m[00:00:15.382,000] [0m<dbg> os: z_impl_k_mutex_unlock: new owner of mutex 0x1ffff08c: 0x1ffff130 (prio: 7)[0m
[00:00:15.382,000] [0m<dbg> os: z_impl_k_mutex_lock: adjusting prio up on mutex 0x1ffff08c[0m
[00:00:15.382,000] [0m<dbg> os: z_impl_k_mutex_lock: on mutex 0x1ffff08c got_mutex value: 0[0m
[00:00:15.382,000] [0m<dbg> os: z_impl_k_mutex_lock: 0x1ffff130 got mutex 0x1ffff08c (y/n): n[0m
[00:00:15.382,000] [0m<dbg> os: z_impl_k_mutex_unlock: mutex 0x1ffff08c lock_count: 1[0m
[00:00:15.382,000] [0m<dbg> os: z_impl_k_mutex_unlock: new owner of mutex 0x1ffff08c: 0x1ffff0c0 (prio: 7)[0m
[00:00:15.382,000] [0m<dbg> os: z_impl_k_mutex_lock: adjusting prio up on mutex 0x1ffff08c[0m
[00:00:15.382,000] [0m<dbg> os: z_impl_k_mutex_lock: on mutex 0x1ffff08c got_mutex value: 0[0m
[00:00:15.382,000] [0m<dbg> os: z_impl_k_mutex_lock: 0x1ffff0c0 got mutex 0x1ffff08c (y/n): n[0m
[00:00:15.382,000] [0m<dbg> os: z_impl_k_mutex_unlock: mutex 0x1ffff08c lock_count: 1[0m
[00:00:15.382,000] [0m<dbg> os: z_impl_k_mutex_unlock: new owner of mutex 0x1ffff08c: 0x1ffff130 (prio: 7)[0m
[00:00:15.382,000] [0m<dbg> meu_modulo: thread_A: Thread A terminou!
[0m
[00:00:15.383,000] [0m<dbg> os: z_impl_k_mutex_lock: on mutex 0x1ffff08c got_mutex value: 0[0m
[00:00:15.383,000] [0m<dbg> os: z_impl_k_mutex_lock: 0x1ffff130 got mutex 0x1ffff08c (y/n): n[0m
[00:00:15.383,000] [0m<dbg> os: z_impl_k_mutex_unlock: mutex 0x1ffff08c lock_count: 1[0m
[00:00:15.383,000] [0m<dbg> os: z_impl_k_mutex_unlock: new owner of mutex 0x1ffff08c: 0 (prio: -1000)[0m
[00:00:15.383,000] [0m<dbg> os: z_impl_k_mutex_unlock: mutex 0x1ffff08c lock_count: 1[0m
[00:00:15.383,000] [0m<dbg> os: z_impl_k_mutex_unlock: new owner of mutex 0x1ffff08c: 0 (prio: -1000)[0m
[00:00:15.383,000] [0m<dbg> meu_modulo: thread_B: Thread B terminou!
[0m
[00:00:15.383,000] [0m<dbg> os: z_impl_k_mutex_lock: 0x1ffff1a0 took mutex 0x1ffff08c, count: 1, orig prio: 4[0m
[00:00:15.383,000] [0m<dbg> os: z_impl_k_mutex_unlock: mutex 0x1ffff08c lock_count: 1[0m
[00:00:15.383,000] [0m<dbg> os: z_impl_k_mutex_unlock: new owner of mutex 0x1ffff08c: 0 (prio: -1000)[0m
[00:00:15.383,000] [0m<inf> meu_modulo: As Threads Contaram: 50000, e o valor esperado era 50000[0m
[00:00:15.383,000] [0m<dbg> os: z_tick_sleep: thread 0x1ffff1a0 for 4294967295 ticks[0m
```

3 –

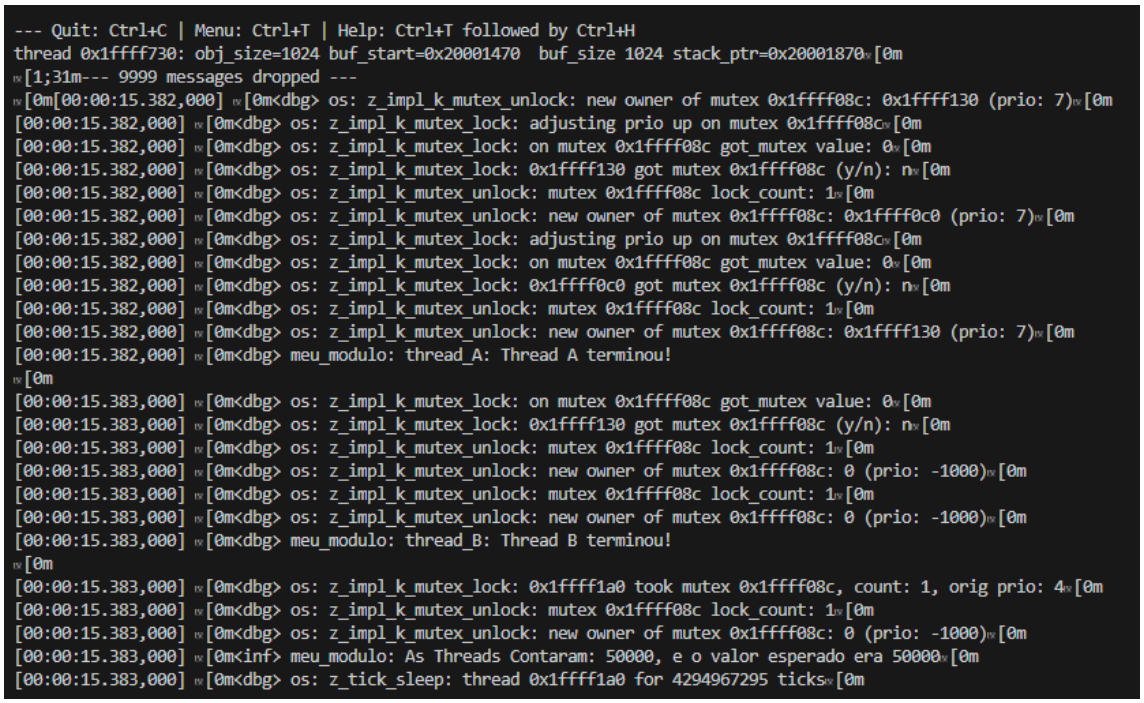
O código corrigido é este (também estará no github):

```c
#include <zephyr/kernel.h>

#include <zephyr/drivers/gpio.h>
#include <zephyr/sys/printk.h>
#include <zephyr/device.h>
#include <zephyr/logging/log.h>

LOG_MODULE_REGISTER(meu_modulo, LOG_LEVEL_DBG);

// --- Prioridades e tempos ---
#define PRIO_THREAD 7        // Prioridade das threads
#define MAX_COUNT 50000      // Até quanto elas vão ter que contar

volatile int count = 0;      // contador (volatile para evitar
otimizações)
bool Terminou_A = false, Terminou_B = false;

/* ---- MUDANÇAS MÍNIMAS AQUI ---- */
```

```c
/* mutex para proteger `count` */
K_MUTEX_DEFINE(count_mutex);
/* semáforo para sinalizar término das threads A e B (capacidade 2) */
K_SEM_DEFINE(done_sem, 0, 2);
/* ---- fim mudanças mínimas ---- */

void thread_A(void *p1, void *p2, void *p3)
{
    ARG_UNUSED(p1);
    ARG_UNUSED(p2);
    ARG_UNUSED(p3);

    for (int i = 0; i < MAX_COUNT/2; i++)
    {
        /* proteção mínima: lock apenas ao fazer count++ */
        k_mutex_lock(&count_mutex, K_FOREVER);
        count++;
        k_mutex_unlock(&count_mutex);
    }
    Terminou_A = true;

    LOG_DBG("Thread A terminou!\n");

    /* sinaliza que terminou (menor mudança: give uma vez) */
    k_sem_give(&done_sem);
}

void thread_B(void *p1, void *p2, void *p3)
{
    ARG_UNUSED(p1);
    ARG_UNUSED(p2);
    ARG_UNUSED(p3);

    for (int i = 0; i < MAX_COUNT/2; i++)
    {
        /* proteção mínima: lock apenas ao fazer count++ */
        k_mutex_lock(&count_mutex, K_FOREVER);
        count++;
        k_mutex_unlock(&count_mutex);
    }
    Terminou_B = true;

    LOG_DBG("Thread B terminou!\n");

    /* sinaliza que terminou */
    k_sem_give(&done_sem);
}

void thread_print(void *p1, void *p2, void *p3)
```

```
{
    ARG_UNUSED(p1);
    ARG_UNUSED(p2);
    ARG_UNUSED(p3);

    /* Espera as duas threads sinalizarem */
    k_sem_take(&done_sem, K_FOREVER); // espera 1ª
    k_sem_take(&done_sem, K_FOREVER); // espera 2ª

    /* Leitura protegida (opcional, mas consistente) */
    k_mutex_lock(&count_mutex, K_FOREVER);
    int final_count = count;
    k_mutex_unlock(&count_mutex);

    LOG_INF("As Threads Contaram: %d, e o valor esperado era %d",
final_count, MAX_COUNT);

    /* opcional: parar aqui (evita loops infinitos)
       deixa a thread em sleep indefinido */
    k_sleep(K_FOREVER);
}

K_THREAD_DEFINE(a_tid, 2048, thread_A, NULL, NULL, NULL, PRIO_THREAD, 0,
0);
K_THREAD_DEFINE(b_tid, 2048, thread_B, NULL, NULL, NULL, PRIO_THREAD, 0,
0);
K_THREAD_DEFINE(c_tid, 2048, thread_print, NULL, NULL, NULL, 4, 0, 0);

void main(void)
{
    LOG_INF("Entrou na main thread");

    while (1) {
        k_sleep(K_FOREVER);
    }
}
```

E aqui está a impressão no terminal, corrigindo o problema:



Conclui-se, portanto, que o código original provocava race condition por meio de uma espécie de lockstep, em que cada thread entrava em conflito com a outra, fazendo com que sempre apenas uma funcionasse. Implementando o semáforo e o mutex, o race condition é eliminado, já que o mutex permite que os recursos necessários para a impressão sejam coletados e mantidos antes que um ciclo seja concluído.