**Real-Time Operating System (RTOS) Concepts**

What is the difference between a traditional operating system (OS) like Linux and a real-time operating system (RTOS) like Zephyr, VxWorks, ThreadX, or FreeRTOS? The “real-time” part of the name tells a lot of the story. Traditional OSes like Linux, OSX, and Windows are large software systems that run on powerful computing platforms and can run many applications, graphical and terminal-based, simultaneously. These OSes do a lot of housekeeping in the background, organizing pools of memory for use by applications (“garbage collection”), maintaining caches of data to be written to storage devices such as hard drives or solid-state drives, and so on. By virtue of the powerful processors in personal computers, applications generally appear to be very responsive, accepting input from a keyboard and mouse and providing calculated and graphical output quickly.

However, we have all experienced situations where the responsiveness of one of our apps lags, due to the OS devoting an inordinate amount of computing power to other apps or to background tasks.

RTOSes, on the other hand, tend to be much more compact (in terms of their requirements for computing power, non-volatile memory, and volatile memory) and are designed for situations where it is imperative that certain operations are always executed within a predictable and consistent period of time.

The terms and characteristics used to describe these two different types of operating systems are:

* **Deterministic** – RTOSes such as Zephyr, VxWorks, ThreadX, and FreeRTOS, are deterministic in their latency to respond to asynchronous events, such as data being received by a communications peripheral
* **Non-deterministic** – OSes such as Linux, OSX, and Windows are non-deterministic in how quickly they respond to asynchronous events.
* **Latency –** The maximum time specified by an RTOS to respond to an asynchronous event, such as an interrupt that results when data is received by a communications peripheral.

In general, RTOSes are **compact** and **modular** and **configurable** to include only the modules needed for a particular application. Although an RTOS can run on powerful Intel or ARM processors like the ones found in personal computers, they are generally used in embedded devices with less processing power, non-volatile memory (flash), and volatile memory (RAM). Once configured for a particular application, RTOSes are generally **static in the support they provide for a fixed set of peripherals**.

In general, OSes are **large**, **monolithic**, and come **pre-configured** with many modules to support the myriad of computer peripherals in common use. They run on powerful Intel or ARM processors with large non-volatile memory (solid-state drives with capacities of 512GB and up) and volatile memory (DRAM with capacities of 8GB and up). OSes offer **dynamic support for peripherals by way of loadable device drivers**.

In COMP-GENG 421 we learned “bare-metal” programming of microcontrollers using state machines in the foreground to divide up servicing peripherals and running the application code, and interrupts in the background to provide low-latency response to asynchronous events. Why do we need an RTOS?

First, for small projects running on constrained microcontrollers, the bare-metal approach works fine and is the most efficient approach. The bare-metal approach is definitely “what you see is what you get” because you write all of the code – the initialization of the GPIOs and peripherals, the main loop, the state machines, and the interrupt service routines. So long as all the state machines execute within a tick and the ring buffers (FIFOs) between the foreground and background code are sufficient in size for the worst-case situation, the bare-metal approach is straightforward and lean and mean.

There are two factors when deciding between the bare-metal approach and using an RTOS for an embedded application:

* **Use of a communications stack** – If your embedded design includes a standard communications peripheral such as a Bluetooth radio, a WiFi radio, or a USB port, it will require a communications **stack** to implement the protocol required for **interoperability** with other such peripherals. Implementing a **Bluetooth stack**, or a **WiFi stack**, or a **USB stack**, is a LOT of work. These communications protocols have been established by standards bodies over many years and are updated periodically as new features are added and as problems are fixed. The manufacturers of these communications peripherals (which are sometimes embedded into a microcontroller and sometimes are external and connected to a microcontroller communications bus, such as I2C or SPI) realize no one will use their peripheral if they don’t supply a stack with it. As a result, they select one or more RTOSes and write a communications stack for their peripheral, saving the embedded developer a great deal of time and hassle. Further, they will often get their stack certified by a standards body, ensuring interoperability with other devices.
* **Complexity and flexibility of the embedded application code** – If you expect the embedded application to be **relatively complex**, or if you feel it will **grow, expand, and change over time**, or if you are developing **several variations** of an embedded device, each with a different set of features or peripherals, then you may want to go with an RTOS, which provides you with **higher-level constructs** than the bare metal approach.

Let’s discuss the main RTOS concepts and constructs.

* **Threads** – Threads are similar to the state machines in bare-metal embedded software development. Threads can control peripherals and threads can implement application code. The advantage of threads over state machines is that you can **create** and **destroy** threads dynamically and you can **prioritize** threads.
* **Semaphores** – Semaphores are a simple form of inter-thread communication. They allow a thread to notify another thread about an event it needs to react to.
* **Mutexes** – A mutex (from “mutual exclusion”) is a mechanism that allows a thread to monopolize the use of a shared peripheral, such as a serial port. For example, if a thread needs to print out a command prompt on the serial port, it does not want another thread to print characters on the same serial port in the middle of the command prompt. If a thread locks a peripheral with a mutex and another thread requests a mutex for that same peripheral, the second thread goes to sleep until the first thread releases the lock on the peripheral.
* **Queues** – A queue is a first-in, first-out (FIFO) buffer used to send data between threads. If a thread tries to read an empty queue, it will suspend until something is placed in the queue. It is possible to associate a timeout with a queue, so that if it remains empty for the period of the timeout, the thread will come out of suspension.
* **Timers** – A timer allows a function to be scheduled to run at a specified interval, such as taking a temperature measurement every minute.

The RTOS maintains a list of threads that are running, idle, or halted. It also prioritizes which thread will run next. The part of the RTOS that performs these functions is called the **scheduler**.

There are two main methods for managing the threads:

* **Preemptive** – The CPU controls which thread is running and can start and halt threads without the threads having to yield control. This requires a CPU with protected modes to run the scheduler, so that it can stop a thread by itself (preempting the executing of the thread). Linux, OSX, and Windows are preemptive OSes.
* **Non-preemptive or cooperative** – Each thread must be a “good citizen” and yield control back to the RTOS. This is similar to the state machines in bare-metal programing, in that they could not monopolize the CPU, but had to execute quickly and return (thus, yielding to the main loop, allowing other state machines to run). Delays, semaphores, mutexes, and queues all yield control back to the RTOS.

**Class Notes**