COEN-4720 Embedded Systems Design
Lecture 9
Real Time Operating Systems (RTOS)
Part 1: Processes/Tasks and Threads

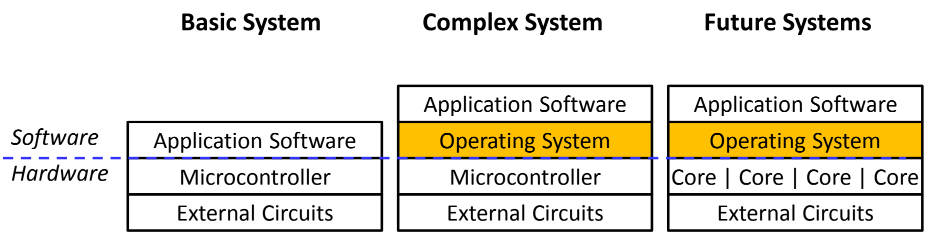
Cristinel Ababei
Dept. of Electrical and Computer Engineering
Marquette University

Overview

- What is an Operating System?
- Processes or Tasks, Scheduling
- Threads
- OS, RTOS
- The Kernel
- Cortex-M3

What is an Operating System?

- A software layer between the application software and the hardware



What is an Operating System?

- Typical embedded system (ES) solves a problem by decomposing it into smaller pieces called **tasks** that work together in an organized way
- System is called **multitasking** system and design aspects include:
  - Exchanging/sharing data between tasks
  - Synchronizing tasks
  - Scheduling tasks
- The piece of software that provides the required coordination is called an **operating system (OS)**
- When the control must ensure that task execution satisfies a set of specified time constraints, the OS is called a **real-time operating system (RTOS)**

Process or Task

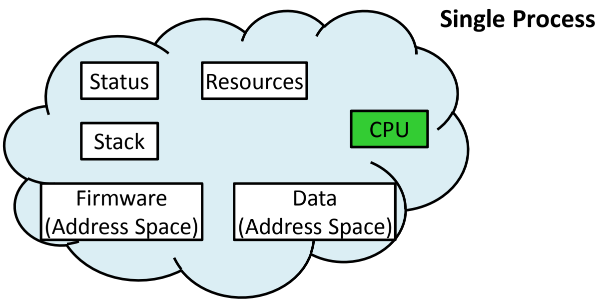
- Embedded program (a static entity) = a collection of firmware modules
- When a firmware module is executing, it is called a **process** or **task**
- A task is usually implemented in C by writing a function
- A task or process simply identifies a job that is to be done within an embedded application
- When a process is created, it is allocated a number of resources by the OS, which may include:
  - Process stack
  - Memory address space
  - Registers (through the CPU)
  - A program counter (PC)
  - I/O ports, network connections, file descriptors, etc.
- These resources are generally not shared with other processes

Types of Tasks

- Periodic tasks
  - Found in hard real-time applications
  - Examples: control, every 10 ms; multimedia, every 22.727us;
- Intermittent tasks
  - Found in all types of applications
  - Examples: send email every night at 4am; calibrate a sensor on startup; save all data when power goes down;
- Background tasks
  - A soft real-time or non real-time task
  - Will be accomplished only if CPU time is available
- Complex tasks
  - Found in all types of applications
  - Examples: Microsoft Word; Apache web server;

Single Process

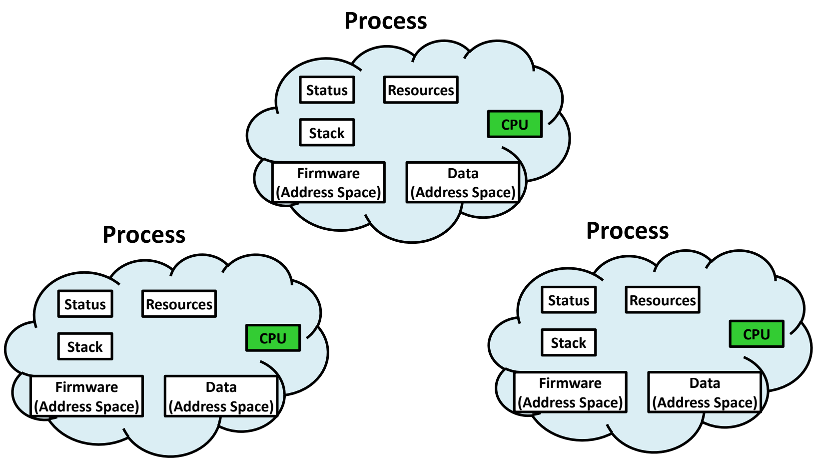
- Traditional view of computing: focuses on **program**. One says that the program (or task within the program) runs on the computer
- In embedded applications, we change the p.o.v. to that of microprocessor: CPU is used to execute the firmware. CPU is just another resource
- The time it takes a task to complete is called **execution time**



Multiple Processes

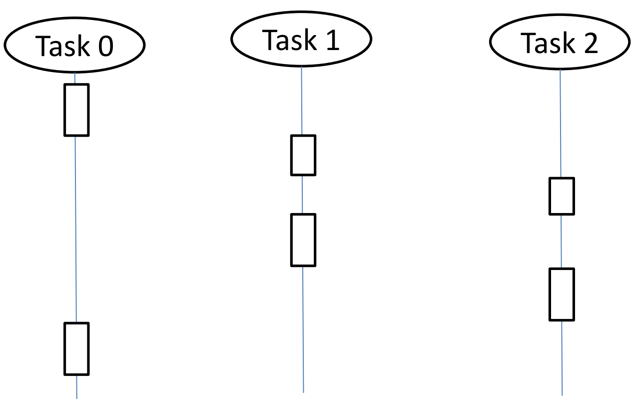
- If another task is added to the system, potential resource contention problems arise
- This is resolved by carefully managing how the resources are allocated to each task and by controlling how long each can retain the resources
- The main resource, CPU, is given to tasks in a time multiplexed fashion (i.e., time sharing); when done fast enough, it will appear as if both tasks are using it at the same time
- The execution time of the program will be extended, but operation will give the *appearance* of simultaneous execution. Such a scheme is called **multitasking**

Multiple Processes



Sequence Diagram

- At any instant in time, only one process is actively executing; it said to be in **run** state
- The other processes are in **ready waiting** state



Task Scheduling

- A **schedule** is set up to specify when, under what conditions, and for how long each task will be given the use of the CPU (and other resources)
- The criteria for deciding which task is to run next are collectively called a **scheduling strategy**, which generally falls into three categories:
  - Multiprogramming
    - each task continues until it performs an operation that requires waiting for an external event
  - Real-Time
    - tasks with specified temporal deadlines are guaranteed to complete before those deadlines expire
  - Time sharing
    - running task is required to give up the CPU so that another task may get a turn

Task States

- Primarily 4 states
  1. Running or Executing
  2. Ready to Run (but not running)
  3. Waiting (for something other than the CPU)
  4. Inactive
- Transition between states is referred to as **context switch**
- Only one task can be Running at a time, unless we use a multicore CPU
- Task waiting for CPU is Ready to Run
- When a task has requested I/O or put itself to sleep, it is Waiting
- An Inactive task is waiting to be allowed into the schedule

Address Space of a Process

- When a process is created by the OS, it is given a portion of the physical memory in which to work
- The set of addresses delimiting that code and the data memory, proprietary to each process, is called its address space
- Processes are segregated
  - Supervisor mode
  - User mode – limited to a subset of instructions
- A process may create or spawn **child processes** (each with its own **data** address space, data, status, and stack)
- A process may create multiple threads (each with its own stack and status information)

Overview

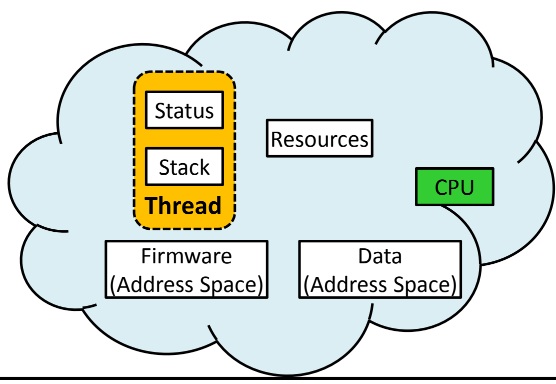
- What is an Operating System?
- Processes or Tasks, Scheduling
- Threads
- OS, RTOS
- The Kernel
- Cortex-M3

Threads

- A process or task is characterized by a collection of resources that are utilized to execute a program
- The smallest subset of these resources (a copy of the CPU registers including the PC and a stack) that is necessary for the execution of the program is called a **thread**
- A thread is a unit of computation with code and context, but no private data
- A thread can be in only one process; a process without a thread can do nothing!

Single-process single-thread

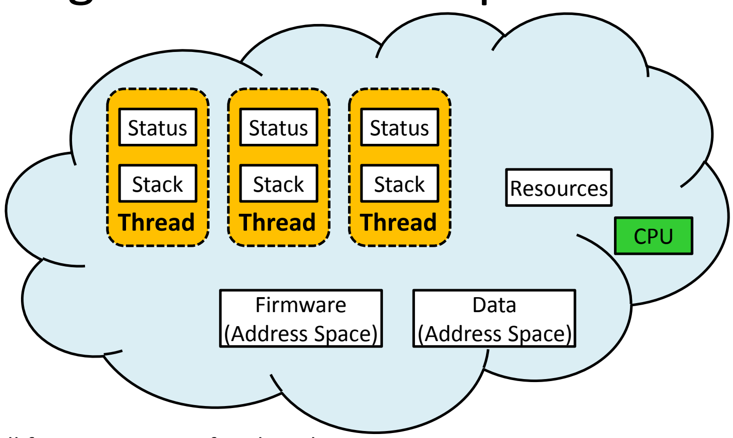
- The sequential execution of a set of instructions through a task or process in an embedded application is called a **thread of execution** or **thread of control**
- This model is referred as single-process single-thread



Multiple Threads

- During partitioning and functional decomposition of the function intended to be performed by an ES → identify which actions would benefit from parallel execution
  - For example, allocate a subjob for each type of I/O
- Each of the subjobs has its own thread of execution
  - Such a system is called a **single-process multithread** design
- Threads are not independent of each other (unlike processes or tasks)
  - Threads can access any address within the process, including other threads' stacks
- An OS that supports tasks with multiple threads is called a **multithreaded operating system**

Single-Process Multiple-Threads



- All four categories of multitasking operating system:
  - Single process single thread
  - Multiprocess single thread
  - Single process multiple threads
  - Multiprocess multiple threads

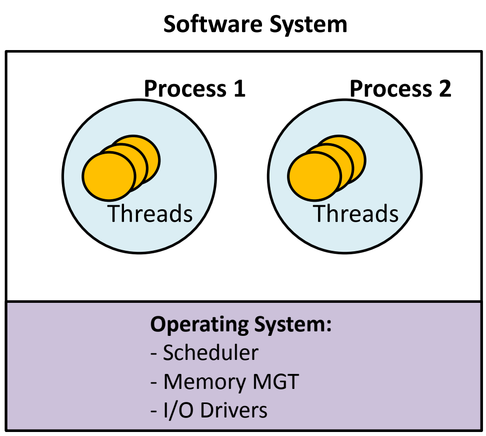
Processes (tasks) vs. Threads

- At the minimum, a process or task needs the following:
  1. The code or firmware, the instructions
     - These are in the memory and have addresses
  2. The data that the code is manipulating
     - The data starts in the memory and may be moved to registers. The data has addresses
  3. CPU and associated physical registers
  4. A stack
  5. Status information

Shared among member Threads

Proprietary to each Thread

Example: complete software system with two processes



Reentrant Code

- Child processes (and their threads) share the same firmware memory area → two different threads can execute the same function
- Functions using only local variables are inherently **reentrant**
- Functions using global variables, variables local to the process, variables passed by reference, or shared resources are not reentrant
- Any shared functions must be designed to be reentrant

Overview

- What is an Operating System?
- Processes or Tasks, Scheduling
- Threads
- OS, RTOS
- The Kernel
- Cortex-M3

The Operating System (OS)

- Embedded **Operating System** provides an environment within which firmware pieces, the tasks that make up the embedded application, are executed
- An OS provides or supports three functions:
  1. **Schedule** task execution
  2. **Dispatch** a task to run
  3. Ensure **communication and synchronization** among tasks

The Kernel

- Scheduler
  – Determines which task will run and when it will do so
- Dispatcher
  – Performs the necessary operations to start the task
- Intertask or interprocess communication
  – Mechanism for exchanging data and information between tasks and processes on the same machines or different ones
- The **Kernel** is the smallest portion of the OS that provides these functions

Services

- The above functions are captured in the following types of services:
  - Process or task management
    - Creation and deletion of user and system processes
  - Memory management
    - Includes tracking and control of which tasks are loaded into memory, monitoring memory, administer dynamic mem
  - I/O System management
    - Interaction with devices done through a special piece of software called a **device driver**
    - The internal side of that software is called a **common calling interface** (an **application programmer's interface, API**)
  - File system management
  - System protection
  - Networking
  - Command interpretation

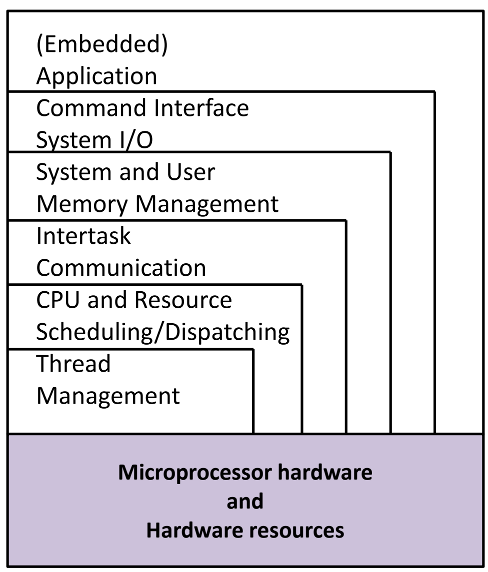
The Real-Time Operating System (RTOS)

- Primarily an operating system, which in addition ensures that (rigid) time constraints can be met
- Commonly found in embedded applications
- Key characteristic of an RTOS is that it has **deterministic behavior** = given the same state and the same state of inputs, the next state (and associated outputs) will be the same each time the control algorithm utilized by the system is executed

Hard vs. Soft Real Time

- Real time
  - A software system with specific speed or response time requirements
- Soft real time
  - Critical tasks have priority over other tasks and retain that priority until complete
  - If performance is not met, performance is considered low
- Hard real time
  - System delays are known or at least bound
  - If deadlines are not met, the system has failed
- Super hard real time
  - Mostly periodic tasks: OS system tick, task compute times, and deadlines are very short

Architecture of Operating System



Architecture of Operating System

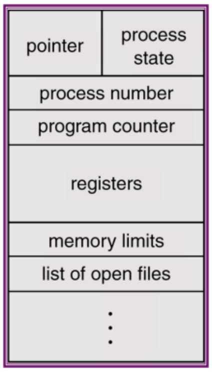
- Organized like the onion model
  - The hierarchy is designed such that each layer uses functions/operations and services of lower layers → increased modularity
- In some architectures, upper layers have access to lower layers through system calls and hardware instructions

Process or Task Control Block (PCB or TCB)

- An RTOS "orchestrates" the behavior of an application by executing each of the tasks that comprise the design according to a specified schedule
- Each task or process is represented by a **task or process control block (TCB)**
- A TCB is a **data structure** in the operating system kernel containing the information needed to manage a particular process
- The TCB is "the manifestation of a process in an operating system"

Task Control Block (TCB)

- PCB allocation
  - Static: used typically in ES's
  - Dynamic
- A fixed number of TCBs is allocated at system generation time and placed in dormant (unused) state
- When a task is initiated, a TCB is created and the appropriate information is entered
- TCB is placed in Ready state by scheduler
- TCB will be moved to Execute state by dispatcher
- When task terminates, associated TCB is returned to a dormant state
- With fixed number of TCBs, no runtime memory management is necessary



Queue or Job Queue

- When a task enters the system, it will be placed into a queue called the Entry Queue or Job Queue
- May be implemented as a linked list or as an array

A Simple Kernel

- For the description of several versions of a primitive operating system kernel, read Ch. 11 of:

[Book] James K. Peckol, Embedded Systems, A Contemporary Design Tool, John Wiley & Sons, Inc., 2007.

Overview

- What is an Operating System?
- Processes or Tasks, Scheduling
- Threads
- OS, RTOS
- The Kernel
- Cortex-M3

Examples of Embedded Operating Systems Supporting Cortex-M3

Company	Product[2]
FreeRTOS (www.freertos.org)	FreeRTOS
Express Logic (www.expresslogic.com)	ThreadX(TM) RTOS
Micrium (www.micrium.com)	µC/OS-II
Accelerated Technology (www. Acceleratedtechnology.com)	Nucleus
Pumpkin Inc. (www.pumpkininc.com)	Salvo RTOS
CMX Systems (www.cmx.com)	CMX-RTX
Keil (www.keil.com)	ARTX-ARM
Segger (www.segger.com)	embOS
IAR Systems (www.iar.com)	IAR PowerPac for ARM

RL-RTX with uVision

- **See lab#9 of this course!**
- RL-RTX is one of the components of RL-ARM, the RealView Real-Time Library (RL-ARM)
- The RTX kernel is a real time operating system (RTOS) that enables one to create applications that simultaneously perform multiple functions or tasks (statically created processes)
- The RTX kernel uses the execution priorities to select the next task to run (preemptive scheduling). It provides additional functions or services for inter-task communication, memory management, and peripheral management.
- RTX programs are written using standard C constructs and compiled with the RealView Compiler
- The **`RTL.h`** header file defines the RTX functions and macros that allow you to easily declare tasks and access all RTOS features.

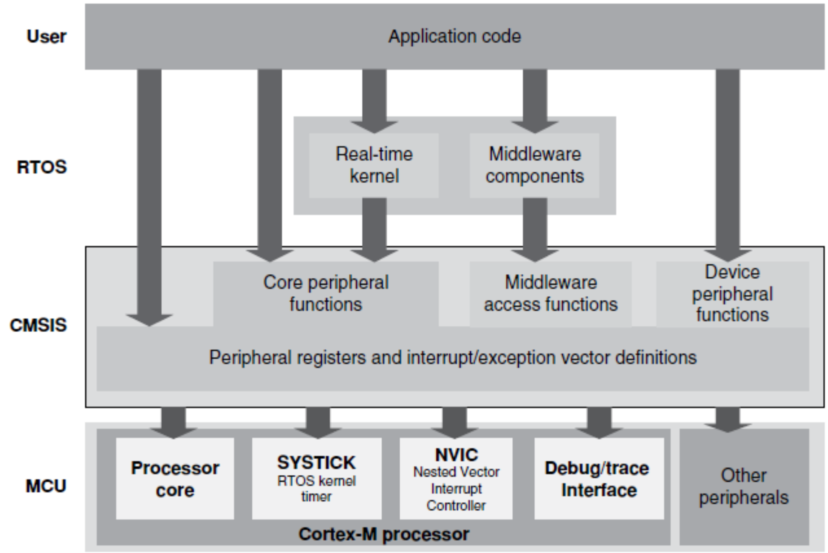
CMSIS – Cortex Microcontroller Software Interface Standard

- CMSIS is a vendor-independent hardware abstraction layer (HAL) for the Cortex-M processor series
- CMSIS enables consistent and simple software interfaces to the processor and the peripherals, simplifying software re-use, reducing the learning curve for new microcontroller developers and reducing the time to market for new devices
- Standardizing the software interfaces across all Cortex-M silicon vendor products → significant cost reductions in software development

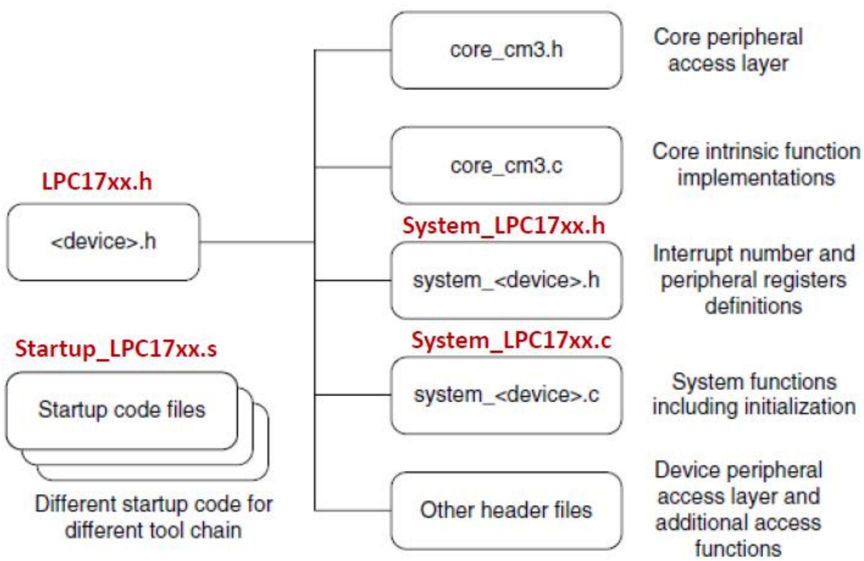
CMSIS

- Hardware Abstraction Layer (HAL) for Cortex-M processor registers
  - NVIC, MPU
- Standardized system exception names. For example:
  - void SVC_Handler()
  - void UART0_IRQHandler()
- Standardized method of header file organization
- Common method for system initialization
  - SystemInit()
- Standardized intrinsic functions. For example:
  - void __disable_irq(void)
  - void __enable_irq(void)
- Common access functions for communication
- Standardized way for embedded software to determine system clock frequency
  - SystemFrequency variable is defined in device driver code

CMSIS Architecture



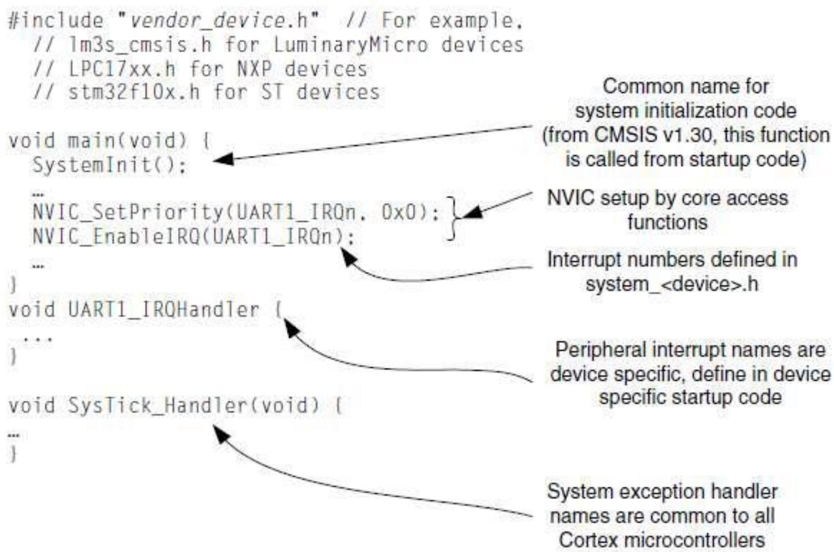
CMSIS Files



External Interrupt Programming

- IRQn is defined in <device.h> file (i.e., LPC17xx.h)
- A set of functions
  - `void NVIC_EnableIRQ(IRQn_Type IRQn)`
  - `void NVIC_DisableIRQ(IRQn_Type IRQn)`
  - `void NVIC_SetPriority(IRQn_Type IRQn, int32_t priority)`
  - `uint32_t NVIC_GetPriority(IRQn_Type IRQn)`
  - `void NVIC_SetPendingIRQ(IRQn_Type IRQn)`
  - `void NVIC_ClearPendingIRQ(IRQn_Type IRQn)`
  - `IRQn_Type NVIC_GetPendingIRQ(IRQn_Type IRQn)`

Example



Credits, References

- Chapters 11,12 of: James K. Peckol, Embedded Systems, A Contemporary Design Tool, John Wiley & Sons, Inc., 2007.
- Chapters 3,4 of: Jonathan W. Valvano, Real-Time Operating Systems for ARM Cortex-M Microcontrollers, 2012.
- Lecture Notes at UWash (Chapter 5); http://abstract.cs.washington.edu/~shwetak/classes/ee472/