Lab instructions – Coursework 2, Part 2

Threading and Communication

Communication with host

So far, you have sent messages to the host using the serial port with `printf()` commands in the main function. This is not scalable with multiple tasks running because only one thread can call printf() at once and others will block. You also need to receive messages and you cannot make multiple threads receive simultaneously from the same serial port.

Creating dedicated tasks for communication handles the resource sharing neatly and allows you to buffer data to overcome bursty events or communication flows. It also makes it easier to analyse the latency of tasks since it removes dependencies that could block for a long time while communication takes place.

Outgoing communication

The first thread will receive messages from other parts of the code and write them to the serial port.

1. Refer to the example code for `Thread` and instantiate a new thread that will be used for the outgoing communication task: https://os.mbed.com/teams/mbed_example/code/rtos_basic.

   The class reference can be found here: https://os.mbed.com/docs/mbed-os/latest/apis/thread.html

   ⚠ If you need to import the mbed OS library, use the option 'import from URL' and enter the repository https://github.com/ARMmbed/mbed-os. Don't search for the library within the IDE – it doesn't return the correct version. Mbed examples use a variety of libraries but mbed-os contains everything you need to access RTOS functions.
2. You need a thread-safe method of passing messages to the output thread. A `Mail` object is ideal since it contains a FIFO buffer that can queue up several messages: https://os.mbed.com/docs/mbed-os/v5.11/apis/mail.html.

   Study the example code for `Mail` and run it on your microcontroller, if you like: https://os.mbed.com/teams/mbed_example/code/rtos_mail/. The function is similar to what is required for the communication task – one thread places messages into the buffer while another (the loop in `main()`) takes them out and prints them on the serial port.
3. `Mail` is a C++ template, which means it is customised during compilation to use a data type of your choosing. The example code packs three fields ('voltage', 'current' and 'counter') into a `struct` type so they are all passed together. Define your own struct type to contain the fields you need to pass.

4. It's worth creating a separate function to add messages to the queue because you will need to do this at multiple points in the code. Here is an example based on the contents of the while loop in `send_thread` in the Mail code example:

```
void putMessage(float voltage, float current, uint32_t
counter){
        mail_t *mail = mail_box.alloc();
        mail->voltage = (i * 0.1) * 33;
        mail->current = (i * 0.1) * 11;
        mail->counter = i;
        mail_box.put(mail);
    }
```

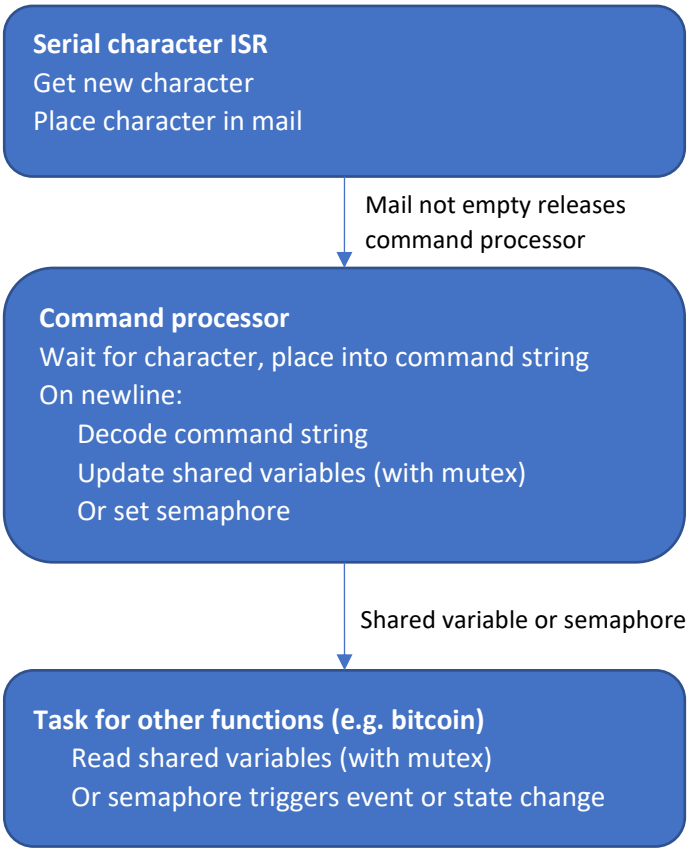
   First, call `Mail::alloc()` to return a pointer to the memory that will be used to store the message. Then, the code and data are written into that data structure and, finally, `Mail::put()` places the message pointer in the queue. The function is reentrant because the methods of `Mail` are documented as thread-safe and all other variables are local. It can be called by multiple threads simultaneously with no risk of data corruption.
5. The Mail code example uses a loop in main to receive messages and print them out. You will need to place similar code in your output thread. `Mail::get()` is a blocking function which waits for an item to be available in the buffer. `Mail::get()` returns type `osEvent`, a data structure that contains a pointer to the message itself. The pointer is dereferenced to access the values of `voltage, current and counter`, which are then printed to the serial port. Finally, the memory unit that was used is returned to the pool using `Mail::free()`.
6. Update your bitcoin miner so it sends matching nonces via the output message function. You don't need to place the entire message text on the queue: an integer code would suffice if it is replaced with the full message by the output thread.

Incoming communication

The specification requires you to act on certain commands sent by the serial port. This means receiving the incoming characters, decoding complete messages and acting on the commands.

The `Serial` class used in the starter code inherits C++ `Stream`, which makes it easy to use but difficult to analyse from a real-time perspective because, for example, it contains a 'hidden' interrupt handler. Instead, you should build functionality from scratch to handle incoming data byte-by-byte using the alternative `RawSerial`.

The following diagram shows an implementation with three tasks (including the final recipient of the message) and the dependencies between them:



The ISR allows the system to respond to each new byte before the next byte arrives – a thread would not handle the byte quickly enough because the scheduler tick interval is too long.

However, decoding the command is too complex for an ISR, so a mail is used to pass the bytes to a command processor thread. The mail buffers the data and allows the command processor to work in bursts when the scheduler permits.

Once the command is decoded, it is actioned by passing data or signals to other threads. Race conditions are avoided by using thread-safe objects such as mutexes and semaphores.

1. Change the serial class in your code from `Serial` to `RawSerial` and create an ISR that will receive each incoming byte and place it into a queue.
    a. Method `uint8_t RawSerial::getc()` retrieves a byte from the serial port.
    b. Create a global instance of class `Mail<uint8_t, 8>` to buffer incoming characters

```
RawSerial pc;
Mail<uint8_t, 8> inCharQ;

void serialISR(){
        uint8_t* newChar = inCharQ.alloc();
         *newChar = pc.getc();
        inCharQ.put(newChar);
    }
```

2. Create a new thread to decode commands. In the thread function, begin by attaching the new ISR to serial port events, then, in a while loop, use method `Mail::get()` to wait for each new character:

```
pc.attach(&serialISR);
while(1) {

osEvent newEvent = inCharQ.get();
uint8_t* newChar = (uint8_t*)newEvent.value.p;
...    //Store the new character
inCharQ.free(newChar);
...    //Decode the command if it is complete
```

   Handle the character by writing it on the end of a `string` or `char` array and check for a newline character `'\r'` to indicate the end of the command. Use `Mail::free()` to release the slot in the buffer when you have finished with the character.
   When a newline is received, decode the command by iterating through the string, possibly helped with library functions such as `sscanf()`. There is no need to use a regex library. Start by writing code to decode the 'new key' command, which is the letter 'K' followed by 16 hexadecimal digits.

   ⚠ Guard against buffer overflows arising from malformed or malicious transmissions that never terminate with a newline. If using a char array, make sure the string is appended with a termination character `'\0'` before passing it to library functions.

3. Acting on the command depends on its function. For example, a new bitcoin key can be written into a global variable where it will be read by the bitcoin function. The variable should be protected by a mutex to prevent simultaneous accesses:

```
//In command decode thread
      newKey_mutex.lock();
      newKey = receivedKey;
      newKey_mutex.unlock();
```

   The bitcoin task reads the key whilst it is calculating the SHA256 hash. The hash function takes a while and protecting the whole thing with a mutex would block the decode thread, so instead copy the global, shared key into the sequence array before each hash. Then, only the copy needs to be protected by the mutex:

```
//In bitcoin (main) thread
      newKey_mutex.lock();
      *key = newKey
      newKey_mutex.unlock();

      //sequence now contains the new key
      hasher.computeHash(hash, sequence, 64);
```

4. You should now be able to handle the input to and output from the bitcoin miner using the serial communication tasks all running in parallel with threads and interrupts.