# Scheduling

## What is Scheduling in Zephyr OS?

**Scheduling** decides which thread (a small task or program) gets to run on the CPU. Since multiple threads may want the CPU at the same time, the scheduler picks one to run.

## Basic Concepts

### **Current thread**:

The one that is running right now.

### **Reschedule points**:

Moments when the scheduler checks if another thread should run, such as:

1) When a thread goes to sleep (e.g. k\_sleep()).

**-** Function prototype -

**int32\_t k\_sleep(k\_timeout\_t timeout);**

- timeout — how long the thread should sleep, passed as a *k\_timeout\_t* object.

- Returns 0 ,if the thread sleeps for the mentioned time .

- Returns time left , if the thread was woken up early by something like *k\_wakeup().*

2) When a thread becomes ready to run (e.g. k\_thread\_start()).

-Function prototype -

**static void k\_thread\_start( k\_tid\_t thread );**

- k\_tid\_t thread — This is the ID or handle of the thread you want to start.

3) After handling an interrupt.

4) When a thread yields the CPU voluntarily (*k\_yield()*).

- Function prototype -

**void k\_yield(void);**

When the CPU switches threads, it **saves** the old thread’s data and **restores** it when that thread runs again.

## How the Scheduler Chooses a Thread

* Pick the highest priority ready thread.
* If two threads have the same priority, the one waiting the longest runs first.
* You can enable deadline-based scheduling (via *CONFIG\_SCHED\_DEADLINE*) to prefer the thread with the earliest deadline.
* Interrupts (ISR) always get priority over threads.

## Types of Ready Queue Implementations[Scheduling Algorithm]

### **Simple Linked List (***CONFIG\_SCHED\_SIMPLE***)**:

* + Threads that are ready to run are stored in a basic, unordered linked list.There's no sorting by priority or other complex rules.The scheduler goes through the list and picks the next runnable thread.

#### **Pros:**

* Very small code size (saves ~2 KB of flash).
* Very fast when there are only 1–3 threads.
* Ideal for small or simple embedded devices (like MCUs with limited memory).

#### **Cons**:

* Not efficient for systems with many threads.
* No priority-based selection—just picks the next thread in the list.
* Poor performance if many threads are competing for the CPU.

#### Points to be remembered -

- CONFIG\_SCHED\_SIMPLE makes the scheduler super lightweight.

- Great for small devices with few threads.  
 - Avoid if you need priority-based or complex scheduling.

### **Red-Black Tree (***CONFIG\_SCHED\_SCALABLE***)**:

* + It’s a special kind of **binary search tree** that keeps itself balanced by following rules when nodes (threads) are added or removed.
  + This structure represents the ready queue implemented using a Red/Black Tree.Each node is a thread that is ready to run.Nodes are either red or black, reflecting the balancing rules of a red-black tree.The threads are organized based on priority (higher priority threads are toward the left/top).

#### Pros:

* Scales well: Efficient even with 1000+ threads.
* Supports priority-based scheduling.
* Ensures predictable performance as the system grows.

#### Cons:

* Slightly slower insert/remove operations compared to simple lists.
* Adds about ~2 KB extra code size to your firmware.
* More complex code, not suitable for small devices.

### **Multi-Queue (***CONFIG\_SCHED\_MULTIQ***)**:

* + The **multi-queue implementation** (*CONFIG\_SCHED\_MULTIQ*) organizes this queue as an **array of linked lists**, with **one list for each priority level**.

### 

### **Example:**

* If there are threads with priority:
* Thread A – priority 1
* Thread B – priority 2
* Thread C – priority 1

The ready queue would look like:

Priority 1 → [Thread A] → [Thread C]

Priority 2 → [Thread B]



The scheduler will pick from priority 1 first, as it's higher than 2.

#### Pros-

* Fast scheduling — very low overhead, O(1) in most cases.
* Simple to understand — matches traditional textbook algorithms.
* Small code size — efficient in terms of flash usage.

#### Cons-

* More RAM usage — you need memory for each priority level's list (even if empty).
* Not compatible with advanced features like:
* Deadline scheduling (where you need to sort by time)
* SMP (Symmetric Multi-Processing) affinity (where threads may be bound to specific CPUs)

## Time Slicing:

* CPU time is split into slices.
* When time's up, the thread may get paused so another one of the same priority can run.
* Threads with **high priorities** can **skip** time slicing (they are not interrupted).

### 

### 

### 

### Cooperative Time Slicing

* A cooperative thread keeps running until it gives up the CPU (voluntarily).
* Useful for important tasks like device drivers.
* It gives up CPU by:  
  + *k\_yield()*: Gives others a chance, but will resume soon.
  + *k\_sleep()*: Goes to sleep for a set time.

**Note -** If it runs too long, it can delay other threads.

### Preemptive Time Slicing

* When a preemptive thread is running, it keeps control of the CPU until:
  + A higher priority thread becomes ready, or
  + It does something that makes it unready (like sleeping or waiting).

## Scheduler Locking

* A thread can lock the scheduler (*k\_sched\_lock()*), so it won’t get interrupted during critical operations.
* It must unlock it later (*k\_sched\_unlock()*).
* Even if the thread goes to sleep, the lock is remembered.

## 

## Thread Sleeping

* *k\_sleep(time)*: Pause for some time and let others run.
* *k\_wakeup(thread)*: Another thread can wake it up early.
* If the thread is not sleeping, *k\_wakeup()* does nothing.

## Busy Waiting

* *k\_busy\_wait(time)*: Waits without giving CPU to anyone else.
* Used for very short delays where switching threads is not worth it.

## When to Use What

| | **Thread Type** | **Use for** | | --- | --- | | **Cooperative** | Device drivers, short tasks, custom mutual exclusion. | | **Preemptive** | Time-sensitive tasks, like real-time sensor data. | |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |
|  |  |

# 

# 

# 

# CPU Idling

## What is CPU Idling?

CPU idling means putting the processor into a low-power or sleep state where it pauses all operations until something important (usually an interrupt) wakes it up.

Think of it like this:

* If there's nothing to do, the CPU takes a nap to save energy, and it wakes up only when someone rings the bell (i.e., an interrupt).

## Concepts

Normally, Zephyr uses a special idle thread whose only job is to put the CPU to sleep when no other threads are ready to run.

However, in some situations (like power-constrained devices), your own thread may want to idle the CPU instead of relying on the default idle thread.

## How to Make the CPU Idle

* k\_cpu\_idle()
* Tells the CPU to sleep until an event (like an interrupt) occurs.
* Often used in an infinite loop.
* Interrupts get unmasked automatically when it returns.

**Example (Simple Case):**

static k\_sem my\_sem;

void my\_isr(void \*unused)

{

k\_sem\_give(&my\_sem); // Wake up the main thread when an interrupt happens

}

int main(void)

{

k\_sem\_init(&my\_sem, 0, 1); // Semaphore starts at 0

for (;;) {

if (k\_sem\_take(&my\_sem, K\_NO\_WAIT) == 0) {

// Do work triggered by ISR

}

k\_cpu\_idle(); // Sleep until next interrupt

}

}



**Problem**: There is a risk of a race condition:

* The interrupt might happen between checking the semaphore and calling k\_cpu\_idle(). In this case, the CPU misses the event and sleeps forever.

## How to Idle Atomically

* k\_cpu\_atomic\_idle()
* Use this to avoid race conditions.
* Lock interrupts while checking the semaphore and idling.
* Make sure the CPU won’t miss any interrupt before going to sleep.

**Fixed Example (Safe Version):**

for (;;) {

unsigned int key = irq\_lock(); // Disable interrupts

if (k\_sem\_take(&my\_sem, K\_NO\_WAIT) == 0) {

irq\_unlock(key); // Got semaphore, unlock interrupts

// Do processing

} else {

k\_cpu\_atomic\_idle(key); // No work? Idle safely while still locked

}

}



**Note -** This ensures that if an interrupt happens, it will be handled after waking up. The system won’t hang.

## When to Use Which?

| **Use Case** | **Function** |
| --- | --- |
| Thread only idles the CPU (does nothing else) | k\_cpu\_idle() |
| Thread does real work + idles | k\_cpu\_atomic\_idle() |

**Example for k\_cpu\_idle():**

int main(void)

{

// Init code here...

for (;;) {

k\_cpu\_idle(); // Just sleep to save power

}

}



## Summary

* *k\_cpu\_idle()* = Basic CPU sleep. Use it only when no work is being done.
* *k\_cpu\_atomic\_idle()* = Safe sleep when you’re also doing useful processing in the same thread.
* These APIs are useful in low-power systems to save energy when nothing important is happening.