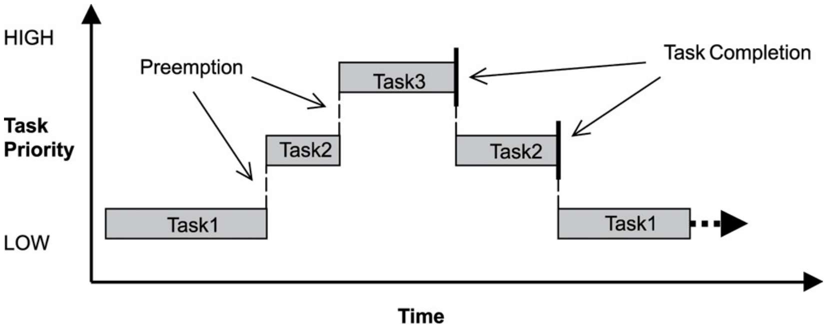
Embedded systems engineering
Distributed real-time systems

David Kendall

Introduction

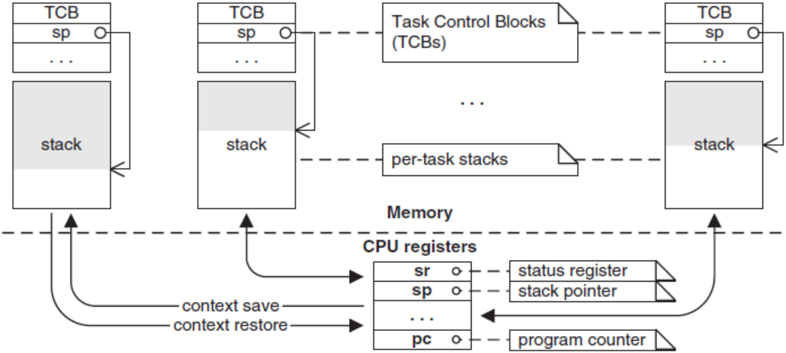
- Implementation of preemptive scheduling
- μC/OS-II (uC/OS-II)
  - Task management
  - Delay
  - Semaphores

Fixed-priority preemptive scheduling



- We focus on fixed-priority pre-emptive scheduling

Inside a preemptive scheduler



(Samek, 2008, p.259)

uC/OS-II: A real-time operating system

- Multi-tasking
- Preemptive
- Predictable
- Robust and reliable
- Standards-compliant
- Portable
- Scalable
- Source code available

uC/OS-II Services

- Task management
- Delay management
- Semaphores
- Mutual exclusion semaphores
- Event flags
- Message mailboxes
- Message queues
- Memory management
- Timers
- Miscellaneous

Task behaviour

- The behaviour of a task is defined by a C function that:
  1. never terminates
  2. blocks repeatedly

Example of task behaviour definition

```
static void appTaskLedGreen(void *pdata) {
  DigitalOut green(LED_GREEN);

  green = 0;
  while (true) {
    ledToggle(green);
    OSTimeDlyHMSM(0,0,0,500);
  }
}
```

Tasks: other requirements

Tasks need a priority level:

Priority

- Used for fixed-priority pre-emptive scheduling
- a number between 0 and `OS_LOWEST_PRIO`
- low number ⇒ high priority
- high number ⇒ low priority
- OS reserves priorities 0 to 3 and `OS_LOWEST_PRIO` - 3 to `OS_LOWEST_PRIO`
- Advice: give your highest priority task priority level 4 by declaring an enumeration of priority constants, starting at 4. Write the enumeration in priority order and let the compiler assign the values of the remaining priorities
- Example

```
typedef enum {
  LED_RED_PRIO = 4,
  LED_GREEN_PRIO
} taskPriorities_t;
```

Tasks: other requirements

Stack

- Each task needs its own data area (stack) for storing
  - context
  - local variables
- Example stack definition

```
#define  LED_GREEN_STK_SIZE          256
static OS_STK ledGreenStk[LED_GREEN_STK_SIZE];
```

User data

- Optionally tasks can be given access to user data when they are created
- We will not use this feature in this module
- Advice: always specify this as `(void *)0` when creating a task

Task creation

- A task is created using the OS function

```
INT8U OSTaskCreate(
        void (*task)(void *pdata), /* function for the task */
        void *pdata,               /* pointer to user data for the task function */
        OS_STK *ptos,              /* pointer to top of stack */
        INT8U priority             /* task priority */
);
```

Example

```
OSTaskCreate(appTaskLedGreen,
             (void *)0,
             (OS_STK *)&ledGreenStk[LED_GREEN_STK_SIZE - 1],
             LED_GREEN_PRIO);
```

Task delay

- Often, a task will block itself by explicitly asking the OS to delay it for some period of time
- `void OSTimeDly(INT16U ticks);`
- Causes a context switch if `ticks` is between 1 and 65535
- If `ticks` is 0, `OSTimeDly()` returns immediately to caller
- On context switch uC/OS-II executes the next highest priority task
- Task that called `OSTimeDly()` will be made ready to run when the specified number of ticks elapses - actually runs when it becomes the highest priority ready task
- Resolution of the delay is between 0 and 1 tick
- Another task can cancel the delay by calling `OSTimeDlyResume()`

Task delay

- `OSTimeDly()` specifies delay in terms of a number of ticks
- Use `OSTimeDlyHMSM()` to specify delay in terms of Hours, Minutes, Seconds and Milliseconds
- Otherwise `OSTimeDlyHMSM()` behaves as `OSTimeDly()`

Complete example – Data declarations

```
#include <stdbool.h>
#include <ucos_ii.h>
#include <mbed.h>

typedef enum {
  LED_RED_PRIO = 4,
  LED_GREEN_PRIO
} taskPriorities_t;

#define  LED_RED_STK_SIZE              256
#define  LED_GREEN_STK_SIZE            256

static OS_STK ledRedStk[LED_RED_STK_SIZE];
static OS_STK ledGreenStk[LED_GREEN_STK_SIZE];

static void appTaskLedRed(void *pdata);
static void appTaskLedGreen(void *pdata);

static void ledToggle(DigitalOut led);
```

Complete example – Main function

```
int main() {

  /* Initialise the OS */
  OSInit();

  /* Create the tasks */
  OSTaskCreate(appTaskLedRed,
               (void *)0,
               (OS_STK *)&ledRedStk[LED_RED_STK_SIZE - 1],
               LED_RED_PRIO);

  OSTaskCreate(appTaskLedGreen,
               (void *)0,
               (OS_STK *)&ledGreenStk[LED_GREEN_STK_SIZE - 1],
               LED_GREEN_PRIO);


  /* Start the OS */
  OSStart();

  /* Should never arrive here */
  return 0;
}
```

Complete example – Tasks

```
static void appTaskLedRed(void *pdata) {
  DigitalOut red(LED_RED);

  /* Start the OS ticker -- must be done in the highest priority task */
  SysTick_Config(SystemCoreClock / OS_TICKS_PER_SEC);

  red = 1;

  /* Task main loop */
  while (true) {
    ledToggle(red);
    OSTimeDlyHMSM(0,0,0,500);
  }
}

static void appTaskLedGreen(void *pdata) {
  DigitalOut green(LED_GREEN);

  green = 0;
  while (true) {
    ledToggle(green);
    OSTimeDlyHMSM(0,0,0,500);
  }
}

static void ledToggle(DigitalOut led) {
  led = 1 - led;
}
```

A problem with preemptive scheduling: Interference

- What is the problem?
  - Interference
  - One or more tasks are prevented from generating a correct result because of interference from another task
  - Sometimes known as a race condition
- Why is it caused?
  - Arbitrary interleaving of task instructions
  - Interleaving caused by the scheduler
- How can it be prevented?
  - Avoid shared variables, or
  - Enforce mutual exclusion of critical sections

How to enforce mutual exclusion of critical sections

- Memory interlock
- Mutual exclusion algorithms: Dekker, Peterson, Lamport
- Disable interrupts
    - `OS_ENTER_CRITICAL()`, `OS_EXIT_CRITICAL()`
    - Use with extreme caution – preferably not at all at the application level
- Semaphores
    - Interrupt latency unaffected
    - Higher priority task runs when ready

Semaphores

Semaphore definition

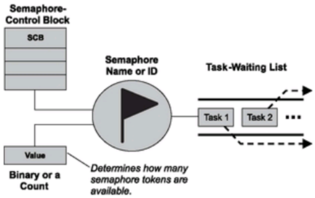
A semaphore is a kernel object that one or more tasks can acquire or release for the purposes of synchronisation or mutual exclusion.

- Binary semaphore proposed by Edsger Dijkstra in 1965 as a mechanism for controlling access to critical sections
- Two operations on semaphores:
  - acquire (aka: pend, wait, take, P)
  - release (aka: post, signal, put, V)

Semaphore operations

- Semaphore value initially 1
- Task calling `acquire(s)` when `s == 1` acquires the semaphore and s becomes 0
- Task calling `acquire(s)` when `s == 0` is suspended
- Task calling `release(s)` makes ready a previously suspended task if there are any
- Task calling `release(s)` restores value of s to 1 if there are no suspended tasks

Counting semaphores (Carel Scholten)



- Idea of binary semaphore can be generalised to counting semaphore (car park example)
- Each `acquire(s)` decreases value of s by 1 down to 0
- Each `release(s)` increases value of s by 1 up to some maximum
- Task waiting list used for tasks waiting on unavailable semaphore
- Waiting list may be FIFO or priority-ordered or ...
  - ... implementation dependent (important to know what your particular implementation does here)

Uses of semaphores

- Semaphores can be used to solve a variety of synchronisation problems:
  - Mutual exclusion
  - Signalling
  - Rendezvous

uC/OS-II semaphores: Create

- Must create a semaphore before using it
  `OS_EVENT *OSSemCreate(INT16U count);`
  - `count` specifies the initial value of the semaphore
  - `OSSemCreate` creates and returns a pointer to an `OS_EVENT` block that the OS uses to store info about the state of the semaphore
- Example

```
OS_EVENT *lcdSem;

...

lcdSem = OSSemCreate(1);
```

uC/OS-II semaphores: Pend

- Acquire the semaphore

```
void OSSemPend(OS_EVENT *pevent,
               INT32U timeout,
               INT8U *perr);
```

  - `pevent` must be a pointer to the `OS_EVENT` representing the semaphore that you want to acquire
  - `timeout` specifies how many ticks to wait before giving up waiting for the semaphore (if `timeout` is 0, then wait as long as it takes)
  - `perr` is a pointer to an integer that the OS can use to tell the caller whether the operation was successful or not

- Example

```
INT8U error;

OSSemPend(lcdSem, 0, &error);
```

uC/OS-II semaphores: Post

- Release the semaphore
  `INT8U OSSemPost(OS_EVENT *pevent);`
  - `pevent` must be a pointer to the `OS_EVENT` representing the semaphore that you want to release
  - the result returned is an integer that the OS can use to tell the caller whether the operation was successful or not
- Example

```
error = OSSemPost(lcdSem);
```

- Suspended tasks are made ready by `OSSemPost` in priority order

Mutual exclusion using semaphores

```
static void appTaskCount1(void *pdata) {
  uint8_t error;

  while (true) {

    OSSemPend(lcdSem, 0, &error);

    count1 += 1;
    display(1, count1);
    total += 1;

    error = OSSemPost(lcdSem);

    if ((count1 + count2) != total) {
      flashing = true;
    }
    OSTimeDlyHMSM(0,0,0,20);
  }
}
```

Mutual exclusion using semaphores

```
static void appTaskCount1(void *pdata) {
  uint8_t error;

  while (true) {

    OSSemPend(lcdSem, 0, &error);

    count1 += 1;
    display(1, count1);
    total += 1;

    error = OSSemPost(lcdSem);

    if ((count1 + count2) != total) {
      flashing = true;
    }
    OSTimeDlyHMSM(0,0,0,20);
  }
}
```

ENTRY PROTOCOL

Mutual exclusion using semaphores

```
static void appTaskCount1(void *pdata) {
  uint8_t error;

  while (true) {

    OSSemPend(lcdSem, 0, &error);

    count1 += 1;
    display(1, count1);
    total += 1;

    error = OSSemPost(lcdSem);

    if ((count1 + count2) != total) {
      flashing = true;
    }
    OSTimeDlyHMSM(0,0,0,20);
  }
}
```

ENTRY PROTOCOL

CRITICAL SECTION

Mutual exclusion using semaphores

```
static void appTaskCount1(void *pdata) {
  uint8_t error;

  while (true) {

    OSSemPend(lcdSem, 0, &error);          ENTRY PROTOCOL

    count1 += 1;                           CRITICAL SECTION
    display(1, count1);
    total += 1;

    error = OSSemPost(lcdSem);             EXIT PROTOCOL

    if ((count1 + count2) != total) {
      flashing = true;
    }
    OSTimeDlyHMSM(0,0,0,20);
  }
}
```

Acknowledgements

- Cooling, N., Mutex vs Semaphores Parts 1 and 2, Sticky Bits Blog, 2009
- Labrosse, J., MicroC/OS-II: The Real-time Kernel, CMP, 2002
- Li, Q. and Yao, C., Real-time concepts for embedded systems, CMP, 2003