MASTER OF SCIENCE
IN ENGINEERING

TSM_AdvEmbSof

Tasks and concurrency

Serge Ayer | 03.11.2023 | Cours MSE

swissuniversities

Hes·so Master

Organize an Embedded Software into multiple tasks

- As we have seen in the previous lecture, it's not usually possible to program every embedded software into a single control loop.
- The code needs to be broken up into smaller elements such that
  - code is readable, structured and documented
  - code can be tested in a modular form
  - development reuses existing code utilities to keep development time short
  - code design supports multiple engineers working on a single project
  - future upgrades to code can be implemented efficiently
- Organize these code elements into classes and methods

Organize an Embedded Software into multiple tasks

- In almost all embedded programs, the program has to undertake a number of different activities. We call these distinct activities tasks.
  - Once a program has more than one task, we enter the domain of multi-tasking.
- Tasks performed by embedded systems tend to fall into two categories:
  - event-triggered: occur when a particular external event happens, at a time which is not predictable
  - time-triggered: happen periodically, at a time determined by the microcontroller.

Our Bike computer example

Task	Event or time-triggered
Reset with push button	Event
Pedal rotation (speed and distance)	Event or time-triggered
Change gear	Event
Get temperature	Time-triggered
Update display	Event or time-triggered

Multitasking and OS

- Use an OS for writing multitasking applications
- What is an Operating System doing?
  - Provides an intermediary interface between applications and computer hardware
  - Facilitates application development (convenience and efficiency)
- Various OSs are available in the market for various hardware platforms, but they have the same missions and must perform the same tasks:

Managing the processor	Managing memory	Managing devices
Managing file systems	System security	Managing network communication
Fault tolerance and error detection	Multitasking and job accounting	Task coordination

Operating System Services

Basic operating system services include:

Scheduling tasks	Allocating memory resources	Managing IOs
Managing file systems	Communicating and networking	Detecting and fixing errors
	Protecting and securing information	

Threads vs Processes

- A process allows to isolate different tasks running on the same platform
  - Process resources are private to the process
    - For instance, the memory attached to a process is private and cannot be easily accessed from other processes.
  - Inter-process communication is a costly process
  - Context switching between processes is more intensive and costly than between threads
- A process can usually run several threads
  - The threads share the process resources
  - The threads own their own context (stack, registers, etc.)
  - Sharing resources among threads is easier and less costly

Multitasking vs Multiprocessing

- In a uniprocessor system, only one process executes at a time.
- Multitasking:
  - Multiple tasks run concurrently on a uniprocessor with interleaved or time shared execution, or simultaneously on multiple processor systems.
  - The concurrent execution of multiple tasks must manage resource sharing:
    - For example, utilization of shared memory by multiple processes correctly without overwriting the values and writing in the correct sequence.
- Multiprocessing:
  - Use of two or more CPUs (processors) or cores within a CPU.
  - Multiple processes can be executed at a time.
  - These processors share the computer bus, sometimes the clock, memory and peripheral devices also.

Tasks and RTOS

- RTOS provides a approach to program development where control of the CPU and all system resources are handed to the operating system (OS).
- It is the OS which determines which section of the program is to run, for how long, and how it accesses system resources.
  - The OS also provides communication and synchronization between tasks.
  - It controls the use of resources shared between the tasks, for example memory and hardware peripherals.
- A program written for an RTOS is structured into tasks, where each task:
  - Is mostly executed in a separate process or thread (though it is not mandatory).
  - Written as a self-contained program module.
  - Can be prioritized

Mbed OS/RTX Threads

- On RTOS, tasks are often associated with threads
  - No process concept
  - Multi-tasking = multiple threads
- Mbed OS provides a Thread API
  - Based on Keil RTX5 RTOS kernel, through the CMSIS-RTOS API
- Full description of the API on Thread API
  - A thread is an instance of the Thread class. It must be created and then started.
  - Threads can be created with different priorities.
  - Important: at system initialization, a special thread function executing the main() function is created

Mbed/RTX Threads

Unlimited number of tasks each with 254 priority levels

Support for multithreading and thread-safe operation

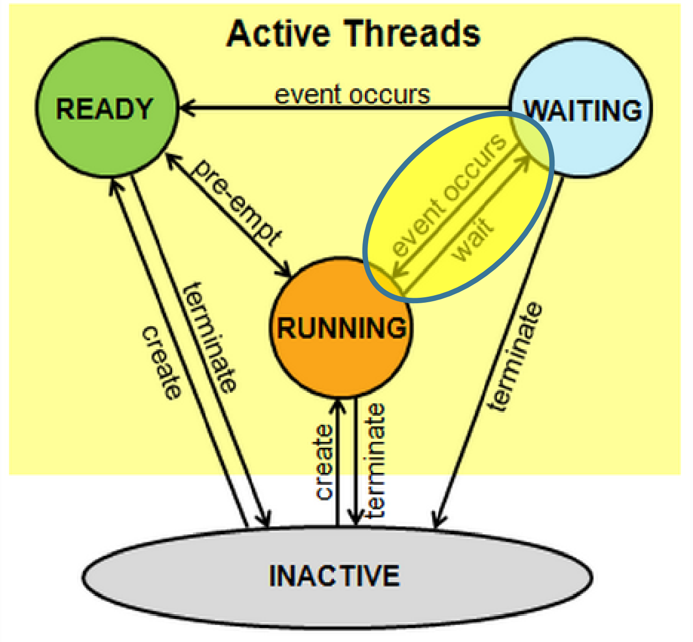
Inter-task communication manages the sharing of data, memory, and hardware resources among multiple tasks

Unlimited number of mailboxes, semaphores, mutex, and timers (hardware-permitting)

Defined stack usage - each task is allocated a defined stack space, enabling predictable memory usage

Mbed OS/RTX Thread States

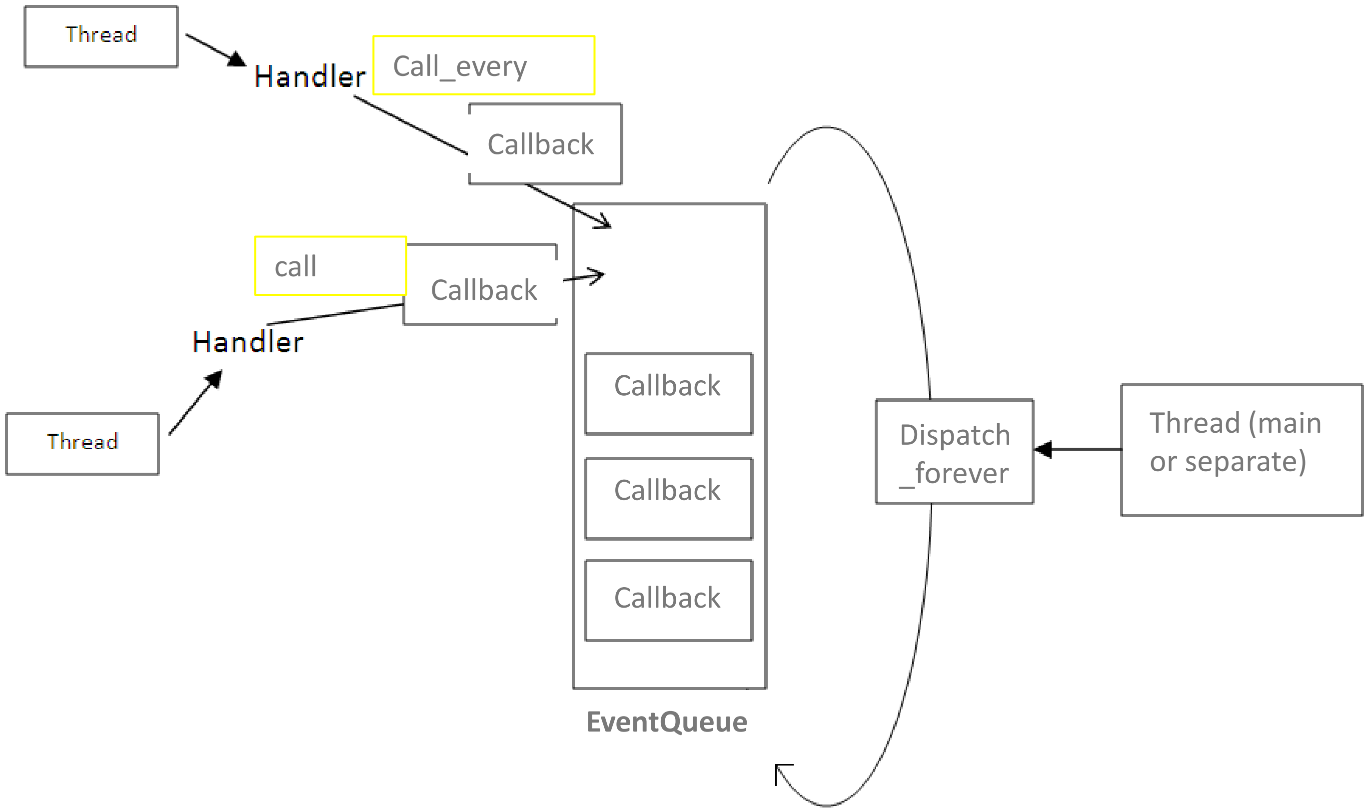
- **Running**:
  - Currently running.
  - Only one thread at a time can be in this state.
- **Ready**:
  - Ready to run are in the Ready state.
  - Once the Running thread has terminated or is Waiting, the next Ready thread with the highest priority becomes the Running thread.
- **Waiting/Blocked:**
  - Waiting for an event to occur.
- **Inactive/Terminated**:
  - Not created or terminated.
  - These threads typically consume no system resources.



Mbed OS EventQueue

- Simple and powerful mechanism for running an event loop
- Periodic tasks can be posted to the queue
- The queue can be used for postponing the execution of a code sequence from an interrupt handler to a user context
- Events must be dispatched by a thread
- Documentation is available here

EventQueue principe



Thread synchronization

- In a multitasking system, the different tasks may compete for shared resources or may wait for different events to happen.
- In some cases, a given task may thus enter a Waiting or Blocked state.
- There are in fact multiple Waiting states:
  - WaitingEventFlag: Waiting for a event flag to be set.
  - WaitingMutex: waiting for a mutex event to occur.
  - WaitingSemaphore: Waiting for a semaphore event to occur.
  - WaitingThreadFlag: Waiting for a thread flag to be set.
  - WaitingMemoryPool: Waiting for a memory pool.
  - WaitingMessageGet: Waiting for message to arrive.
  - WaitingMessagePut: Waiting for message to be sent.
  - WaitingDelay: Waiting for a delay to occur.

Events

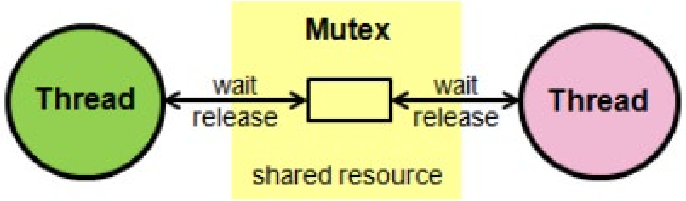
- Events are useful for waiting for specific conditions
  - One thread waits for a specific condition to be met
    - The condition can be made of one specific flag or a combination (AND/OR) of flags
    - Wait with timeout is also possible
  - Another thread notifies (sets) the specific condition
  - No busy waiting !
- In MbedOS, events are made available through the EventFlags API.
- Codelab: Using EventFlags for Waiting for an Event

Mutex

- Even on uniprocessor systems, there is a need for protection when sharing resources
- Mutex controls the access to shared resources
  - Enforces that only one thread of execution can have access to a section of code (called the critical section)
  - Needs to care about deadlock or starvation
    - Be careful when entering more than one mutex
    - Always release a mutex after use
- RTX/Mbed OS implements the priority inheritance scheme
  - No priority ceiling
- Codelabs:

  Shared Resources and Mutual Exclusion

  Deadlock

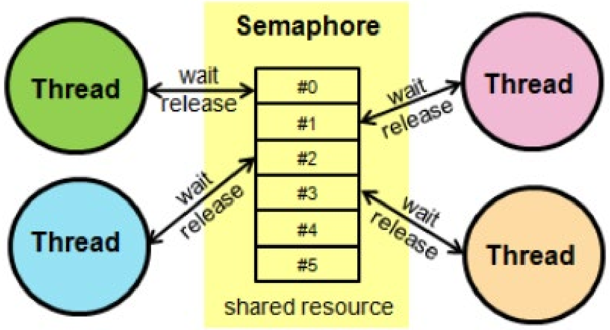


Dealing with Deadlock

- Ostrich algorithm (very famous!)
- Deadlock prevention: if any of the Coffman conditions are false
  - Mutex is inevitable
  - Request all resources at the beginning– either pick two forks at the same time or wait / all-or-none
  - Preemption is inevitable
  - Prevent circular wait condition: Resource hierarchy solution by Dijkstra (as the only practically avoidable condition)
- Deadlock avoidance
  - Evaluate the chance of deadlock while allocating a resource. Grant or deny based on this information
  - Banker's algorithm
- Deadlock detection: what to do with the existing deadlock?
  - Model checking
    - Kill all or part of the deadlocked processes?
  - Resource preemption?
  - Restart? Watchdog for embedded system?

Semaphore

- A semaphore manages thread access to a pool of shared resources of a certain type
  - Unlike a mutex, a semaphore can control access to several shared resources
  - For example, a semaphore enables access to and management of a group of identical peripherals
- Codelab: [Shared Data and Semaphore]



Queue / Mail

- A Queue allows you to queue integer/pointers to data from producer threads to consumer threads
- A Mail works like a queue, but in addition it provides a memory pool for allocating messages
- Codelab: [Shared Data and Queue]