A Real-Time Operating System (RTOS) is a layer of software that is utilized to provide various general-purpose functionalities that can then be utilized by an application-specific layer of software above it.

One primary functionality provided by an RTOS is multithreading (i.e., being able to run multiple tasks/threads concurrently); while only one thread can physically be running at a time (unless the processor has multiple processing cores), the RTOS makes it appear that multiple threads are running at the same time by switching between them.

An RTOS is often utilized to meet application-specific real-time requirements of software. Note that real-time does not necessarily mean fast, but only fast enough for the particular application. Often in embedded systems, more important than speed is predictability/consistency; the time to respond to an event (e.g., responding to an interrupt) is latency while variability in the time to respond to an event is jitter.

Commonly used real-time operating systems in embedded applications include FreeRTOS, ChibiOS, embOS, Keil RTX, etc. In embedded applications wherein more memory (e.g., at least a few MB) is available, RTLinux (a real-time Linux-based RTOS) is also commonly used.
Concurrent threads (also called tasks in an embedded RTOS context) are very useful in applications where an embedded microcontroller is required to do multiple things in parallel, some of which require waiting for an external device/resource. For example, if the microcontroller needs to read a sensor periodically and needs to wait for the sensor to send the data (using polling instead of interrupts), the waiting time can be utilized to run other code if a separate task can run during that waiting time.

For example, a task might have a `while (1)` loop. Giving a task a chance to run for some time means that the function corresponding to that task continues to run for some amount of time and then later, when the task is given the next chance to run, the processor continues from the spot in the function where it had stopped (i.e., the program counter at the time when that task was put into the background and another task started running).

Each task is essentially a function (e.g., a C/C++ function `func`). If the RTOS is given a set of functions (as function pointers), it can switch between the functions either based on elapsed time (each task gets some time to run and then a different task gets a chance to run) or by letting the functions themselves decide when to go into the background (i.e., give another task a chance to run). These two methods are called preemptive multithreading and non-preemptive (cooperative) multithreading, respectively.
Preemptive and Non-Preemptive Multithreading

- Preemptive multithreading: the RTOS gives each task a certain amount of time to run and then that task is put into the background while another task is given a chance to run; many RTOS implementations provide functionalities to set task priorities that will then be used to determine how often or how long a task gets to run.

- Non-preemptive (cooperative) multithreading: a task utilizes a function (often called *yield* or *sleep*) to tell the RTOS that it is ready at that time to go to sleep (i.e., to go to the background and wait for the next chance to continue running) and let another task run. This is *cooperative* multithreading since the tasks cooperate to share the processor.
Thread Switching and Latency

- When switching between tasks, the RTOS saves the context of the task so that the same context can be restored when the task again gets a chance to run.

  As part of the context saving, the contents of the processor registers are saved into memory; the processor registers include the program counter (so that the task knows the exact spot in the function that was being executed when control was transferred to another task so that it can continue executing from the same spot), the condition flags (in the application program status register), stack pointer, the general-purpose registers, etc.

- To enable multiple concurrently running tasks to use stacks, an RTOS usually provides separate per-task stacks (so that each task has a separate region of memory that it utilizes as its stack); when switching between tasks, the stack pointer is updated to point to the top of the corresponding task’s stack.
Thread Switching and Latency

- Thread switching latency: The time required to switch between tasks is called the *thread switching latency* or the *context switching latency*.

- Interrupt latency: The time elapsed between when an external event occurs (e.g., a byte arriving on a serial port) and when the interrupt service routine is called; in real-time applications, it is important to keep this time delay predictable (e.g., a guaranteed max delay). For this, it is good to keep the interrupt service routines short (e.g., just set a flag or increment some variable in the interrupt service routine and then continually check the value of this flag/variable in a separately running task).
Mutexes and Semaphores

- If multiple concurrently running tasks (i.e., threads) need to access a variable shared between them, then some mechanism should be used to ensure that the access to this shared variable is properly coordinated between the tasks so that tasks do not overwrite each other’s updates to the shared variable or access inconsistent shared data (e.g., if two tasks both access a struct and one task has finished updating only half the data members of the struct when the RTOS switches control to the second task which tries to read the struct and thus gets inconsistent data).

- Using a flag (e.g., having one task set a flag `dataUpdated` to 0 before updating a shared variable and then setting the flag to 1 after updating and having the other task check the value of the flag before accessing the shared variable) does not work since in preemptive multithreading, the RTOS can switch between tasks at any time (since each task gets some amount of time to run and then the RTOS automatically switches to another task). In particular, the RTOS can possibly even switch tasks between the time that the flag is checked and before it is updated (e.g., if we have code such as `if (dataUpdated == 0) { dataUpdated = 1; ... }`), the RTOS can possibly switch tasks between the time the comparison `(dataUpdated == 0)` is done and the time `dataUpdated = 1;` is executed. Hence, to properly coordinate access to a shared variable (or a shared resource), some mechanism provided by the RTOS itself must be used (so that the RTOS can ensure that this mechanism itself has atomic functions, i.e., an unpredictable task switch does not occur during the time that one of the functions for this mechanism is being called).

- Semaphores and mutexes are general mechanisms for this purpose provided by most RTOS implementations.
Mutexes and Semaphores

- A mutex (mutual exclusion) is like a lock that can be *acquired* or *released*. So, for example if we have two tasks that need to access a shared variable (e.g., one task writing to it and another task reading from it) and we want to ensure that the two tasks do not access the shared variable at the same time, we can do the following in each task:
  - Acquire the mutex
  - Access the shared variable (read or write)
  - Release the mutex

- If one task has acquired (locked) the mutex and not yet released (unlocked) it and a second task wants to acquire it, then the second task will automatically be placed in a wait queue by the RTOS so that the second task can continue only after the first task has released the mutex.

- A mutex can be utilized to coordinate access to a shared variable or, in general, any shared resource (e.g., if we have two tasks that are printing to a screen and we want to ensure that the characters within strings being printed by the two tasks do not appear intermingled on the screen).

- In RTOS implementations, the function to acquire a mutex is also sometimes called `wait` (e.g., `osMutexWait`) or `lock`. 
 Mutexes and Semaphores

A semaphore is like a counter that can count to some arbitrary number (which is usually related to a number of available resources). A mutex is essentially a *binary* semaphore (i.e., only counts between 0 and 1, i.e., locked and unlocked).

For example, if we are utilizing a software library to write files to an SD card, but the library can only handle a specific number (e.g., 4) of files at the same time, then we can utilize a semaphore that is acquired (i.e., decrementing of the semaphore counter) by any task that wants to write a file and is released (i.e., incrementing of the semaphore counter) after the file has been written. Then, if the max number of files are already open, a task that wants to write another file will automatically be made to wait by the RTOS when the task tries to acquire the semaphore.
Mutexes and Semaphores

- A semaphore has an associated counter variable $C$ that is decremented when the semaphore is acquired and incremented when the semaphore is released, i.e., the counter keeps track of how many of a set of available resources is available. When a task tries to acquire a semaphore, if the semaphore counter would become negative by decrementing by 1, the task is automatically placed on a wait queue (the semaphore wait queue). When a task releases a semaphore, if a task is on the semaphore wait queue, that waiting task will be given a chance to run.

- In RTOS implementations, the function to acquire a semaphore is also sometimes called wait (e.g., `osSemaphoreWait`) or lock. The function to release a semaphore is also sometimes called signal.

- From an application programmer’s perspective, we just need to ensure that we lock a corresponding mutex/semaphore before accessing a shared variable and then unlock the mutex/semaphore after accessing it.
Fixed-Size Memory Pool

- When variables are dynamically allocated (e.g., using `malloc` or `new`) and de-allocated (e.g., using `free` or `delete`), *heap fragmentation* can occur over time wherein there is free memory available, but not in adequately sized contiguous regions. Fixed-size memory pools are a technique to avoid this heap fragmentation.

- In a fixed-size memory pool, memory chunks of the same fixed size are allocated/deallocated, i.e., when creating a memory pool, the size (in bytes) of each *item* is specified and then each allocation/deallocation from the memory pool is always of this fixed size. Hence, any free memory within the memory pool is always in chunks of this fixed size, thus preventing fragmentation.

- For example, if the item type is a struct (or a class) that includes two 32-bit integers and two 32-bit floats, the size of each item is 16 bytes (this can be calculated during compile-time using the `sizeof` operator).
With a fixed-size memory pool, since all allocations and deallocations are of the same fixed size (e.g., 16 bytes), there will not be a fragmentation since any free memory will definitely be of size sufficient to be able to allocate the required number of bytes for an item from it.

A memory pool is basically initialized as a memory region (which is essentially equivalent to an array) of some size and then the memory pool manager allocates fixed-size chunks from it whenever requested by the application.

Since a fixed-size memory pool is a useful functionality for many embedded applications, a memory pool mechanism is often provided by a real-time operating system as part of its functionalities.