RTOS with Mbed

Create RT Application with MBED

Jin Hyun Kim

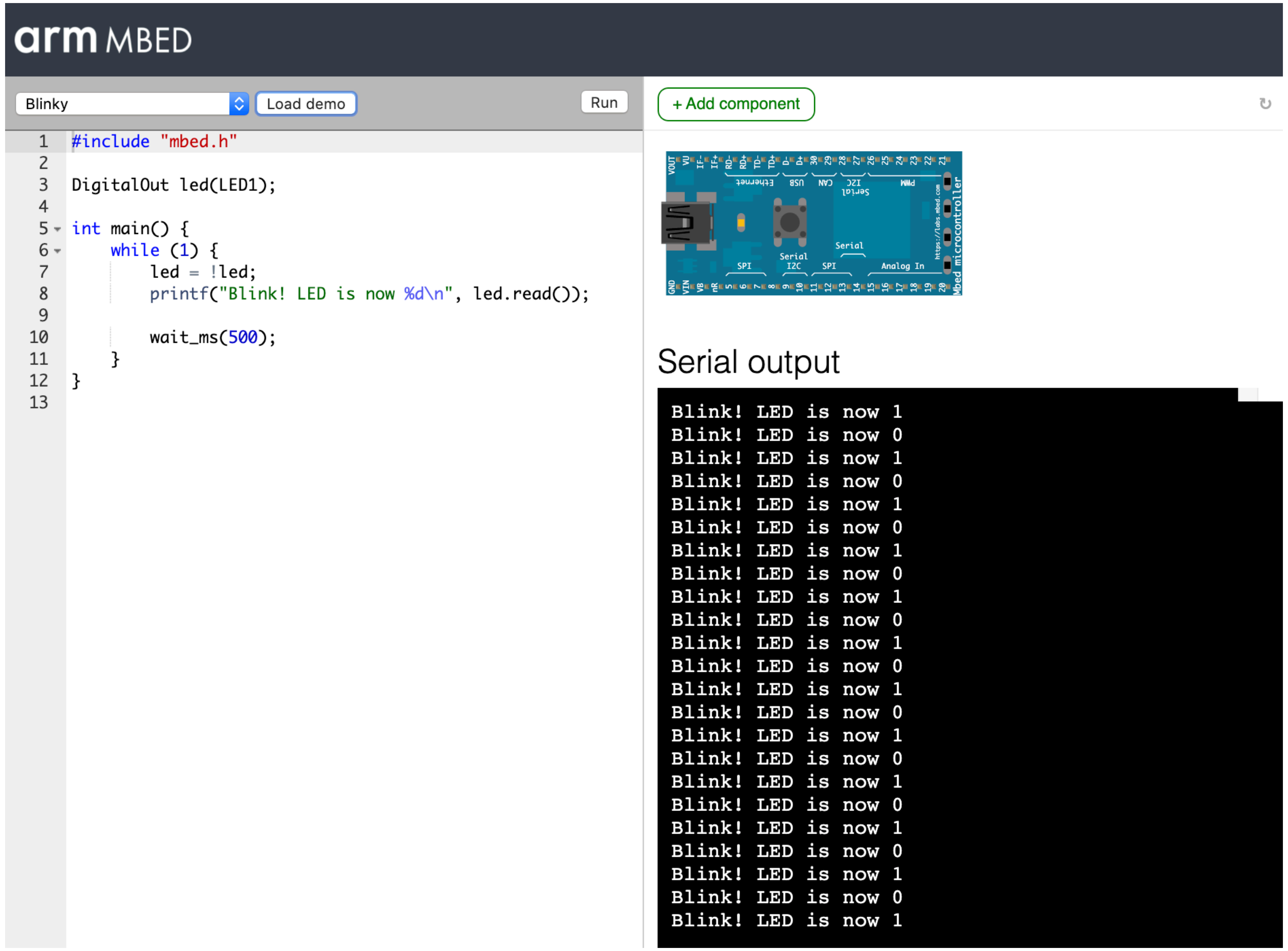
Mbed OS

- A free, open-source embedded operating system designed specifically for the "things" in the Internet of Things.
- Mbed's online compiler provides RTOS-like functionality with the Mbed-RTOS library, which must be imported to the project
- It provides common RTOS functionality such as threads, interrupts, mutexes, etc.

- Mbed OS API List - https://os.mbed.com/docs/mbed-os/v5.15/apis/index.html
- Compiler (Need to create account) - https://developer.mbed.org/compiler/
- **Simulator - https://simulator.mbed.com/**

MBed Simulator

Simulator - https://simulator.mbed.com/



Mbed OS

- https://os.mbed.com/docs/mbed-os/v5.15/apis/rtos.html
- The Mbed OS RTOS capabilities include
  - Managing objects such as threads,
  - Synchronization objects
  - Timers
- Mbed os
  - provides interfaces for attaching an application-specific idle hook function,
  - reads the OS tick count and
  - implements functionality to report RTOS errors.

RTOS Ticker

- Platforms using RTOS, including Mbed OS, need a mechanism for counting the time and scheduling tasks.
- A timer that generates periodic interrupts and is called system tick timer
- Under Mbed OS, we call this mechanism the RTOS ticker.
  - SysTick is a standard timer available on most Cortex-M cores.
  - Its main purpose is to raise an interrupt with set frequency (usually 1ms).
  - The Mbed OS platforms uses SysTick as the default RTOS ticker

Mbed RTOS Services

- Process Management
  - Thread

- IPC::Synchronization
  - Semaphore
  - Mutex
  - Signal

- IPC:: Message Communication
  - Queue
  - MemoryPool
  - Mail
- Time Management
  - RTOS Timer
  - Default Timeouts

- Interrupt Service Routines
- Status and Error Codes
- osEvent
- Implementation

RTOS APIs

- Thread: The class that allows defining, ***creating and controlling*** parallel tasks.
- ThisThread: The class with which you can **control** the current thread.
- Mutex: The class used to ***synchronize*** the execution of threads.
- Semaphore: The class that ***manages thread access to a pool of shared resources*** of a certain type.
- Queue: The class that allows you to queue pointers to data from producer threads to consumer threads.
- EventQueue: The class that provides a flexible queue for scheduling events
- UserAllocatedEvent: The class that provides APIs to create and configure static events

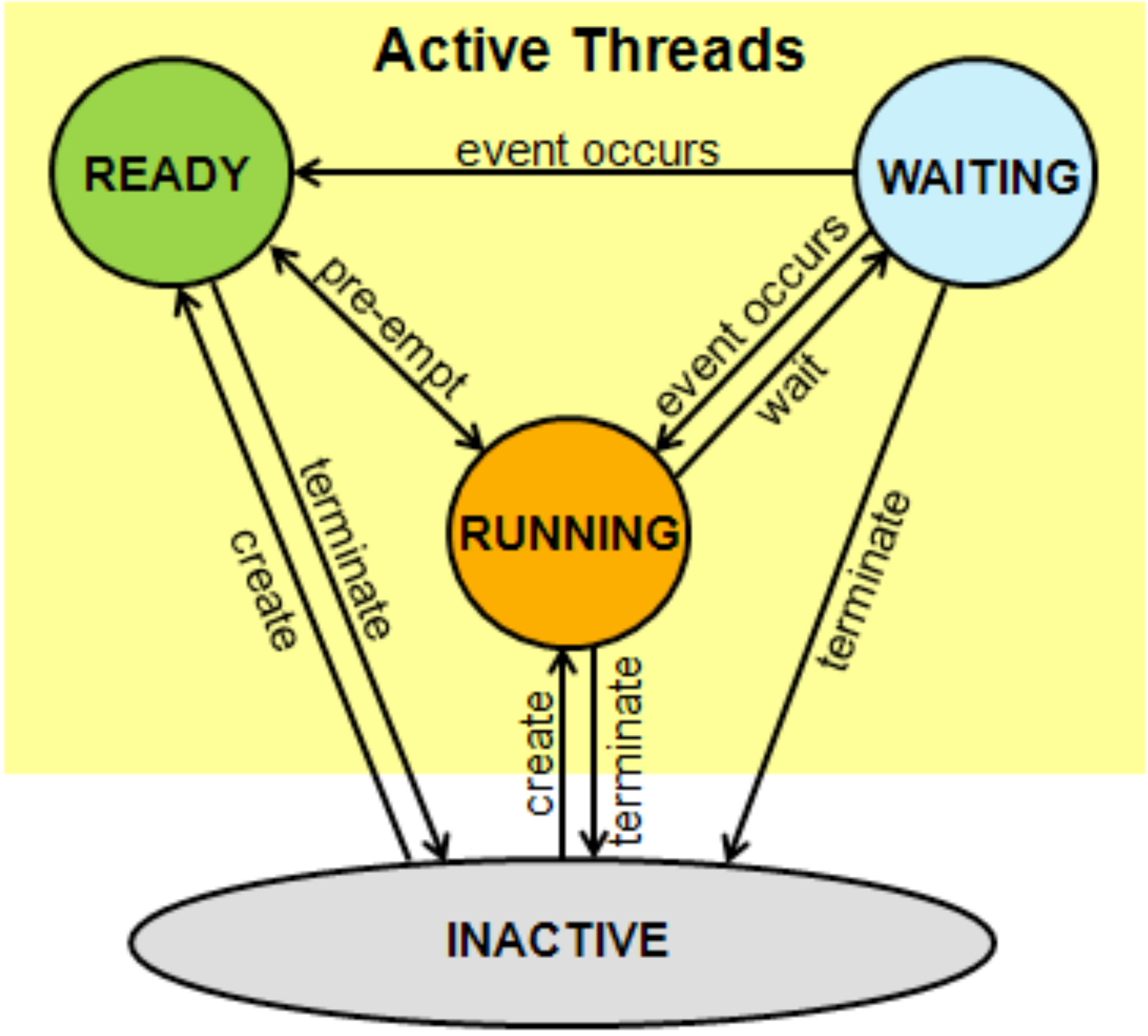
RTOS APIs

- MemoryPool: This class that you can use to define and manage fixed-size memory pools
- Mail: The API that provides ***a queue combined with a memory pool*** for allocating messages.
- EventFlags: An event channel that provides a generic way of notifying other threads about conditions or events. You can call some EventFlags functions from ISR context, and each EventFlags object can support up to 31 flags.
- Event: The queue to store events, extract them and execute them later.
- ConditionVariable: The ConditionVariable class provides a mechanism to safely wait for or signal a single state change. You cannot call ConditionVariable functions from ISR context.
- Kernel: Kernel namespace implements functions to control or read RTOS information, such as tick count.

Thread

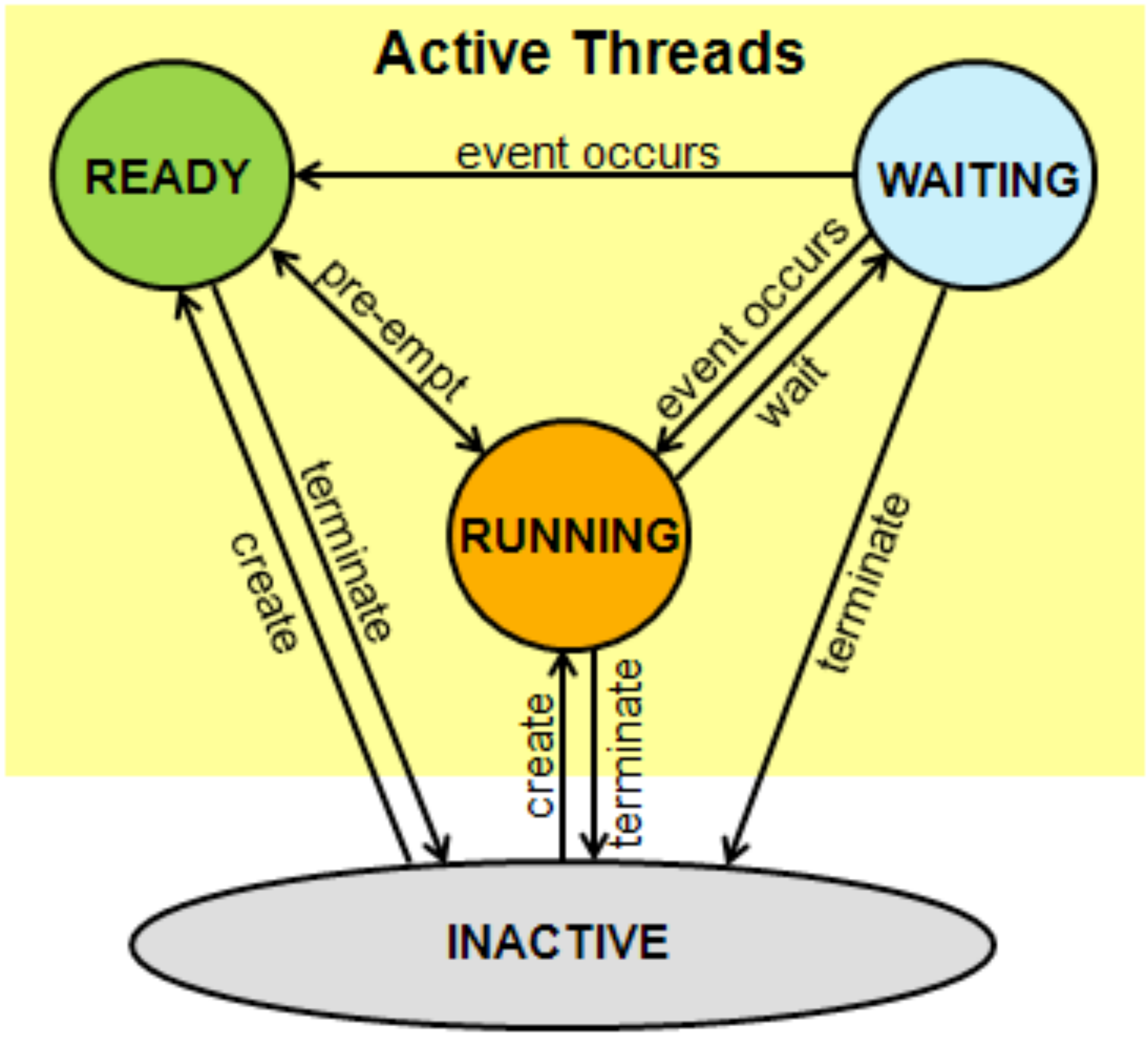
Process Management

- RTOS provides an easy way to turn functions into threads
- Threads start when they are declared
- These threads have priorities
- They can handle receive and handle signals
- They can block or yield from the processor using wait(ms) and yield commands. osWaitForever can be used to block indefinitely



Thread

Process Management



- Running: The currently running thread. Only one thread at a time can be in this state.
- Ready: Threads that are ready to run. Once the running thread has terminated or is waiting, the ready thread with the highest priority becomes the running thread.
- Waiting: Threads that are waiting for an event to occur.
- Inactive: Threads that are not created or terminated. These threads typically consume no system resources.

Thread

Create and Run a Thread

Blinky | Load demo

```
#include "mbed.h"

DigitalOut led(LED1);

int main() {
    while (1) {
        led = !led;
        printf("Blink! LED is now %d\n", led.read());
        wait_ms(500);
    }
}
```

```
#include "mbed.h"

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

void led2_thread() {
    while (true) {
        led2 = !led2;
        wait(1);
    }
}

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

    while (true) {
        led1 = !led1;
        wait(0.5);
    }
}
```

Mutex

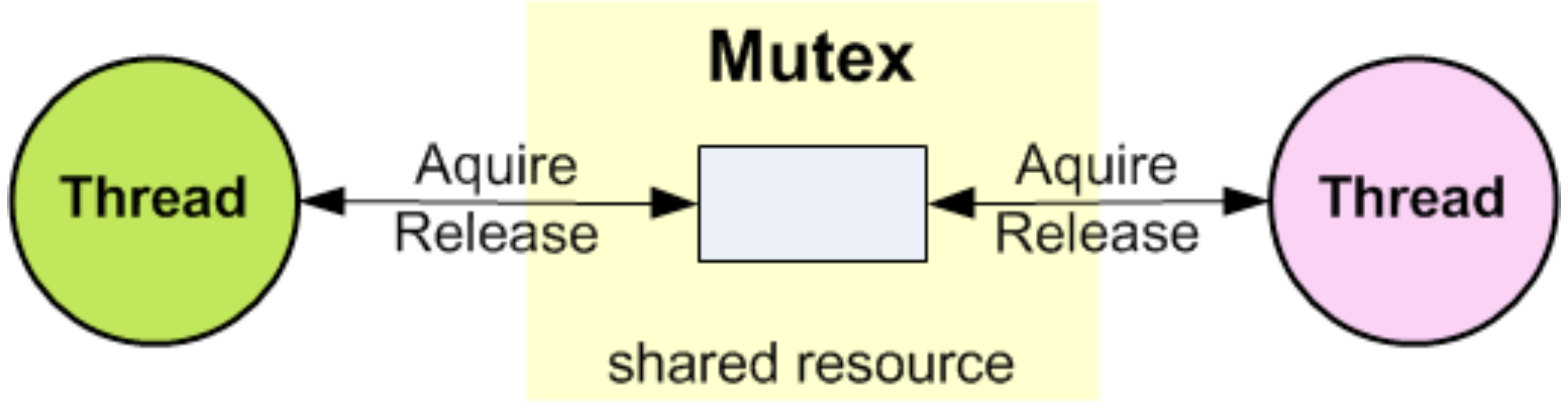
Problem

Process W	Process X	Process Y	Process Z
Read (x) x = x + 1; Write (x)	Read (x) x = x + 1; Write (x)	Read (x) x = x - 2; Write (x)	Read (x) x = x - 2; Write (x)

Mutex

Synchronization

- A mutex ensures mutual exclusion when accessing a resource



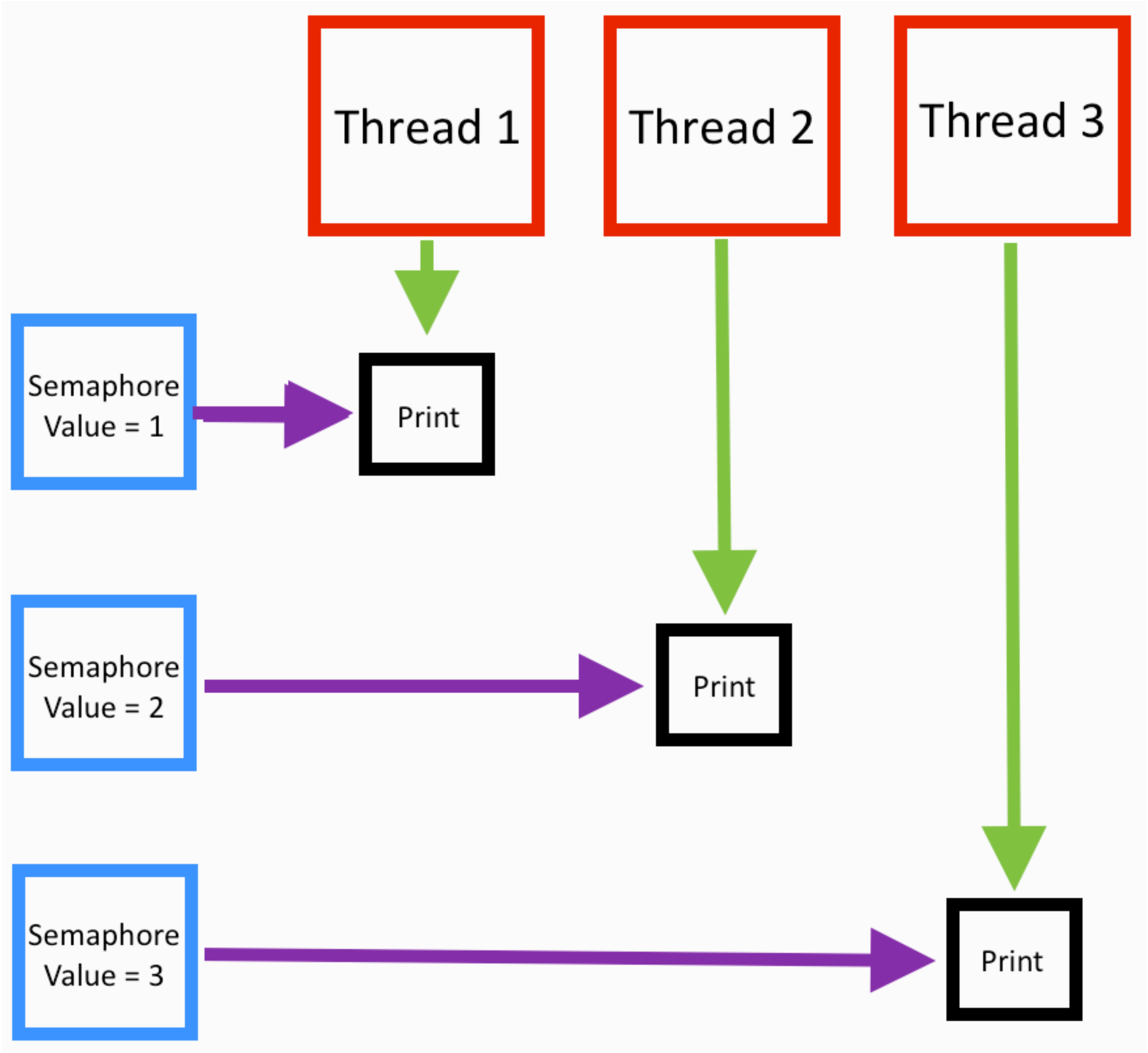
- Provide lock, trylock (non-blocking), and unlock functions
- The Mutex methods cannot be called from interrupt service routines (ISR).

Behavior: Uses a mutex to control printf statement invocations so only one thread can call it at a time

```
#include "mbed.h"

Mutex stdio_mutex;
Thread t2;
Thread t3;

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

void test_thread(void const *args) {
    while (true) {
        notify((const char*)args, 0); wait(1);
        notify((const char*)args, 1); wait(1);
    }
}

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

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

Semaphore
Synchronization

Counting Semaphore



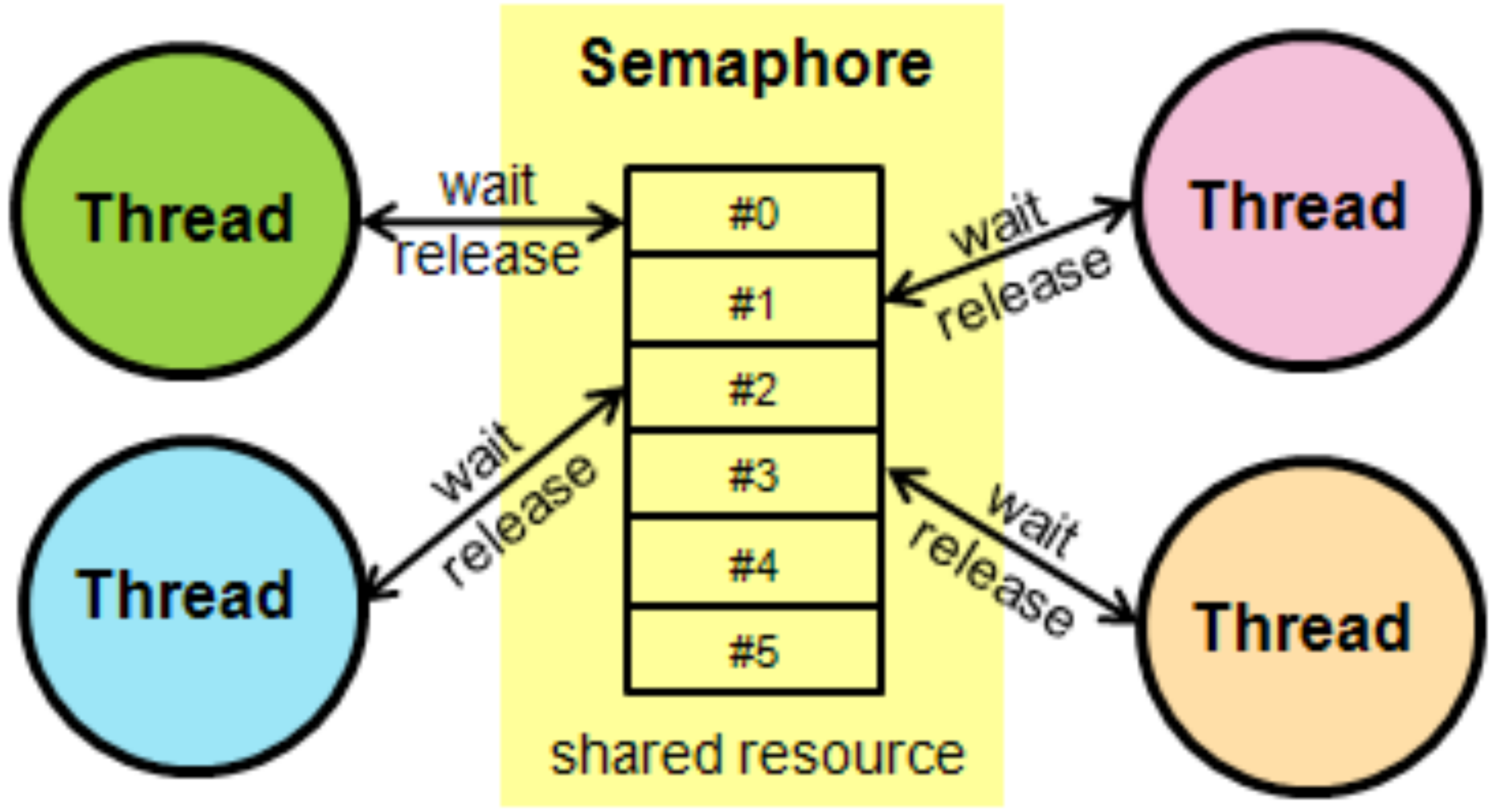
Semaphore

Synchronization

- Semaphores are the generalization of mutex that can handle resources with more than one instance. That is, it can count number of open resources instead of being on/off.

- Mutexes are a special case of semaphores where there is one instance of a resource

- Provide wait and release functions



Semaphore

Synchronization

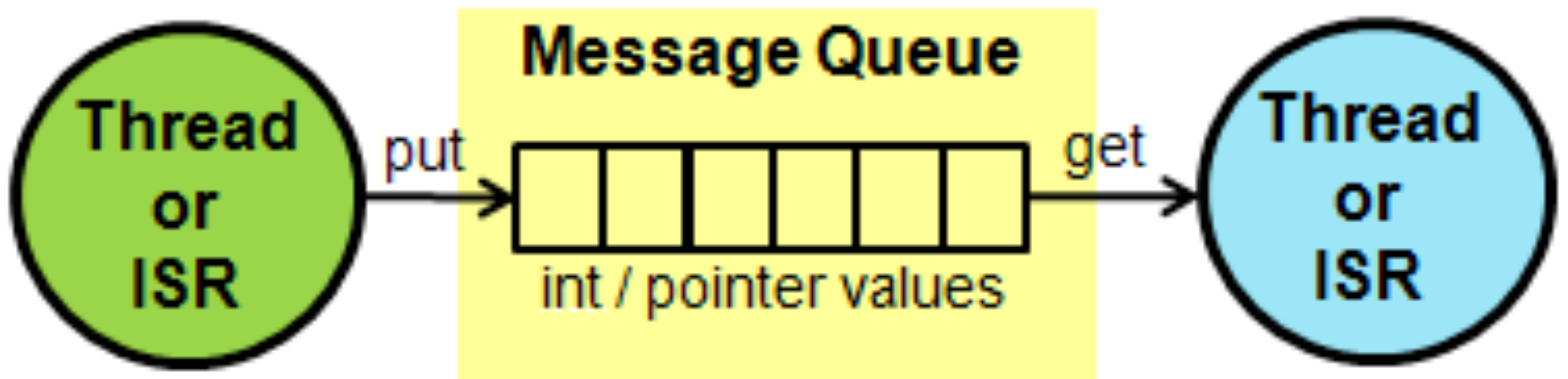
- Use Semaphore to protect printf().

```
#include "mbed.h"

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

void test_thread(void const *name) {
    while (true) {
        one_slot.wait();
        printf("%s\n\r", (const char*)name);
        wait(1);
        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");
}
```

Queue

Message Communication

- A Queue allows you to queue pointers to data from producer threads to consumer threads.

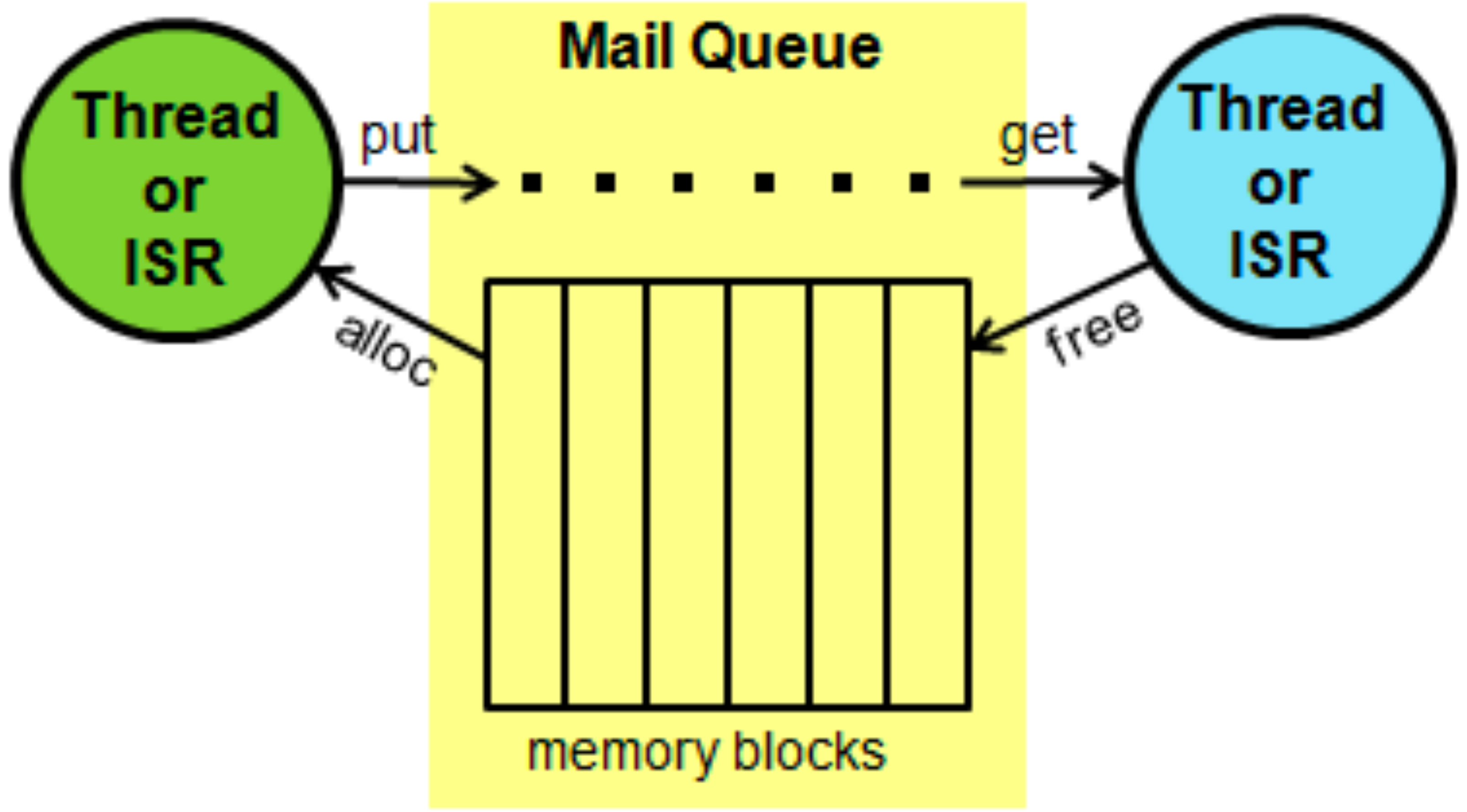



```
#include "mbed.h"

typedef struct {
    float    voltage;   /* AD result of measured voltage */
    float    current;   /* AD result of measured current */
    uint32_t counter;   /* A counter value               */
} message_t;

MemoryPool<message_t, 16> mpool;
Queue<message_t, 16> queue;
Thread thread;

/* Send Thread */
void send_thread (void) {
    uint32_t i = 0;
    while (true) {
        i++; // fake data update
        message_t *message = mpool.alloc();
        message->voltage = (i * 0.1) * 33;
        message->current = (i * 0.1) * 11;
        message->counter = i;
        queue.put(message);
        wait(1);
    }
}

int main (void) {
    thread.start(callback(send_thread));

    while (true) {
        osEvent evt = queue.get();
        if (evt.status == osEventMessage) {
            message_t *message = (message_t*)evt.value.p;
            printf("\nVoltage: %.2f V\n\r"   , message->voltage);
            printf("Current: %.2f A\n\r"     , message->current);
            printf("Number of cycles: %u\n\r", message->counter);

            mpool.free(message);
        }
    }
}
```

Behavior: Adds a message (pointer) to a queue, and then dequeues it using proper dequeuing procedure

Mail

- A Mail works like a queue with the added benefit of providing a memory pool for allocating messages (not only pointers)




```
#include "mbed.h"

/* Mail */
typedef struct {
  float    voltage; /* AD result of measured voltage */
  float    current; /* AD result of measured current */
  uint32_t counter; /* A counter value               */
} mail_t;

Mail<mail_t, 16> mail_box;
Thread thread;

void send_thread (void) {
    uint32_t i = 0;
    while (true) {
        i++; // fake data update
        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);
        wait(1);
    }
}

int main (void) {
    thread.start(callback(send_thread));

    while (true) {
        osEvent evt = mail_box.get();
        if (evt.status == osEventMail) {
            mail_t *mail = (mail_t*)evt.value.p;
            printf("\nVoltage: %.2f V\n\r"   , mail->voltage);
            printf("Current: %.2f A\n\r"     , mail->current);
            printf("Number of cycles: %u\n\r", mail->counter);

            mail_box.free(mail);
        }
    }
}
```

EventQueue

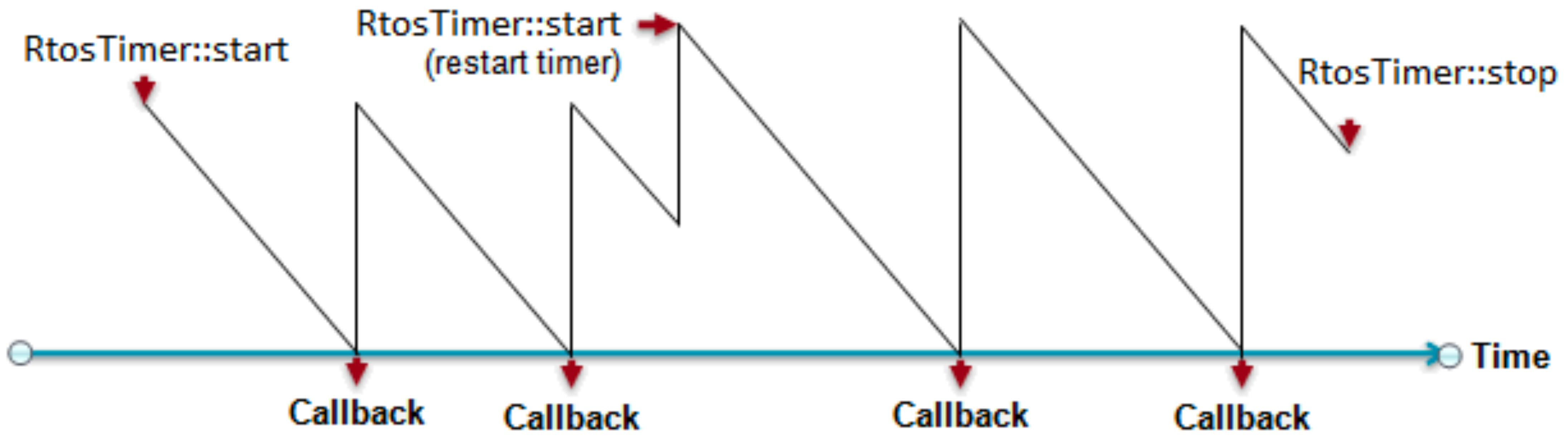
- The EventQueue class provides a flexible queue for scheduling events.
- You can use the EventQueue class for synchronization between multiple threads, or to move events out of interrupt context (deferred execution of time consuming or non-ISR safe operations).
- The EventQueue class is thread and ISR safe

EventQueue example: posting events to the queue

```
#include "mbed_events.h"
#include <stdio.h>

int main() {
    // creates a queue with the default size
    EventQueue queue;

    // events are simple callbacks
    queue.call(printf, "called immediately\n");
    queue.call_in(2000, printf, "called in 2 seconds\n");
    queue.call_every(1000, printf, "called every 1 seconds\n");

    // events are executed by the dispatch method
    queue.dispatch();
}
```

RTOS Timer

- The RtosTimer class allows creating and and controlling of **timer** functions in the system.
- A timer function is called when a time period expires whereby both one-shot and periodic timers are possible.
    - A timer can be started, restarted, or stopped.
    - Timers are handled in the thread osTimerThread.
    - Callback functions run under control of this thread and may use CMSIS-RTOS API calls.



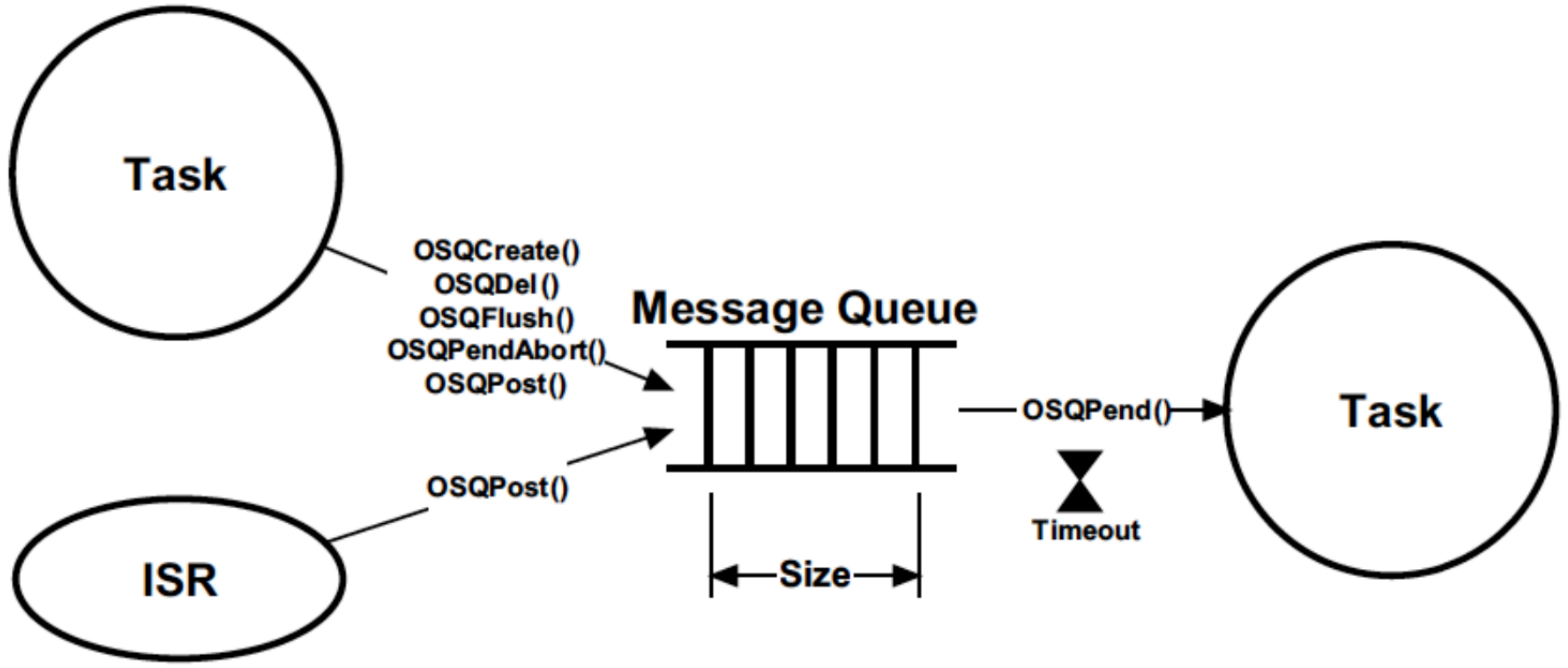
RTOS Timer

rtos_timer - main.cpp

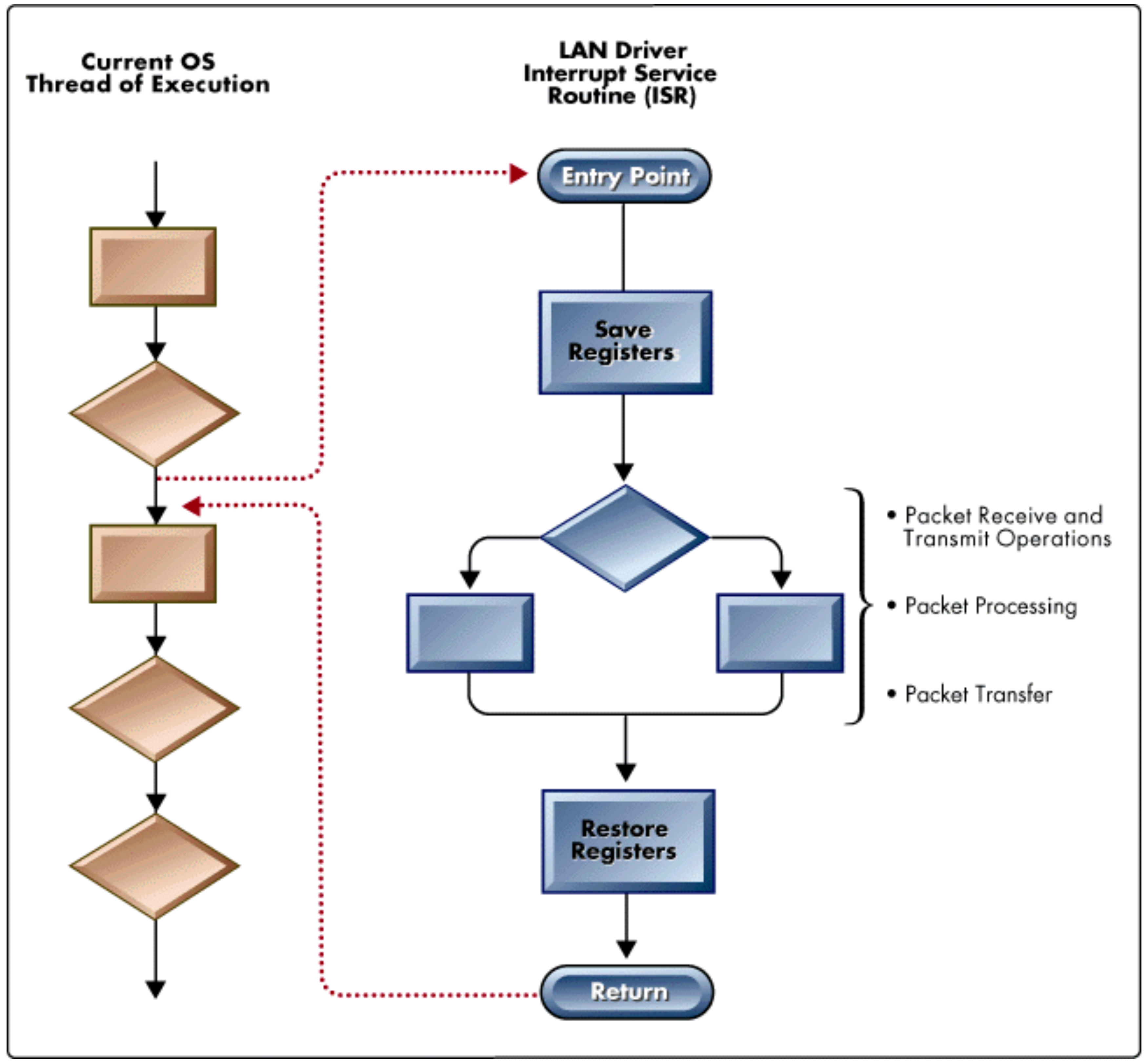
```
#include "mbed.h"
#include "rtos.h"

DigitalOut LEDs[4] = {
    DigitalOut(LED1), DigitalOut(LED2), DigitalOut(LED3), DigitalOut(LED4)
};

void blink(void const *n) {
    LEDs[(int)n] = !LEDs[(int)n];
}

int main(void) {
    RtosTimer led_1_timer(blink, osTimerPeriodic, (void *)0);
    RtosTimer led_2_timer(blink, osTimerPeriodic, (void *)1);
    RtosTimer led_3_timer(blink, osTimerPeriodic, (void *)2);
    RtosTimer led_4_timer(blink, osTimerPeriodic, (void *)3);

    led_1_timer.start(2000);
    led_2_timer.start(1000);
    led_3_timer.start(500);
    led_4_timer.start(250);

    Thread::wait(osWaitForever);
}
```

Interrupt

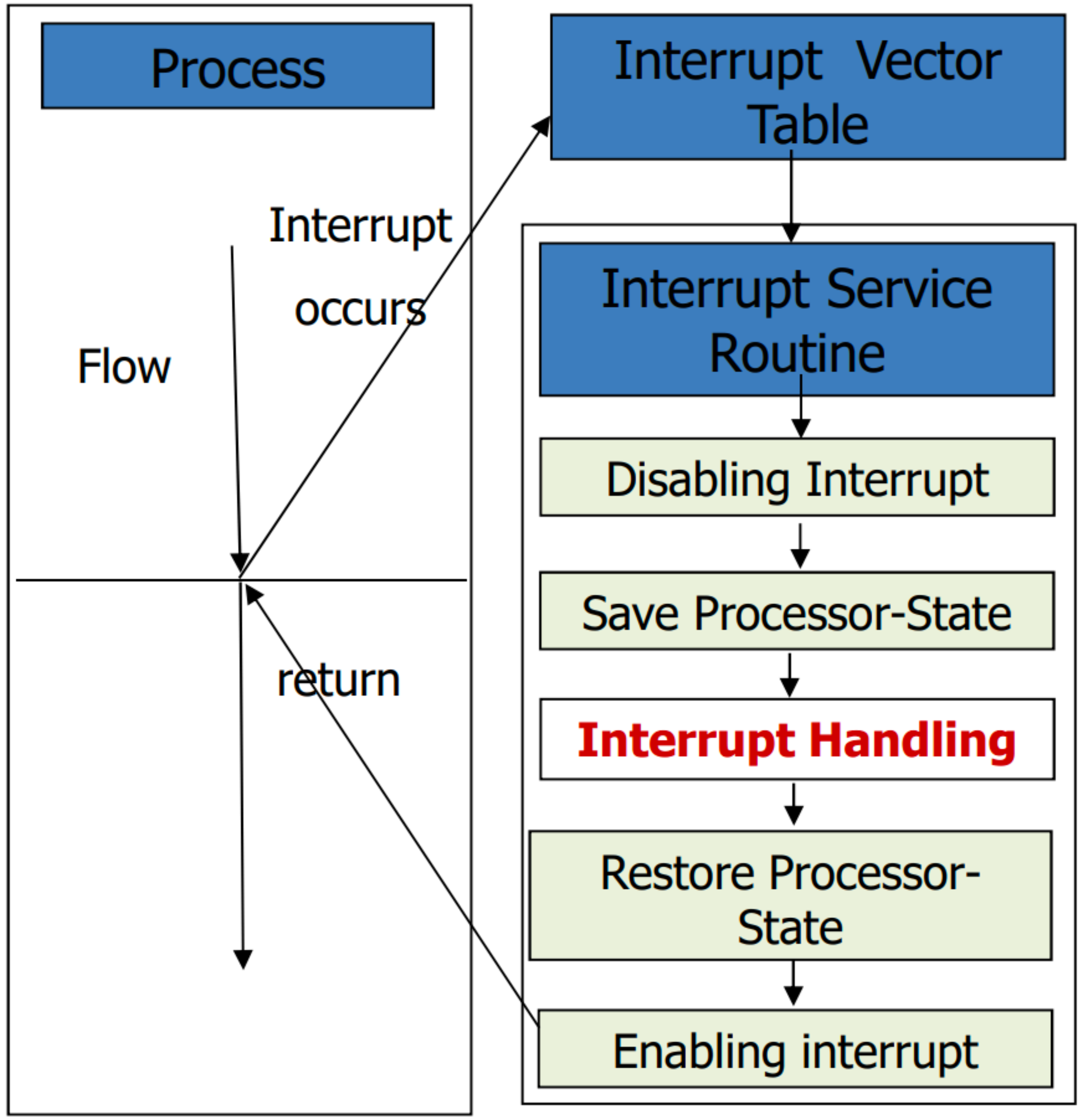
- An event that requires the CPU to stop the current program execution and perform ***some service*** related to the event



Interrupt Service Routines



http://support.novell.com/techcenter/articles/img/ana1995050101.gif



http://csmylov.blogspot.com/2017/08/interrupt.html

Interrupt Service Routines

- The same RTOS API can be used in ISR. The only two warnings are:
  - Mutex can not be used.
  - Wait in ISR is not allowed: all the timeouts in method parameters have to be set to 0 (no wait).

rtos_isr - main.cpp

```cpp
#include "mbed.h"
#include "rtos.h"

Queue<uint32_t, 5> queue;

DigitalOut myled(LED1);

void queue_isr() {
    queue.put((uint32_t*)2);
    myled = !myled;
}

void queue_thread(void const *args) {
    while (true) {
        queue.put((uint32_t*)1);
        Thread::wait(1000);
    }
}

int main (void) {
    Thread thread(queue_thread);

    Ticker ticker;
    ticker.attach(queue_isr, 1.0);

    while (true) {
        osEvent evt = queue.get();
        if (evt.status != osEventMessage) {
            printf("queue->get() returned %02x status\n\r", evt.status);
        } else {
            printf("queue->get() returned %d\n\r", evt.value.v);
        }
    }
}
```

Behavior: Uses a timer ISR to queue values of “2” every second. The main thread then consumes these values and prints them, or the error/status code if one is generated