Threads in Zephyr OS.

Today's Agenda:

1. Understanding Threads vs. Zephyr Threads
2. Key APIs for Thread Programming in Zephyr
3. Writing a Simple Thread Program
4. Hands-on Demonstration

What is Thread:

A thread is the smallest unit of execution within a process. It represents a single sequence of instructions that the CPU can execute independently. Threads share the same memory space and resources of their parent process, including code, data, and open files, but maintain their own registers, stack, and program counter.

Difference between Normal Thread and Zephyr Thread :

1) Threads:

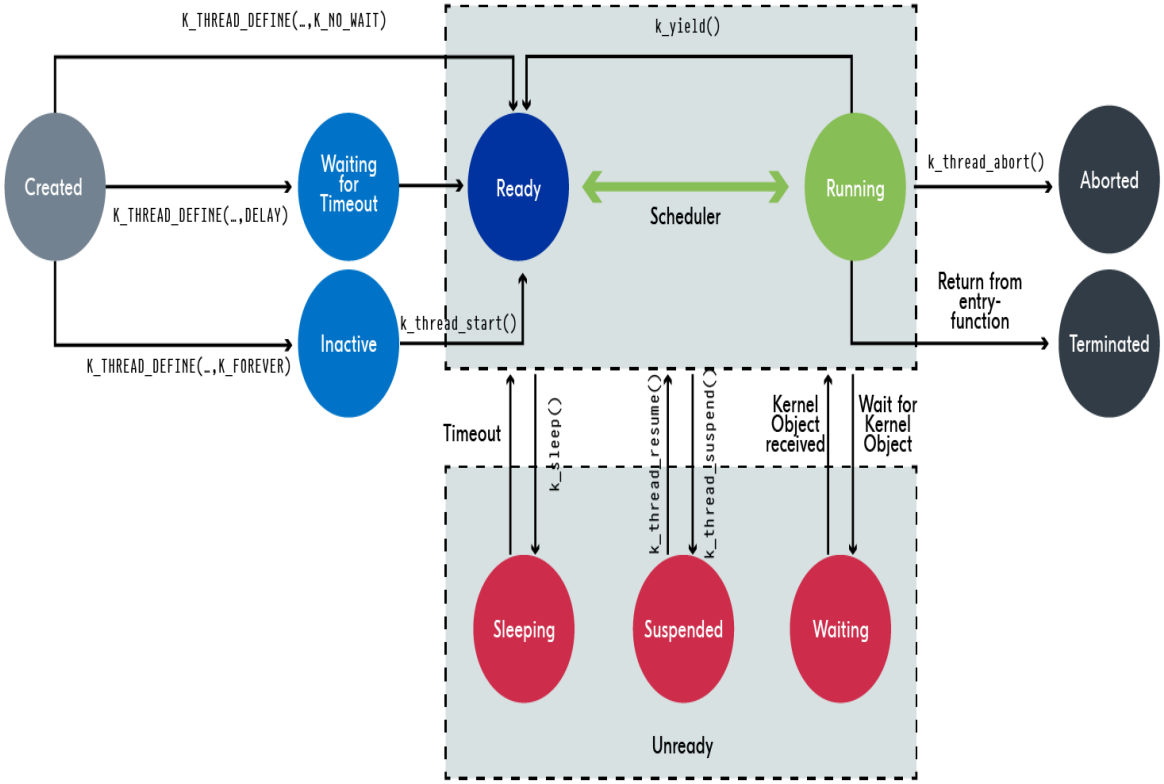
- **General Concept:** A thread is a single unit of execution within a process, allowing for multitasking within an application.

- **Traditional Threads:** In most operating systems, threads are managed by a scheduler that ensures they get time on the CPU.

2) Zephyr Threads:

Zephyr OS provides lightweight threads that can be scheduled to run concurrently, allowing for efficient multitasking even on resource-limited devices.

Key Differences: Zephyr threads are part of the RTOS, so they have real-time characteristics, such as priority scheduling and time-slicing.



Key APIs for Thread Programming in Zephyr

1) Thread Definition & Creation

A. Define stack memory for the thread.

K_THREAD_STACK_DEFINE(name, size);

Parameter Explanation

1)name :-

1. This is just a label or variable name for the stack you're defining.
2. Example: my_stack

2)size :-

1. This is how much memory (in bytes) the stack should have.
2. Must be big enough to fit the thread's needs.
3. Example: 1024 (means 1024 bytes = 1KB stack)

B. Dynamically create and start a thread.

```
k_thread_create(struct k_thread *new_thread,
                k_thread_stack_t *stack,
                size_t stack_size,
                k_thread_entry_t entry,
                void *p1, void *p2, void *p3,
                int priority,
                uint32_t options,
                k_timeout_t delay);
```

Function Call

```
k_thread_create(&my_thread_data,
                My_stack_area,
                STACK_SIZE,
                My_thread_func,
                NULL, NULL, NULL,
                5,
                0,
                K_NO_WAIT);
```

1)new_thread :-

1. This is a pointer to a struct k_thread variable.
2. It acts as a container to store info about the thread you're creating.
3. You must define this before creating the thread.
4. **Example:** &my_thread_data

2)stack :-

1. Pointer to the thread's stack memory (defined using macros like K_THREAD_STACK_DEFINE).
2. You pass the name (not the size) of the stack here.
3. Example: my_stack_area

3)stack_size :-

4. This tells how big the stack is, in bytes.
5. Should match the size you gave when you defined the stack.
6. Example: 1024 (means 1 KB)

4)entry :-

1. This is the function the thread will run.
2. When the thread starts, it begins executing this function.
3. The function must accept 3 void * arguments.
4. Example: my_thread_func

3)p1,p2,p3 :-

1. These are optional arguments you can pass to the thread function.
2. If your thread doesn't need arguments, just pass NULL.
3. Example:NULL, NULL, NULL

4)priority :-

1. Sets the importance level of the thread.
2. Lower number = higher priority
3. Higher number = lower priority
4. Example: 5 (medium priority)

5)options :-

1. Used for special behavior flags.
2. For basic threads, just set it to 0.
3. Example: 0

6)delay :-

1. Controls thread start timing.
2. Use K_NO_WAIT to start immediately.
3. Use K_MSEC(x) to delay for x milliseconds.
4. Example: K_MSEC(1000)

C. Starts a suspended thread.

k_thread_start(struct k_thread *thread);

1. If you create a thread using k_thread_create() with a delay like K_FOREVER, the thread will remain in a suspended state and will not start automatically.

2. You can manually start it later using k_thread_start().

1)thread :-

1. This is a pointer to the thread object you want to start.
2. It must be a thread that was created earlier and hasn't started yet (i.e., it's in the suspended state).
3. **Example:** &my_thread_data

2) Thread Control

a) Pause a thread.

k_thread_suspend(k_tid_t thread_id);

1. This is a Zephyr OS function used to pause (suspend) a thread.
2. Once suspended, the thread will stop executing and will not be scheduled again until it is manually resumed using **k_thread_resume().**

Parameter Explanation:

Thread_id

1. This is the thread identifier of type k_tid_t.
2. It is usually obtained when the thread is created using k_thread_create().
3. You can suspend any active or ready thread, but not the currently running thread itself, as doing so may cause undefined behavior.

b) Resume a suspended thread.

k_thread_resume(struct k_thread *thread);

1. This is a Zephyr function used to resume (restart) a thread that was previously suspended using k_thread_suspend().
2. Once resumed, the thread goes back to the ready-to-run state and can be scheduled by the system again.

Parameter Explanation:

Thread

1. A pointer to the thread object (struct k_thread) that you want to resume.
2. The thread must have been suspended earlier — if it's not suspended, calling this has no effect.
3. **Example**: &my_thread_data

c) Stop a thread permanently.

k_thread_abort(struct k_thread *thread);

1. This is a Zephyr function used to terminate a thread that is currently running or suspended.
2. Once called, the thread is aborted, and it will never resume or run again.

Parameter Explanation:

Thread

1. A pointer to the thread object (struct k_thread) that you want to abort.
2. The thread can be in any state — running, suspended, or ready — but once this function is called, the thread will be terminated permanently.
3. **Example:** &my_thread_data

C) Change thread priority

k_thread_priority_set(struct k_thread *thread, int priority);

1. This Zephyr function is used to change the priority of an already running thread.
2. You can use this function to increase or decrease the importance of a thread during runtime, affecting its scheduling behavior.

Parameter Explanation:

Thread

1. A pointer to the thread object (`struct k_thread`) whose priority you want to change.
2. The thread must be currently running when calling this function.
3. **Example:** `&my_thread_data`

priority

1. The new priority value for the thread.
2. Lower values represent higher priority (priority 1 is the highest).
3. Higher values represent lower priority.
4. **Example:** 3 (medium priority)

3) Timing & Sleep

a. Sleep for milliseconds.

k_msleep(int ms);

1. This Zephyr function is used to pause the execution of the current thread for a specified number of milliseconds.
2. It's similar to saying, "Sleep for a while" before continuing with the next operation in the thread.

Parameter Explanation:

Ms

1. The number of milliseconds you want the thread to sleep (pause execution).
2. You can pass an integer value to specify the sleep duration.
3. Example: 500 (the thread sleeps for 500 milliseconds, or half a second)

B. Sleep using kernel time units.

k_sleep(K_MSEC(ms));

1. This Zephyr function is similar to k_msleep(), but it uses a more flexible time format.
2. It allows you to specify the sleep time in milliseconds using a macro called K_MSEC(ms), which converts the time into a format that k_sleep() can understand.
3. In essence, it tells the thread to pause execution for a specified duration.

Parameter Explanation:

K_MSEC(ms)

1. This is a macro that converts the number of milliseconds (ms) you provide into a time duration format that k_sleep() can understand.
2. **For example :** K_MSEC(500) converts 500 milliseconds into the correct format that k_sleep() uses internally.

k_sleep() function:

1. The k_sleep() function is used to pause the current thread for the specified time duration (in this case, milliseconds).
2. It is commonly used to control the timing between actions or to periodically wake up a thread after a delay.

Writing a Simple Thread Program

Your project directory should look like this:

~/zephyrproject/simple_thread$/

├── CMakeLists.txt

├── prj.conf

└── src/

└── main.c ← (Your Application code)

1)main.c ← (Your Application code)

```
// -------- Header Files --------

#include <zephyr/kernel.h>          // Core kernel APIs: threads, sleep, etc.
#include <zephyr/drivers/gpio.h>    // GPIO control (included for potential expansion)
#include <zephyr/sys/printk.h>      // printk() for debug logging to console
#include <zephyr/sys/atomic.h>      // Atomic operations (for sync_counter)

// -------- Thread Configuration --------

#define STACK_SIZE 512              // Stack size for each thread
#define THREAD_PRIORITY 5           // Thread priority (lower number = higher priority)

// -------- Thread Stack & Data --------

K_THREAD_STACK_DEFINE(stack1, STACK_SIZE);     // Stack for Thread 1
K_THREAD_STACK_DEFINE(stack2, STACK_SIZE);     // Stack for Thread 2

struct k_thread thread1_data;      // Thread structure for Thread 1
struct k_thread thread2_data;      // Thread structure for Thread 2

// -------- Synchronization Utilities --------
```

```
atomic_t sync_counter = ATOMIC_INIT(0);  // Atomic counter to track thread output
struct k_mutex sync_mutex;               // Mutex to protect newline print
/**
 * @brief Tracks how many threads have printed their message.
 * Once all 3 (T1, T2, and MAIN) have printed, a newline is printed for clarity.
 */

void sync_log_complete_cycle(void) {
    atomic_inc(&sync_counter);                // Increment shared counter
    if (atomic_get(&sync_counter) >= 3) {     // If all 3 threads printed once
        k_mutex_lock(&sync_mutex, K_FOREVER);  // Lock before printing newline
        atomic_set(&sync_counter, 0);         // Reset the counter
        printk("\n");                         // Print blank line for separation
        k_mutex_unlock(&sync_mutex);          // Unlock after printing
    }

}

/**

 * @brief Function executed by Thread 1

 */

void thread1_fn(void *a, void *b, void *c)
{
    while (1) {
        printk("[T1] Thread 1 is running...\n");
        sync_log_complete_cycle();            // Track thread log activity
        k_sleep(K_MSEC(1000));                // Sleep 1000ms
    }
}

/**

 * @brief Function executed by Thread 2

 */

void thread2_fn(void *a, void *b, void *c)
{
    while (1) {
        printk("[T2] Thread 2 is working...\n");
        sync_log_complete_cycle();            // Track thread log activity
        k_sleep(K_MSEC(1500));                // Sleep 1500ms
    }
}

/**

 * @brief Entry point of the application.
```

```
 */

void main(void)
{
   printk("[MAIN] Starting Two-Thread Demo\n");

   // Create Thread 1
   k_tid_t tid1 = k_thread_create(&thread1_data, stack1,
      K_KERNEL_STACK_SIZEOF(stack1),
      thread1_fn,
      NULL, NULL, NULL,
      THREAD_PRIORITY, 0, K_NO_WAIT);
   printk("[MAIN] Thread 1 created\n");

   // Create Thread 2
   k_tid_t tid2 = k_thread_create(&thread2_data, stack2,
      K_KERNEL_STACK_SIZEOF(stack2),
      thread2_fn,
      NULL, NULL, NULL,
      THREAD_PRIORITY, 0, K_NO_WAIT);
   printk("[MAIN] Thread 2 created\n");

   // Main thread loop
   while (1) {
      printk("[MAIN] Main thread monitoring...\n");
      sync_log_complete_cycle();            // Track main thread log
      k_sleep(K_MSEC(2000));                // Sleep 2000ms
   }
}
```

2. Prj.conf

```
# Enable printk() function for printing messages to the console.
# Useful for debugging or logging from threads and ISRs.

CONFIG_PRINTK=y

# Enable the system console. Required for outputting messages using printk
# Without this, printk() won't have a destination to send output.

CONFIG_CONSOLE=y

# Allow threads to be named using k_thread_name_set().
# Thread names help with debugging and can be viewed with tools like thread analyzers or logging.

CONFIG_THREAD_NAME=y
```

3. CMakeLists.txt

```
# Specify the minimum required version of CMake for this project.
# Zephyr requires at least version 3.20.0 for compatibility.

cmake_minimum_required(VERSION 3.20.0)

# Locate the Zephyr SDK and import Zephyr's build system.
# $ENV{ZEPHYR_BASE} refers to the environment variable that points to the Zephyr installation directory.

find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})

# Define the name of your project.
# This name is used internally by the build system and doesn't affect the final output file name.

project(simple_thread)

# Specify the source files for your application.
# This tells CMake to compile 'main.c' and link it into the final firmware binary.
# 'app' refers to the application target provided by Zephyr.

target_sources(app PRIVATE src/main.c)
```

Summary

Zephyr OS is a modern RTOS designed for embedded applications. Setting it up requires installing key tools, configuring a Python environment, initializing the project using west, and building and flashing samples onto supported boards. It supports multiple boards and simplifies development through modular structure and build tools.