Threads, mutexes and semaphores

Christian Auby

UiA

Agenda

- What is a real time operating system
- Mbed OS features
- Mbed OS architecture
- Creating threads
- Thread switching
- Protecting data with mutexes
- Sharing data between threads
- Common errors

What is a real time operating system

About

- An RTOS is a small OS for embedded systems
- Mbed OS is one such “Real Time operating System”
- An RTOS has features similar to a full desktop OS:
  - Process / thread / task switching
  - Mutexes / semaphores
  - IPC: inter process / thread / task communication
  - We’ll go through these later in this / the next lecture

Realtime vs traditional OS (Windows etc)

- Lower latency overall (thread switching, interrupt handling etc.)
- Lower OS overhead (no drivers, or lightweight drivers)
- Predictability (important for embedded where timing matters)
- Usually no Memory Management Unit support (bare metal)
- Using an RTOS is useful in larger projects
  - Helps organizing code
  - Separate parts of the program can run in different threads
  - Threads can communicate with each other if / when necessary

Mbed OS features

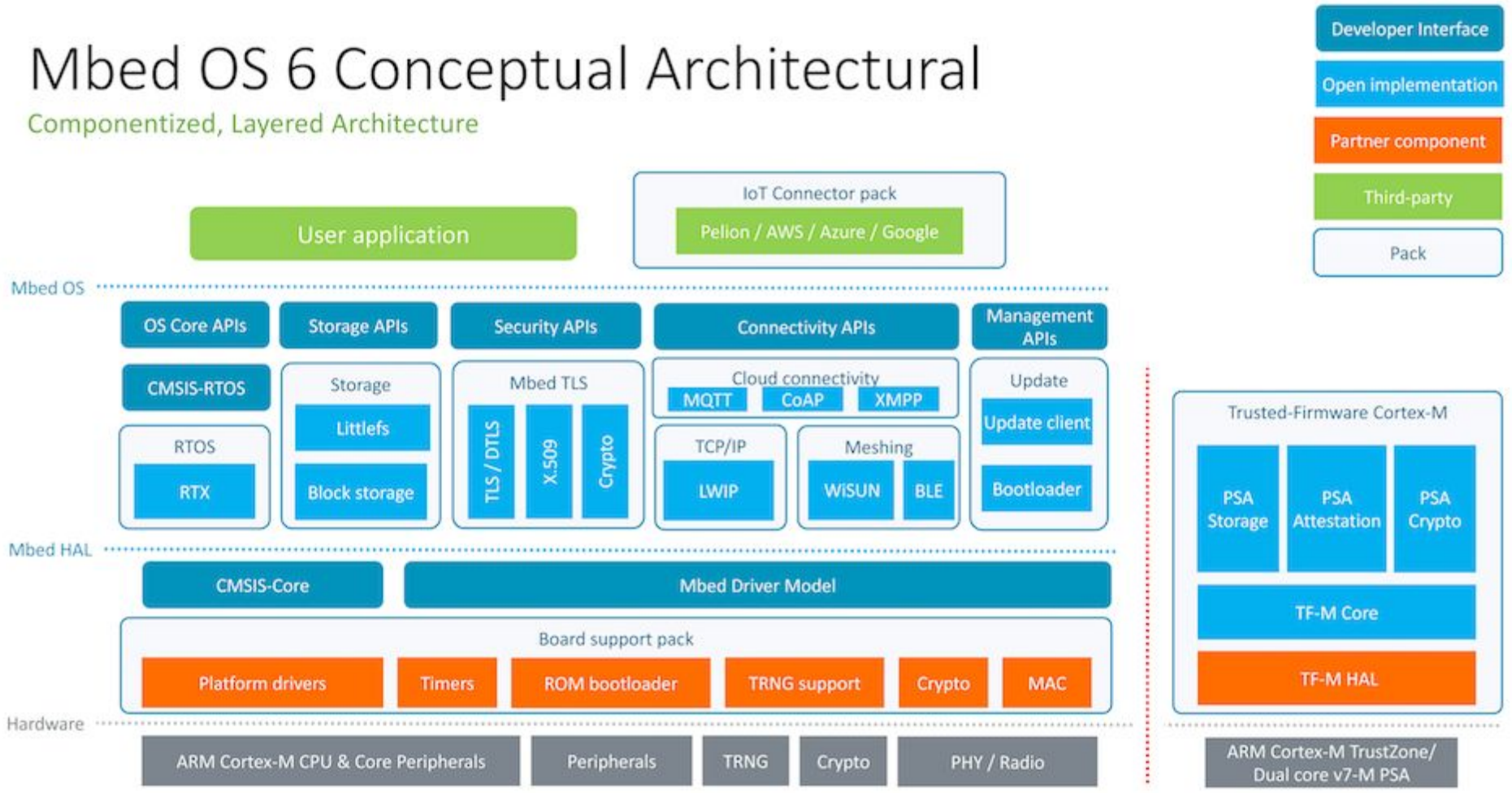
About Mbed OS

- Mbed OS has two main profiles - bare metal and OS
- The bare metal profiles is suitable for small programs on limited devices
- The OS profile is suitable for larger programs on more capable devices

Features

- OS features: Threads, mutexes, semaphores, mailboxes
- Communication features:
  a. Ethernet / Wi-Fi
  b. Bluetooth
  c. LoRaWAN
  d. Cellular
- Security features:
  a. Encryption
  b. NFC/RFID

Mbed OS architecture



UiA

Concept: Thread

About threads

- A thread is a piece of code that runs independent of other threads
- Threads are used to organize larger programs into smaller pieces
- They have their own stacks (for local variables)
- They share the same memory space
- They can use mutexes to protect shared resources
- They can use semaphores to synchronize

Using Threads

- Threads are declared using the class *Thread*
- Functions can then be started with *thread.start(function);*
- Mbed OS switches the active thread automatically to let all threads run
- Threads can sleep to reduce CPU usage:
  `ThisThread::sleep_for(5s);`
- You can wait for a thread to finish with the *join()* function:
  `thread.join();`
- Threads can have different priorities

Preemptive vs cooperative multitasking

Preemptive

- The scheduler pauses and unpauses threads at will
- Threads can be paused and unpaused at any time
- Threads are generally unaware that this happens
- Mbed OS uses preemptive threads

Cooperative

- Threads must pause themselves to let other threads run
- Threads pause by calling sleep, or other functions that wait for data
- The scheduler decides which thread is allowed to run

Example: Thread

```
#include "mbed.h"

DigitalOut led1(LED1);
DigitalOut led2(LED2);
Thread thread;

void led2_thread()
{
    while (true) {
        led2 = !led2;
        ThisThread::sleep_for(1000ms);
    }
}

int main()
{
    thread.start(led2_thread);

    while (true) {
        led1 = !led1;
        ThisThread::sleep_for(500ms);
    }
}
```

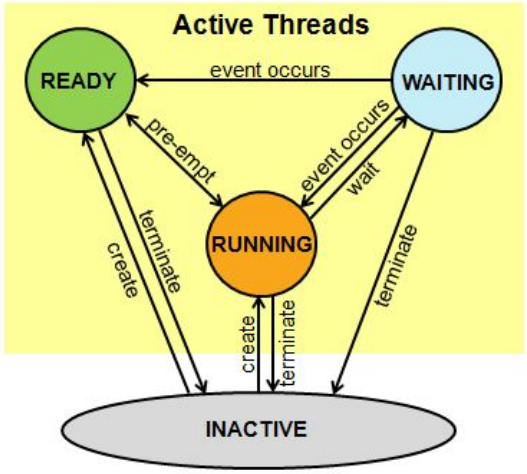
Thread switching

Time slices

- Mbed OS uses a timer internally to switch the active thread
- The minimum time a thread can run is called a *time slice*
- Each thread runs for this amount of time when it is active, at least

Thread priority

- The default priority is *osPriorityNormal*
- You can assign a different priority if you wish
- When a thread is *waiting* or thread switch triggers:
  - Mbed OS looks for threads in the *ready* state
  - Finds the highest priority *ready* thread
  - Runs thread until enter state *waiting* / *inactive* or the next thread switch



Mbed OS thread Finite State Machine

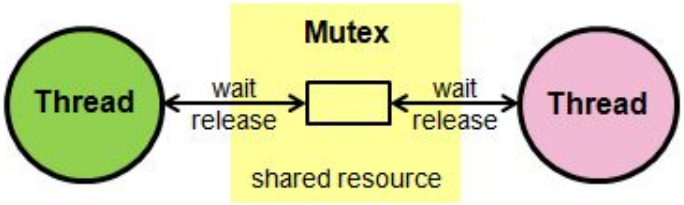
Concept: Mutex (mutual exclusion)

About mutexes

- Imagine you have two threads that access the same variables
- If there's a thread switch while data is written the data could* become corrupt
  - Half the string is cleared
  - Only the two first variables were written
  - Reading these half-written variables might result in errors
- A *mutex* can be used to protect access to shared data
- A mutex ensures that only one thread access the data

Using mutexes

- Mutexes are available through the *Mutex* class
- This class has two main functions:
  - *lock()* - Lock the mutex
  - *unlock()* - Unlock the mutex



The *Mutex* methods cannot be called from ISR, if synchronization is required in ISR, consider using semaphores**

Example: Mutex

```
#include "mbed.h"

Mutex stdio_mutex;
Thread t2;
Thread t3;

void notify(const char *name, int state)
{
    stdio_mutex.lock(); // Use a mutex to lock access to printf()
    printf("%s: %d\n\r", name, state);
    stdio_mutex.unlock();
}

void test_thread(void const *args)
{
    while (true) {
        notify((const char *)args, 0);
        ThisThread::sleep_for(1000ms);
        notify((const char *)args, 1);
        ThisThread::sleep_for(1000ms);
    }
}

int main()
{
    t2.start(callback(test_thread, (void *)"Th 2")); // callback is used to give the thread a parameter
    t3.start(callback(test_thread, (void *)"Th 3"));

    test_thread((void *)"Th 1");
}
```

Example: Shared struct

About

- We can use structs to share data between threads
- This type of data must also be protected against simultaneous access
- The code is just like last year - except the struct now has a mutex
- Threads lock this mutex when they want to access shared data

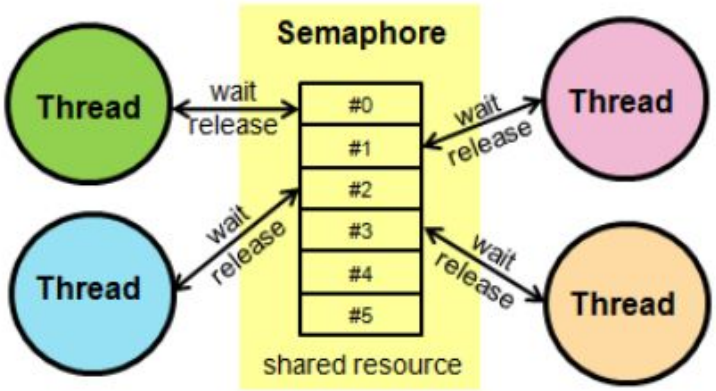
```
// Struct holding our data
typedef struct
{
  // Mutex that protects this data struct
  Mutex mutex;

  // Regular struct members (our data)
  char message[200];
} data_t;
```

Semaphores

About

- Semaphores are very similar to mutexes
- You *acquire* and *release* a semaphore.
  This is similar to *lock* and *unlock* for mutexes
- A semaphore can be acquired more than once.
  Useful when more than 1 shared resource is available
- *count* is the amount currently available, *max_count* is the maximum



Semaphore example

Almost identical to the mutex example.
Notice that a count is given when creating the semaphore.

```
#include "mbed.h"

Semaphore one_slot(1);
Thread t2;
Thread t3;

void test_thread(void const *name)
{
    while (true) {
        one_slot.acquire();
        printf("%s\n\r", (const char *)name);
        ThisThread::sleep_for(1000);
        one_slot.release();
    }
}

int main(void)
{
    t2.start(callback(test_thread, (void *)"Th 2"));
    t3.start(callback(test_thread, (void *)"Th 3"));

    test_thread((void *)"Th 1");
}
```

Common errors

Stack Overflow

This error occurs when a function uses up all its stack space. You might see this when using complex libraries or allocating a lot of local variables. You can increase the stack size in *mbed_app.json* to compensate.

Heap Overflow

This error occurs if any thread tries to allocate dynamic memory and that allocation fails. This means the you have used up all the memory on the microcontroller. This error usually occurs because of *memory leaks*, meaning memory that has been allocated but not freed.

Questions?

Resources

- Mbed OS threads
- Mbed OS mutexes
- Mbed OS semaphores