ECSE 426 – Fall 2015
Microprocessor Systems

Dept. Electrical and Computer Engineering
McGill University

Embedded Operating Systems

- Divide available CPU processing time into a number of time slots
- Execute different tasks in different slots
- Switching between tasks may happen hundreds of times per second
  - Processor appears to be executing tasks in parallel

Embedded Operating Systems

- Anything from simple task scheduler to…
- Fully featured OS
  - Features like Linux (but not Linux or Windows)
    - μClinux : no memory management but high memory (2 Mb SRAM)
  - No virtual memory system support for Cortex-M

Memory management

- Memory Management Unit (MMU)
  - On Cortex-A processors, not Cortex-M
  - Each process sees flat virtual address space
  - MMU does dynamic remapping to physical address space
  - Managing virtual memory can lead to large delays
    - Have to locate and transfer address mapping information
    - Move it from memory (page table) to hardware in the MMU
  - An OS that uses virtual memory cannot guarantee real-time response

Memory protection

- Memory Protection Unit (MPU): available on Cortex-M processors
  - Only guarantees memory protection
  - No memory address translation

  - Programmable device
    - Identify up to 8 memory regions
    - Define memory access permissions (privileged/full)
    - Specify memory attributes (e.g., bufferable, cacheable)

Memory protection

- Memory Protection Unit (MPU)
  - Increased robustness and security
    - Prevent applications corrupting OS kernel stack
    - Prevent unprivileged tasks from accessing certain peripherals
  - If memory access violates memory permissions
    - Transfer blocked
    - Fault exception triggered -> MemManage fault
    - OS can reset system or terminate offending task

Embedded Operating Systems

- More than 30 embedded OSes available for Cortex-M
  - Most are simple (and have no user interface)
  - Main goal is to provide multi-tasking

- Some provide additional software support
  - Communication protocol stack (TCP/IP)
  - File system (FAT)
  - Graphical user interface

Potential OS Services

- Process management
  - Create, terminate, signal
- File management
  - Open, read, write, close, lock
- Memory management
  - Virtual memory, sharing, protection

- Date and time
- User management
- Networking
- Graphical User Interface

When does embedded OS make sense?

- Many applications do not need an embedded OS
- If tasks are all short and don't overlap much: more efficient to use an interrupt-driven arrangement

When does embedded OS make sense?

- Many applications do not need an embedded OS
- If tasks are all short and don't overlap much: more efficient to use an interrupt-driven arrangement

- Benefits:
  - provide scalable way of enabling several concurrent tasks to run in parallel
  - Additional safety features (stack-space checking, MPU support)
  - Feature support can reduce application-specific coding burden

When does embedded OS make sense?

- Many applications do not need an embedded OS
- If tasks are all short and don't overlap much: more efficient to use an interrupt-driven arrangement

- Disadvantages
  - Embedded OS requires extra memory overhead (5 - >100 KB)
  - Execution time overhead for context switching + scheduling (usually small)
  - Possible licence or royalty fees
  - Portability: some OSes are specific to certain toolchains or microcontrollers

Real-time Operating Systems (RTOS)

- When certain event occurs, trigger a corresponding task
  - Must happen within a specified timeframe
- Normally small memory footprint
- Very fast context switching
- Lots of embedded RTOS choices
  - RTX (ARM), Free RTOS, μC/OS
- Support from ARM Cortex hardware
  - SVC (Supervisor Calls), PendSV (Pending Service Calls), SysTick

Abstracting Away OS – CMSIS-RTOS

- CMSIS: Cortex-M Software Interface Standard
- CMSIS-RTOS
  - extension of existing RTOS designs
  - Allows middleware to be designed to work with multiple RTOS products

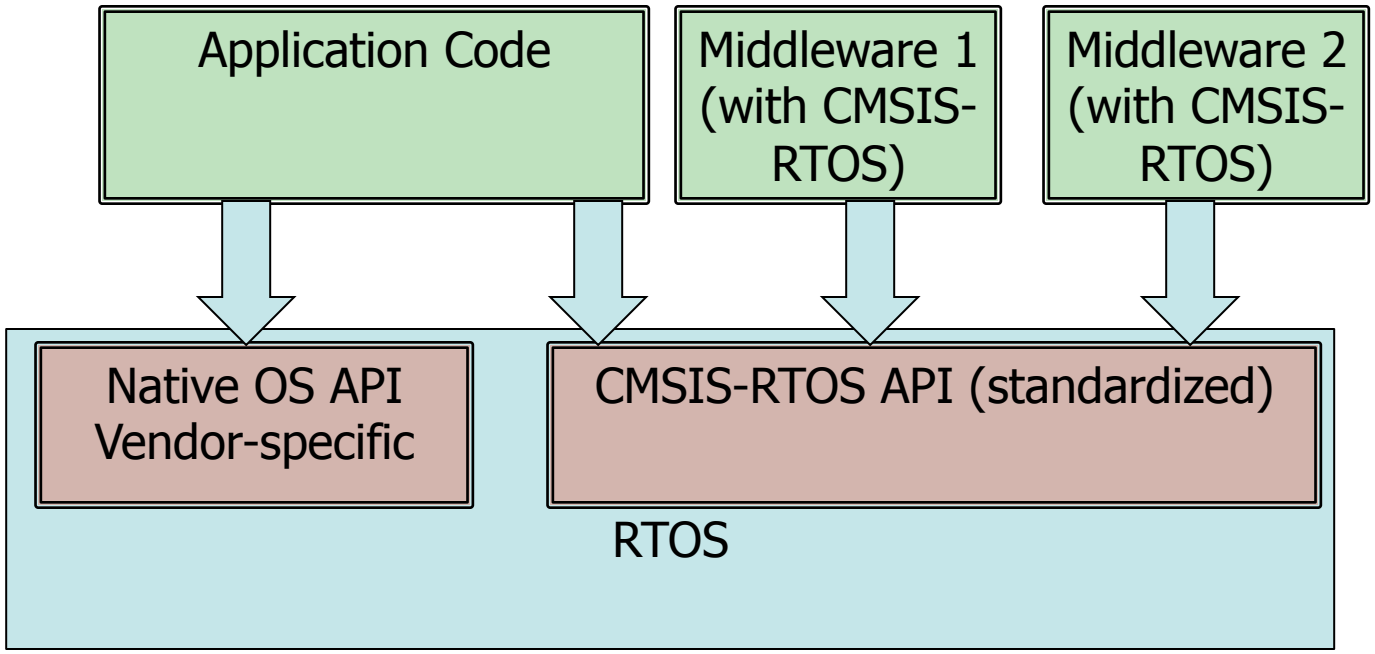
Abstracting Away OS – CMSIS-RTOS

- Middleware products can be complex
  - May need to use OS task scheduling features
  - Example: TCP/IP stack runs as a task and then needs to spawn out additional children tasks

- Traditional solution: middleware includes an OS emulation layer
  - Software integrator needs to port when using a different OS
  - Additional work for developers; increased project risk

Abstracting Away OS – CMSIS-RTOS

- CMSIS-RTOS: Makes RTOS-es look the same
- Implemented as additional set of APIs or wrapper for existing OS APIs
- API is standardized
  - Middleware developed using it should work with any embedded OS that supports CMSIS-RTOS



Thread Management

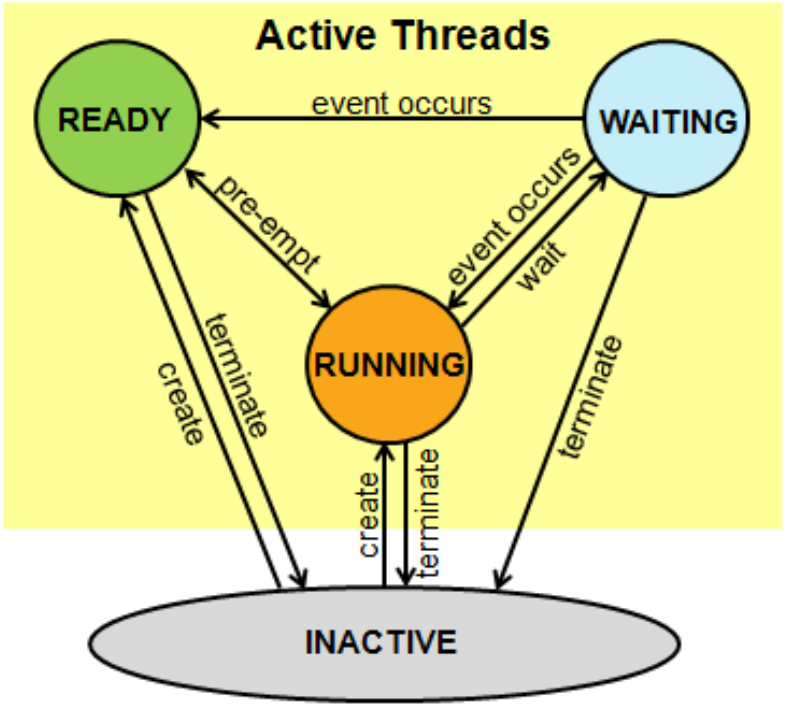
- Thread: each concurrent program

- CMSIS-RTOS provides support to define, create, manage threads
- Each thread has programmable priority
  - `osPriority  = {osPriorityIdle,osPriorityLow, …,osPriorityRealTime,osPriorityError}`

- Thread states allow for controlled multithreading
- Context switching: round robin or deterministic

Thread States

- RUNNING
- READY
  - In queue of thread states ready to run
  - RTX will select next highest priority thread
- WAITING
  - Delay request to complete
  - Event from another thread
- INACTIVE
  - Has not been started or has been terminated



Thread Communication

- Signal Events
- Semaphores
- Mutex
- Mailbox/message

Signal Communication: Example

```
#include cmsis_os.h
osThreadId t_blinky; // thread ID for blink
void blinky(void const *argument); //Thread
void LedOutputCfg(void); //Thread configuration

// Blinky: toggle LED Bit 12 (Unprivileged Thread)
void blinky(void const *argument) {
 while(1) {
  osSignalWait(0x0001, osWaitForever);
  if (GPIOD->IDR & (1<<12)) {
    GPIOD->BSSRH = (1<<12); //Clear Bit 12
  } else {
    GPIOD->BSSRL = (1<<12); //Set Bit 12
  }
 }
}
```

Signal Communication: Example

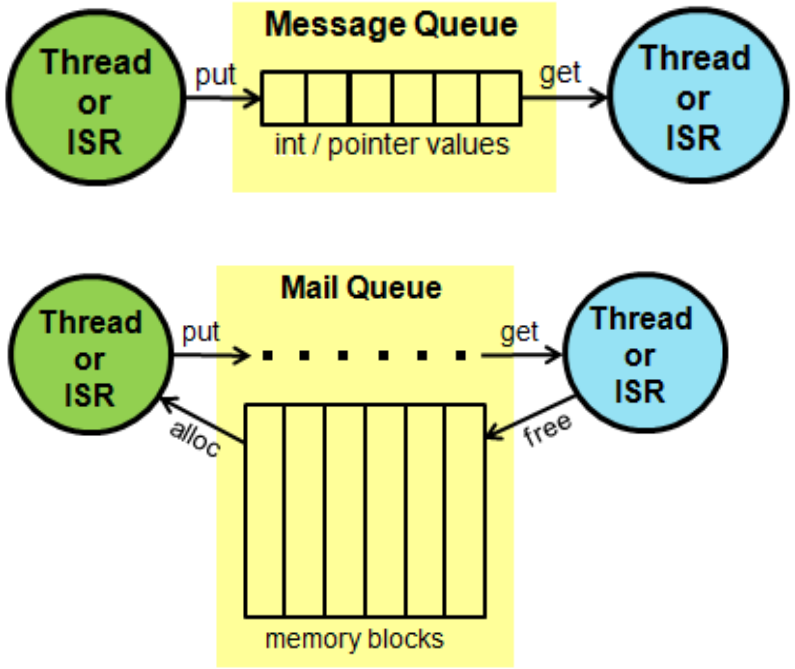
```
osThreadDef(blinky, osPriorityNormal, 1, 0);
// Name, priority, instances, stacksz

int main(void) {
 LedOutputCfg(); // Initialize LED Output
 t_blinky = osThreadCreate(osThread(blinky), NULL);
 while(1) {
  if (GPIOD->IDR & (1<<13)) {
    GPIOD->BSSRH = (1<<13); //Clear Bit 13
  } else {
    GPIOD->BSSRL = (1<<13); //Set Bit 13
  }
  osSignalSet(t_blinky,0x0001); //Set signal
  osDelay(1000);  //delay 1000 msec
 }
}
```

Thread Communication

- Signal Events
- Semaphores
- Mutex
- Mailbox/message

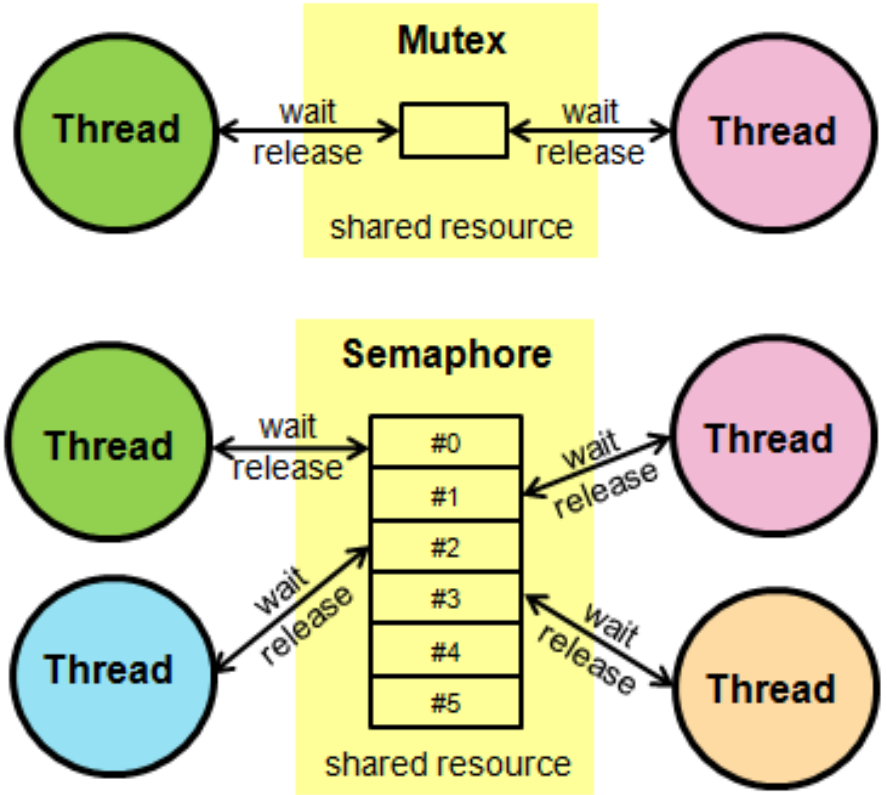
Services: Signals, Messages, Mail

- Messages: queuing
  - Integer or pointer
- Mail: larger blocks



Mutex, Semaphore

- Mutex: thread synchronization for access protection
- Semaphore:
  - Counting
  - Binary



Semaphore: signalling

```
osSemaphoreId sem1;
osSemaphoreDef(sem1);
void thread1 (void)
{
        sem1 = osSemaphoreCreate(osSemaphore(sem1), 0);
        while(1)
        {
                FuncA();
                osSemaphoreRelease(sem1)
        }
}
```

Semaphore: signalling

```
void task2 (void)
{
        while(1)
        {
                osSemaphoreWait(sem1,osWaitForever)
                FuncB();
        }
}
```

Semaphore: multiplexing

```
osSemaphoreId multiplex;
osSemaphoreDef(multiplex);
void thread1 (void)
{
 multiplex =osSemaphoreCreate(osSemaphore(multiplex), FIVE_TOKENS);
 while(1)
        {
                osSemaphoreWait(multiplex,osWaitForever)
                ProcessBuffer();
                osSemaphoreRelease(multiplex);
        }
}
```

Mutex

- Mutual exclusion
- Specialized semaphore (more rigid but safer)
- Mutex can only contain one token – cannot be created or destroyed

- Main use: control access to a peripheral

```
osMutexId uart_mutex; // declare
osMutexDef(uart_mutex); // initialize

uart_mutex = osMutexCreate(osMutex(uart_mutex));//create
osMutexWait(osMutexId mutex_id,uint32_t millisec);//wait
// … process; use peripheral
osMutexRelease(osMutexId mutex_id); // release
```

Data Exchange

- Message and mail queues
- Message queue

```
osMessageQId Q_LED; // Declare

// Define message queue with 16 storage elements
// Each element is an unsigned int
osMessageQDef(Q_LED,16_Message_Slots,unsigned int);

// Define an osEvent variable to retrieve data
// osEvent is a union (multiple ways to retrieve data)
osEvent result;
```

Data Exchange

- Message queue

```
// Declare in a thread
Q_LED = osMessageCreate(osMessageQ(Q_LED),NULL);

// Put data into the queue from one thread
osMessagePut(Q_LED,0x0,osWaitForever);

// Read data from another thread
result = osMessageGet(Q_LED,osWaitForever);
LED_data = result.value.v;
```

- Can also post a pointer to a more complex object