Real-time Systems

Chapter 2:
Real-time Operating System (RTOS)

Ngo Lam Trung

Dept. of Computer Engineering

Content

- Real-time operating system (RTOS)
- Components of RTOS
  - Task
  - Semaphore
  - Scheduler

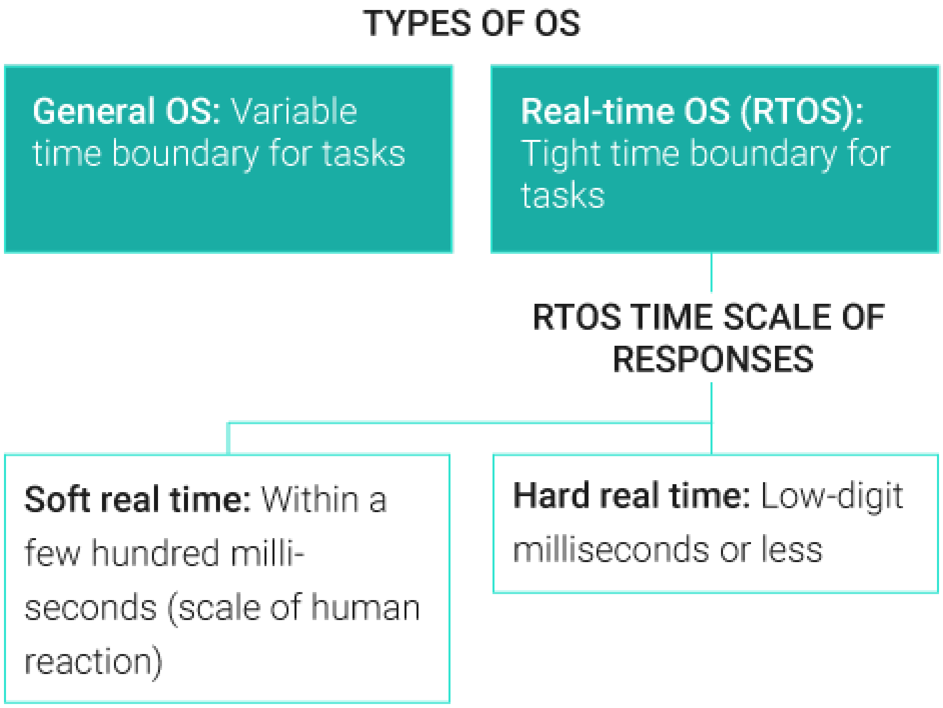
Brief history

- 60’s-70’s: UNIX for multi-user computer systems
- 80’s: Windows for personal computing environments
- Later in 80’s ~:
  - Embedded system changes human’s life, dominates CPU market
    - ➔ demand for RTOS
- UNIX, Windows: General purpose OS (GPOS)
- FreeRTOS, VxWork, QNX: RTOS

RTOSes are usually not known to consumers! Consumers can use electronic devices without knowing what’s inside. And that’s one big point of embedded system.

Real-time operating system

- The OS for real-time systems
- Two key features:
  - Predictability: how long a task will take to execute/finish
  - Determinism: the same input always result in the same output

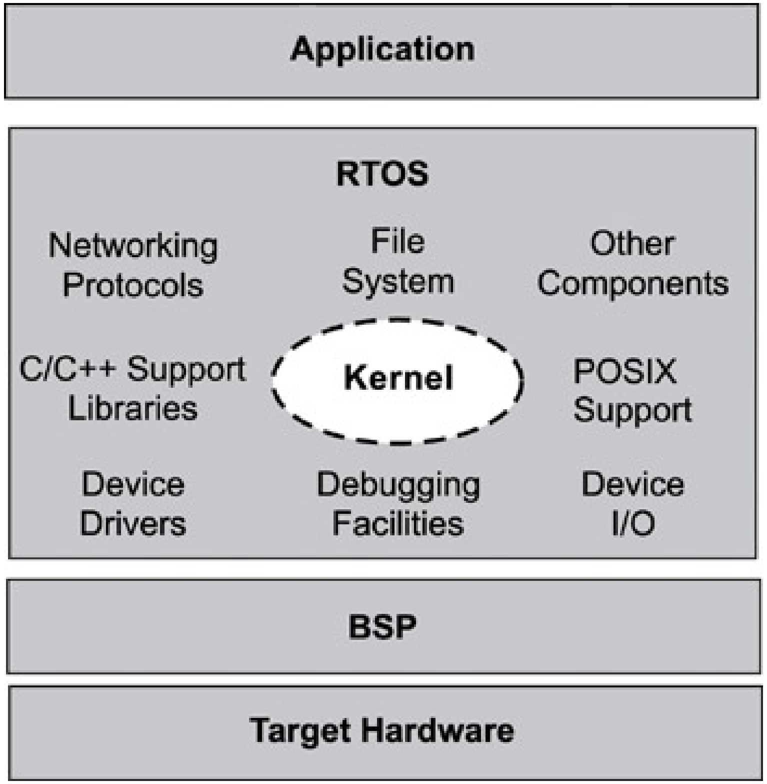


Comparing RTOS and GPOS

- Similarity between RTOS & GPOS
  - Multitasking
  - Software and Hardware resource management
  - Provision of underlying OS services to applications
  - Abstracting the hardware from the software applications
- Difference between RTOS and GPOS
  - Reliability and predictability
  - High performance (in real-time context)
  - Memory footprint
  - Priority-based scheduling

RTOS in a real-time system

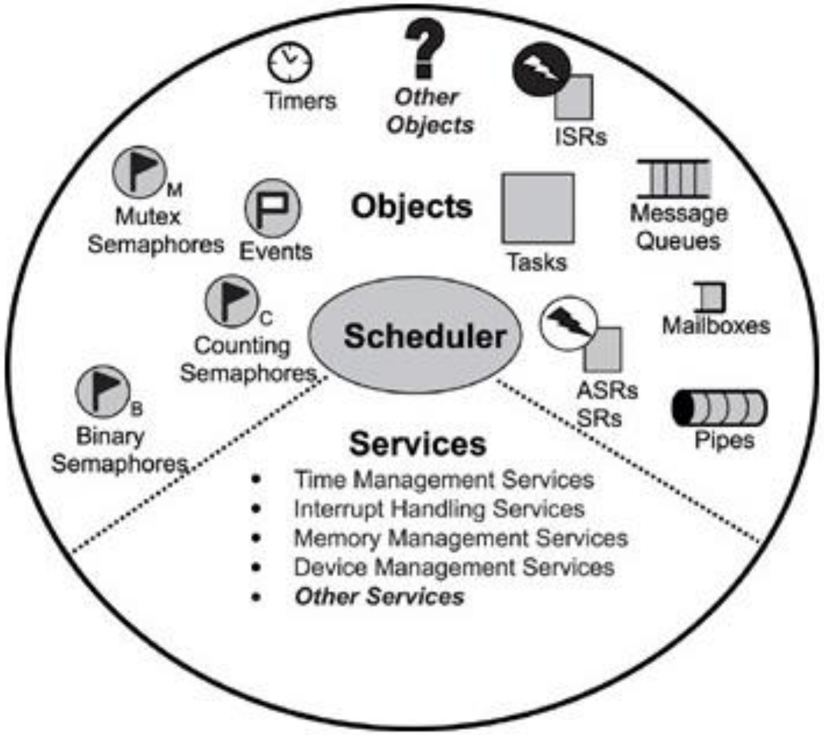
- Hardware and software are usually tightly-coupled



RTOS components

RTOS main tasks

- Scheduling
- Dispatching
- Inter-process communication



RTOS components

Components of RTOS

- RTOS kernel components:
  - Scheduler: provides a set of scheduling algorithms to determine which and when task executes.
    - Two most used approaches: preemptive, round robin
  - Objects: special kernel constructs to help developers create embedded applications
    - Ex: tasks, semaphores, message queues
  - Services: kernel operations on objects
    - Ex: timing, interrupt handling, & resource management

The scheduler

- A scheduler provides the algorithms needed to determine which task executes.
- Issues on schedulers:
  - Scheduling entities
  - Multitasking
  - Context switching
  - Dispatcher
  - Scheduling algorithms

Schedulable entities

- Schedulable entities
  - Kernel objects that can compete for execution time based on a predefined scheduling algorithm
- Task: a schedulable entity
  - Independent thread of execution that contains a sequence of independently schedulable instructions
  - This course focuses on tasks
- Task vs thread?

Multi-tasking

- Multitasking
  - Kernel's ability to handle multiple activities within set deadlines
- Multitasking scenario:
  - The kernel interleaves executions of multiple tasks sequentially based on a preset scheduling algorithms.
  - The kernel should ensure that all tasks meets their deadlines.
  - The tasks follows the kernel's scheduling algorithms, while interrupt service routines are triggered to run because of HW interrupts and their established priorities.
  - Higher CPU performance is required as the number of tasks increases.
    - More frequent context switching

Context switching(1)

- Each task has its own context
  - eg: states of CPU registers
- A context switch occurs when a scheduler switches from one task to another.
- Frequent context switching causes unnecessary performance overhead.

The dispatcher(1)

- The dispatcher:
  - A part of scheduler that performs context switching and changes the flow of execution.
  - At any time an RTOS is running, the flow of execution(flow of control), is passing through one of three areas:
    - through an application task,
    - through an ISR,
    - through the kernel

→ Scheduler decides when and how to execute context switching

Scheduler vs dispatcher

- Ex: which thread will the code below be scheduled on?
- Will context switching happen?

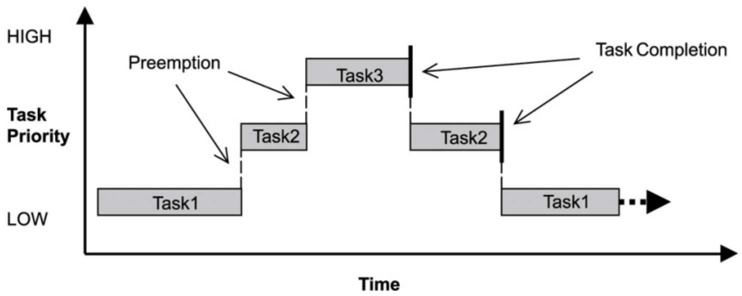
```
int threadProc(int param){
    int ParsingFinished = 0
    // do something
    ParsingFinished++;

    Deployment.Current.Dispatcher.BeginInvoke(() =>
    {
        if (ParsingFinished >= ParsingTotal)
        {
            tbMsg.Text = "Parsing done!";
        }
    });
}
```

Scheduler vs dispatcher

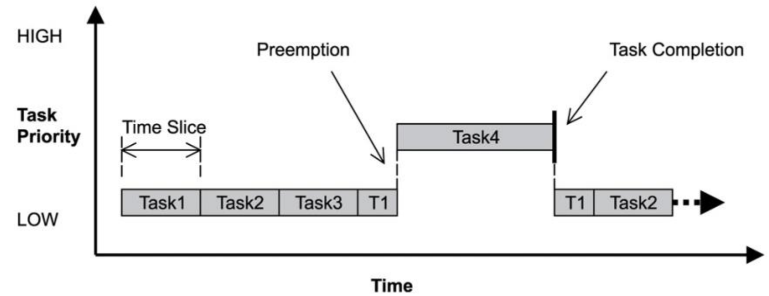
- Dispatcher
  - Low level mechanism
  - Responsibility: context switching
- Scheduler
  - High level policy
  - Responsibility: deciding which task to run

Scheduling algorithms

- Scheduling algorithms
  - Preemptive priority-based scheduling
  - Round-robin scheduling



Preemptive priority-based:
The task that gets to run at any point is the task with the highest priority



Priority-based round robin:
Combine advantages of priority based and round robin algorithm

Other algorithm will be the main focus in the later parts of this course.

Objects

- Used to develop concurrent embedded application.
- Tasks
  - concurrent & independent threads of execution that can compete for CPU execution time
- Semaphores
  - Token-like objects for synchronization or mutual exclusion
  - Incremented or decremented by tasks
- Message queues
  - Buffer-like data structures for synchronization, mutual exclusion, & data exchange between tasks

Services

- Services provided by RTOS kernels, eg:
  - timer management
  - interrupt handling
  - device I/O
  - memory management
- Some are programming language integrated (eg: alloc, malloc, new…)
- Others are accessible via API

➔ refer to OS's document

RTOS key characteristic

- Reliability
- Predictability
- Performance
- Compactness
- Scalability

Reliability

- Embedded system might need to operate for long periods without human intervention. Eg: network devices
- Different degrees of reliability may be required.
  - A digital solar-powered calculator reset: acceptable.
  - A telecom switch reset: unacceptable, costly to recover.
  - The OSes in these applications require different degrees of reliability.

Table 4.1: Categorizing highly available systems by allowable downtime.[1]

Number of 9s	Downtime per year	Typical application
3 Nines (99.9%)	~9 hours	Desktop
4 Nines (99.99%)	~1 hour	Enterprise Server
5 Nines (99.999%)	~5 minutes	Carrier-Class Server
6 Nines (99.9999%)	~31 seconds	Carrier Switch Equipment
1 Source: 'Providing Open Architecture High Availability Solutions,' Revision 1.0, Published by HA Forum, February 2001.		

Predictability

- Meeting time requirements is key for proper operation.
- The RTOS needs to be predictable (at least to a certain degree).
- Deterministic operation: RTOSes with predictable behavior, in which the completion of operating system calls occurs within known timeframes.

Performance

- Fast enough to fulfill timing requirements
- The more deadlines to be met the faster the system's CPU must be
- Both hardware and software contribute to system performance

Compactness

- Constraints of embedded systems:
  - Application design constraints
  - Cost constraints.
- Design requirements
  - Limit system memory
  - Limits the size of the application
- Cost constraints:
  - Limit power consumption
  - Avoid expensive hardware
- ➔ RTOS must be small and efficient!

Scalability

- RTOSes can be used in a wide variety of embedded systems
  - Scale up or down to meet application-specific requirements
  - Scale up: Adding additional hardware and its corresponding software modules, eg. disk drive, sensor…
  - Scale down: removing unnecessary hardware drivers and software modules.

Popular RTOS: FreeRTOS

- Open source RTOS developed by Richard Barry, very popular for small embedded systems.
- Stewardship transferred to Amazon in 2017.

Tiny, power-saving kernel

Scalable size, with usable program memory footprint as low as 9KB. Some architectures include a tick-less power saving mode

Support for 40+ architectures

One code base for 40+ MCU architectures and 15+ toolchains, including the latest RISC-V and ARMv8-M (Arm Cortex-M33) microcontrollers

Modular libraries

A growing number of add-on libraries used across all industry sectors, including secure local or cloud connectivity

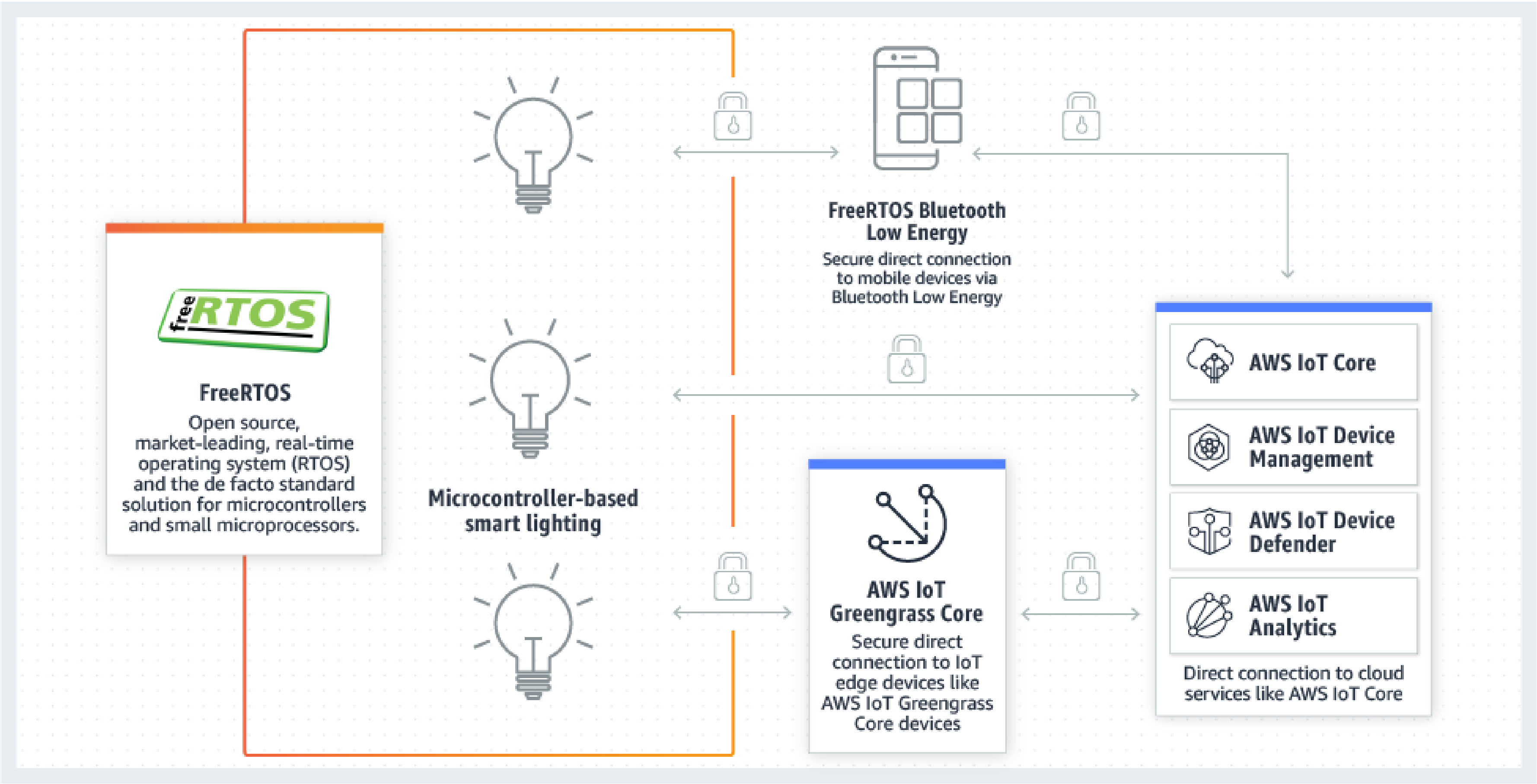
AWS Reference Integrations

Take advantage of tested examples that include all the libraries essential to securely connect to the cloud

MIT licensed, with options

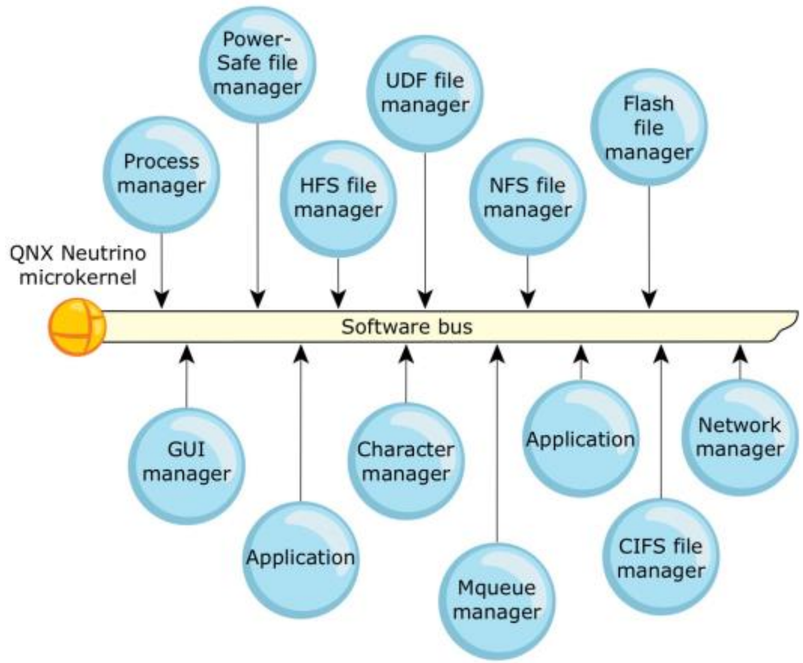
FreeRTOS can be used for any purpose under its MIT license. Our strategic partner also provides commercial licenses, and safety certification.

Amazon AWS FreeRTOS



QNX

- RTOS developed by QNX Software Systems in 1980s, acquired by BlackBerry in 2010.
- Has large market share in automotive industry: telematics, infotainment, navigation, and later ADAS
- Micro-kernel architecture

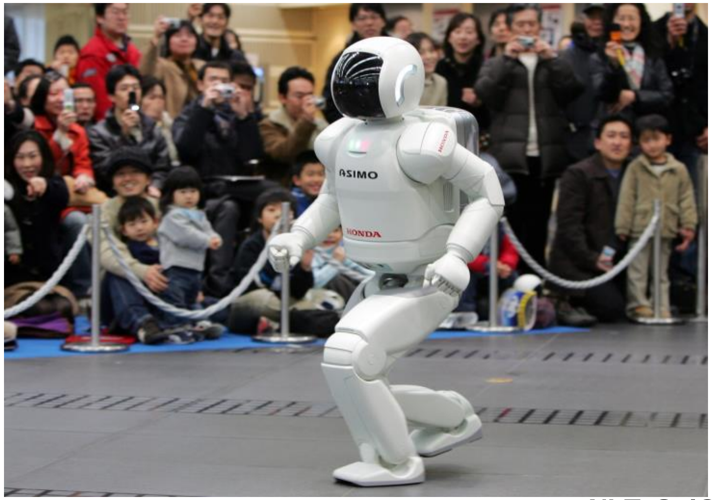


QNX

- QNX SDKs
  - QNX Software Development Platform 6.5.x
  - QNX SDK for Bluetooth® Connectivity
  - QNX SDK for Apps and Media 1.1
  - QNX Acoustics Middleware
  - QNX CAR Platform for Infotainment 2.1
  - QNX Platform for ADAS 1.0
  - QNX Wireless Framework 1.0
  - QNX OS for Automotive Safety 1.0
  - QNX Hypervisor 1.0
  - QNX OS for Medical 1.1
  - QNX OS for Safety
- Education license available: QNX Download Center

VxWork

- RTOS by Wind River since 1987
- Very famous, widely used in aerospace, automotive, industrial, medical, networking infrastructure…
  - Spacecraft, space telescopes, and rovers…
  - Aircraft and defense systems…
  - Medical robots, CT, MRI scanners…
  - Industrial robots…
  - Transportation safety



VxWork

- System components