Embedded systems engineering

David Kendall

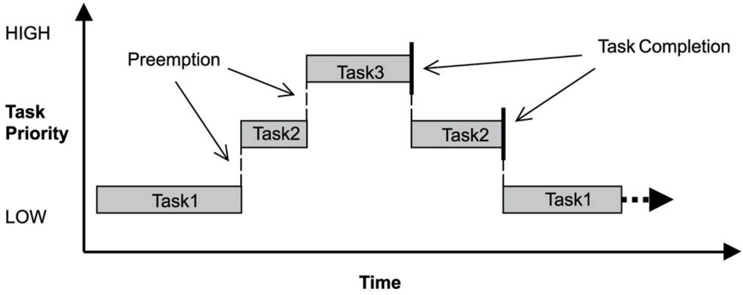
Introduction

- Preemptive scheduling
- Review of μC/OS-II (uC/OS-II)
  - Task management
  - Delay
  - Semaphores; Priority inversion; Mutexes

The move to pre-emptive scheduling

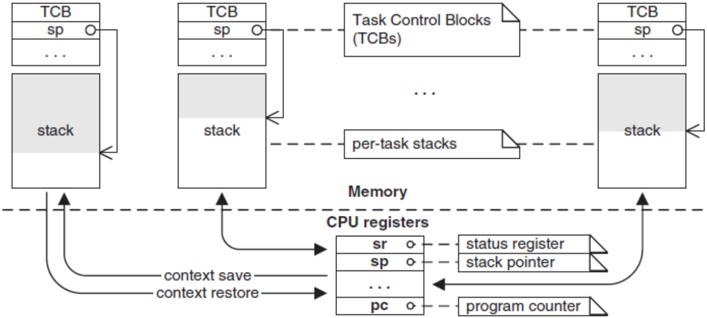
- Cooperative scheduling (cyclic executive, time-triggered)
  - Tasks voluntarily yield the CPU and signal the next task to begin
  - Can be a challenge to accommodate long-running tasks while maintaining the responsiveness of the system.
- Pre-emptive scheduling attempts to address this challenge.
- Pre-emptive scheduling
  - Task is forced to yield the CPU
  - Main approaches:
    - Round robin
    - Priority-based

Fixed-priority preemptive scheduling



- We focus on fixed-priority pre-emptive scheduling

Inside a preemptive scheduler



(Samek, 2008, p.259)

uC/OS-II: A small operating system

- Main features:
  - Multi-tasking
  - Preemptive
- Other features:
  - 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

Tasks 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 appTaskConnectLed(void *pdata) {
  while (true) {
    OSTimeDlyHMSM(0,0,0,500);
    ledToggle(USB_CONNECT_LED);
  }
}
```

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 $\Rightarrow$ high priority
- high number $\Rightarrow$ 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 and then go up in steps of 4 for the remaining tasks (up to 28 for our OS configuration)
- Example

```
#define    APP_TASK_CONNECT_PRIO       8
```

Tasks: other requirements

Stack

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

```
#define   APP_TASK_CONNECT_STK_SIZE                    256
static  OS_STK  appTaskConnectStk[APP_TASK_CONNECT_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,               /* any data for the task function */
        OS_STK *ptos,              /* pointer to top of stack */
        INT8U priority             /* task priority */
);
```

Example

```
#define APP_TASK_CONNECT_PRIO            8
#define APP_TASK_CONNECT_STK_SIZE      256

static OS_STK appTaskConnectStk[APP_TASK_CONNECT_STK_SIZE];

OSTaskCreate(appTaskConnectLed,
             (void *)0,
             (OS_STK *)&appTaskConnectStk[APP_TASK_CONNECT_STK_SIZE - 1],
             APP_TASK_CONNECT_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

```
#include <stdbool.h>
#include <ucos_ii.h>
#include <osutils.h>
#include <bsp.h>
#include <leds.h>

/*
*******************************************************************************
*                         APPLICATION TASK PRIORITIES
*******************************************************************************
*/

#define  APP_TASK_LINK_PRIO                        4
#define  APP_TASK_CONNECT_PRIO                     8

/*
*******************************************************************************
*                         APPLICATION TASK STACKS
*******************************************************************************
*/

#define  APP_TASK_LINK_STK_SIZE                  256
#define  APP_TASK_CONNECT_STK_SIZE               256

static OS_STK appTaskLinkStk[APP_TASK_LINK_STK_SIZE];
static OS_STK appTaskConnectStk[APP_TASK_CONNECT_STK_SIZE];
```

Complete example

```
/*
*******************************************************************************
*                          APPLICATION FUNCTION PROTOTYPES
*******************************************************************************
*/

static void appTaskLinkLed(void *pdata);
static void appTaskConnectLed(void *pdata);

/*
*******************************************************************************
*                          GLOBAL FUNCTION DEFINITIONS
*******************************************************************************
*/
int main() {

  /* Initialise the board support package and the OS */
  bspInit();
  OSInit();

  /* Create the tasks */
  OSTaskCreate(appTaskLinkLed,
               (void *)0,
               (OS_STK *)&appTaskLinkStk[APP_TASK_LINK_STK_SIZE - 1],
               APP_TASK_LINK_PRIO);

  OSTaskCreate(appTaskConnectLed,
               (void *)0,
               (OS_STK *)&appTaskConnectStk[APP_TASK_CONNECT_STK_SIZE - 1],
               APP_TASK_CONNECT_PRIO);
```

Complete example

```
  /* Start the OS */
  OSStart();

  /* Should never arrive here */
  return 0;
}
/*
******************************************************************************
*                          APPLICATION TASK DEFINITIONS
******************************************************************************
*/
static void appTaskLinkLed(void *pdata) {
  /* Start the OS ticker -- must be done in the highest priority task */
  osStartTick();

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

static void appTaskConnectLed(void *pdata) {
  while (true) {
    OSTimeDlyHMSM(0,0,0,500);
    ledToggle(USB_CONNECT_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
  - created by the scheduler
    - round-robin – problems?
    - priority preemptive – problems?
- 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, Monitors
  - Interrupt latency unaffected
  - Higher priority task runs when ready

Semaphores

Semaphore definition

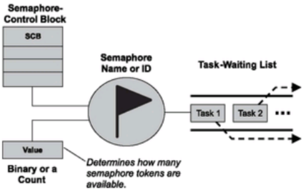
A semaphore is a kernel object that one or more tasks can acquire or release for the purposes of synchronisation or mutual exclusion.

- 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);
  }
}
```

ENTRY PROTOCOL

CRITICAL SECTION

EXIT PROTOCOL

(See mutexsem.c)

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

EXIT PROTOCOL

(See mutexsem.c)

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

EXIT PROTOCOL

(See mutexsem.c)

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

EXIT PROTOCOL

(See mutexsem.c)

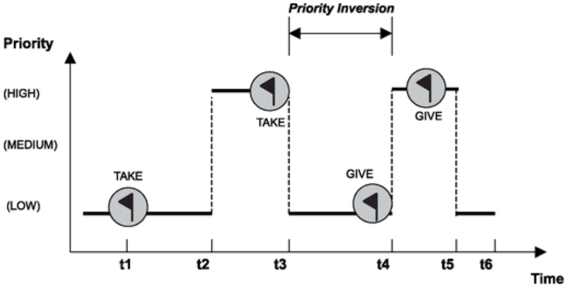
Problems with the use of semaphores

- Accidental release
- Recursive deadlock
- Task-death deadlock
- Priority inversion
- Semaphore as a signal
- For details see: Cooling, N, *Mutex vs Semaphores* Parts 1 and 2, Sticky Bits Blog, 2009

Priority Inversion

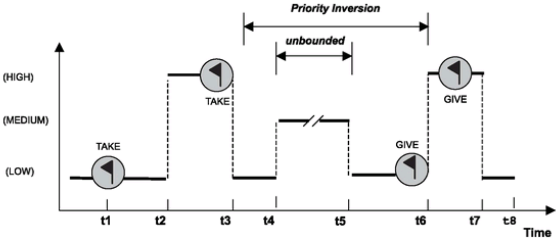
- Fixed priority preemptive OS with blocking on access to shared resources can suffer priority inversion.
- Low priority task is allowed to execute while higher priority task is blocked.
- Look at priority inversion in more detail
- Consider possible solution to the priority inversion problem

Bounded priority inversion



- At time t_1 low priority (LP) task acquires semaphore
- At time t_2 high priority (HP) task preempts LP
- At time t_3 HP tries to acquire semaphore and is blocked
- At time t_4 LP returns semaphore, HP acquires semaphore, is unblocked and runs
- At time t_5 HP finishes and LP runs again

Unbounded priority inversion

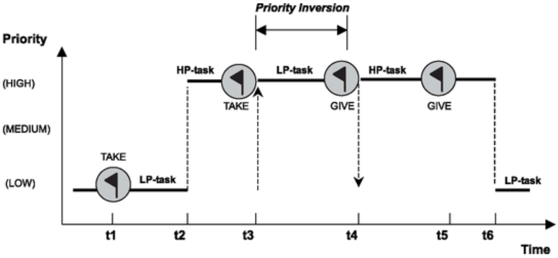


- Priority inversion again occurs at time t_3
- At time t_4 LP is preempted by a medium priority task (MP) that is able to run because it does not need to acquire the locked semaphore
- MP runs until completion at time t_5
- Duration of period from t_4 to t_5 is very difficult to predict (unbounded)

Problems caused by priority inversion

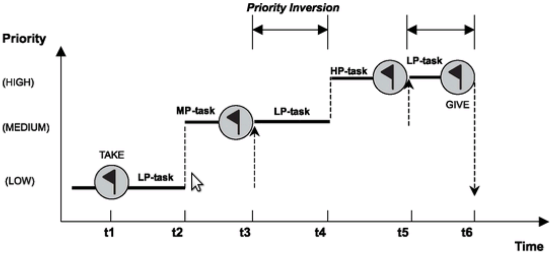
- Task completion times vary more widely
- e.g. MP completes earlier in the priority inversion case than in the cases when priority inversion does not occur:
  - HP runs first, acquires semaphore, runs to completion, MP runs, ...
  - MP runs first, preempted by HP, ...
- Task completion time of HP is delayed in the priority inversion case – may miss deadline

Avoiding priority inversion: Priority Inheritance Protocol (PIP)



- Consider a task T trying to acquire a resource R
- If R is in use, T is blocked
- If R is free, R is allocated to T
- When a task of higher priority attempts to access R, priority of T is raised to priority of higher priority task
- When it releases R, priority of T is set to maximum of its original priority and priorities of any tasks it's still blocking by holding other resources

Transitive priority promotion in PIP



- PIP is *dynamic* – a task does not have its priority raised until a higher priority task tries to acquire a resource that it holds
- Priority continues to rise as other higher priority tasks try to acquire its resource. . . and falls again as it releases resources

Pros and cons of PIP

- Pros
  - All priority inversions are bounded when PIP is used
- Cons
  - Frequent changes of priority may become a significant overhead
  - Deadlock is possible – MP acquires some resources needed by HP, HP acquires some resources needed by MP, when LP releases resource, HP runs to deadlock

Priority ceiling protocols

- Two main versions of priority ceiling protocol
  - Original Ceiling Priority Protocol (OCPP)
  - Immediate Ceiling Priority Protocol (ICPP)
- Both intended to reduce the number of priority changes required and to prevent deadlock

Immediate ceiling priority protocol (ICPP)

- Each task has a static default priority
- Each resource has a static ceiling value that is the maximum priority of the tasks that use it
- A task also has a dynamic priority that is the maximum of its own static priority and the ceiling values of any resources it has locked

Properties of ICPP

- No deadlock
- A task can be blocked only at the very beginning of its execution
    - Once a task starts executing, all the resources it needs must be free
    - If they were not, some task would have an equal or higher priority and the task's execution would be postponed

Mutex: fixing the semaphore?

- Mutex introduces the principle of ownership
  - a mutex can be released only by the task that acquired it
  - a task that tries to release a mutex that it didn't acquire causes an error and the mutex remains locked
- accidental release much more difficult
  - signalling not allowed
- Depending on implementation, additional features can be supported:
  - recursive locking can be allowed - mutex must be released as many times as it has been acquired
  - task-death deadlock can be recovered by recognising that mutex is owned by task that no longer exists and released
  - priority inversion can be reduced using e.g. priority ceiling protocol

Mutexes in uC/OS-II

- Principle of ownership enforced
  - Mutex can be released only by the task that acquired it
- Recursive locking not supported
- Task-death deadlock not detected/recovered
- Priority inversion tackled by a hybrid ceiling priority protocol
  - Each mutex must be given a priority that is greater than that of any task that uses it (the ceiling value).
  - If a task holding a mutex blocks a higher priority task then its priority is raised to the ceiling of the resource that causes the blocking
  - Properties:
    - Bounded priority inversion ...
    - ... but deadlock is not prevented

uC/OS-II mutexes: Create

- Create mutex

  ```
  OS_EVENT *OSMutexCreate(INT8U prio, INT8U *osSta
  ```

- Mutexes must be created before they are used.
- `prio` is the priority ceiling priority (PCP), ie `prio` must be a higher priority than that of any of the tasks that will attempt to acquire the mutex
- `prio` must not be used already
- `osStatus` is a pointer to a variable that is used to hold a status code: ok, no event control blocks, problem with priority etc.

uC/OS-II mutexes: Create

- Create mutex

  ```
  OS_EVENT *OSMutexCreate(INT8U prio, INT8U *osSta
  ```

- Mutexes must be created before they are used.
- `prio` is the priority ceiling priority (PCP), ie `prio` must be a higher priority than that of any of the tasks that will attempt to acquire the mutex
- `prio` must not be used already
- `osStatus` is a pointer to a variable that is used to hold a status code: ok, no event control blocks, problem with priority etc.

uC/OS-II mutexes: Create

- Create mutex

  OS_EVENT *OSMutexCreate(INT8U prio, INT8U *osSta

- Mutexes must be created before they are used.
- `prio` is the priority ceiling priority (PCP), ie `prio` must be a higher priority than that of any of the tasks that will attempt to acquire the mutex
- `prio` must not be used already
- `osStatus` is a pointer to a variable that is used to hold a status code: ok, no event control blocks, problem with priority etc.

uC/OS-II mutexes: Create

- Create mutex

```
OS_EVENT *OSMutexCreate(INT8U prio, INT8U *osSta
```

- Mutexes must be created before they are used.
- `prio` is the priority ceiling priority (PCP), ie `prio` must be a higher priority than that of any of the tasks that will attempt to acquire the mutex
- `prio` must not be used already
- `osStatus` is a pointer to a variable that is used to hold a status code: ok, no event control blocks, problem with priority etc.

uC/OS-II mutexes: Create

- Create mutex

  ```
  OS_EVENT *OSMutexCreate(INT8U prio, INT8U *osSta
  ```

- Mutexes must be created before they are used.
- `prio` is the priority ceiling priority (PCP), ie `prio` must be a higher priority than that of any of the tasks that will attempt to acquire the mutex
- `prio` must not be used already
- `osStatus` is a pointer to a variable that is used to hold a status code: ok, no event control blocks, problem with priority etc.

uC/OS-II mutexes: Pend

- pend, acquire, wait, P(s)

```
void OSMutexPend(OS_EVENT *pevent,
                 INT32U timeout,
                 INT8U *osStatus);
```

- `pevent` is a pointer to the mutex
- `timeout` is used to allow calling task to resume if mutex is not posted within `timeout` ticks. If `timeout` is 0, task will wait for as long as it takes.
- `osStatus` is a pointer to a variable that is used to hold a status code: ok, timeout occured, not a mutex, priority problem etc.
- You should not call `OSMutexPend()` from an ISR
- The priority of a task that tries to pend on a mutex must be lower than the PCP of the mutex
- You should not call any OS function that could cause the task that owns the mutex to be suspended

uC/OS-II mutexes: Pend

- pend, acquire, wait, P(s)

```
void OSMutexPend(OS_EVENT *pevent,
                 INT32U timeout,
                 INT8U *osStatus);
```

- `pevent` is a pointer to the mutex
- `timeout` is used to allow calling task to resume if mutex is not posted within `timeout` ticks. If `timeout` is 0, task will wait for as long as it takes.
- `osStatus` is a pointer to a variable that is used to hold a status code: ok, timeout occured, not a mutex, priority problem etc.
- You should not call `OSMutexPend()` from an ISR
- The priority of a task that tries to pend on a mutex must be lower than the PCP of the mutex
- You should not call any OS function that could cause the task that owns the mutex to be suspended – hurry up and

uC/OS-II mutexes: Pend

- pend, acquire, wait, P(s)

```
void OSMutexPend(OS_EVENT *pevent,
                 INT32U timeout,
                 INT8U *osStatus);
```

- `pevent` is a pointer to the mutex
- `timeout` is used to allow calling task to resume if mutex is not posted within `timeout` ticks. If `timeout` is 0, task will wait for as long as it takes.
- `osStatus` is a pointer to a variable that is used to hold a status code: ok, timeout occured, not a mutex, priority problem etc.
- You should not call `OSMutexPend()` from an ISR
- The priority of a task that tries to pend on a mutex must be lower than the PCP of the mutex
- You should not call any OS function that could cause the task that owns the mutex to be suspended – hurry up and

uC/OS-II mutexes: Pend

- pend, acquire, wait, P(s)

```
void OSMutexPend(OS_EVENT *pevent,
                 INT32U timeout,
                 INT8U *osStatus);
```

- `pevent` is a pointer to the mutex
- `timeout` is used to allow calling task to resume if mutex is not posted within `timeout` ticks. If `timeout` is 0, task will wait for as long as it takes.
- `osStatus` is a pointer to a variable that is used to hold a status code: ok, timeout occured, not a mutex, priority problem etc.
- You should not call `OSMutexPend()` from an ISR
- The priority of a task that tries to pend on a mutex must be lower than the PCP of the mutex
- You should not call any OS function that could cause the task that owns the mutex to be suspended – hurry up and

uC/OS-II mutexes: Pend

- pend, acquire, wait, P(s)

```
void OSMutexPend(OS_EVENT *pevent,
                 INT32U timeout,
                 INT8U *osStatus);
```

- `pevent` is a pointer to the mutex
- `timeout` is used to allow calling task to resume if mutex is not posted within `timeout` ticks. If `timeout` is 0, task will wait for as long as it takes.
- `osStatus` is a pointer to a variable that is used to hold a status code: ok, timeout occured, not a mutex, priority problem etc.
- You should not call `OSMutexPend()` from an ISR
- The priority of a task that tries to pend on a mutex must be lower than the PCP of the mutex
- You should not call any OS function that could cause the task that owns the mutex to be suspended – hurry up and

uC/OS-II mutexes: Pend

- pend, acquire, wait, P(s)

```
void OSMutexPend(OS_EVENT *pevent,
                 INT32U timeout,
                 INT8U *osStatus);
```

- `pevent` is a pointer to the mutex
- `timeout` is used to allow calling task to resume if mutex is not posted within `timeout` ticks. If `timeout` is 0, task will wait for as long as it takes.
- `osStatus` is a pointer to a variable that is used to hold a status code: ok, timeout occured, not a mutex, priority problem etc.
- You should not call `OSMutexPend()` from an ISR
- The priority of a task that tries to pend on a mutex must be lower than the PCP of the mutex
- You should not call any OS function that could cause the task that owns the mutex to be suspended – hurry up and

uC/OS-II mutexes: Pend

- pend, acquire, wait, P(s)

```
void OSMutexPend(OS_EVENT *pevent,
                 INT32U timeout,
                 INT8U *osStatus);
```

- `pevent` is a pointer to the mutex
- `timeout` is used to allow calling task to resume if mutex is not posted within `timeout` ticks. If `timeout` is 0, task will wait for as long as it takes.
- `osStatus` is a pointer to a variable that is used to hold a status code: ok, timeout occured, not a mutex, priority problem etc.
- You should not call `OSMutexPend()` from an ISR
- The priority of a task that tries to pend on a mutex must be lower than the PCP of the mutex
- You should not call any OS function that could cause the task that owns the mutex to be suspended – hurry up and

uC/OS-II mutexes: Pend

- The behaviour of `OSMutexPend()` is quite sophisticated . . .
  - . . . it has features of both the priority inheritance protocol and the priority ceiling protocol described earlier
- If a task calls pend when the mutex is available, it becomes the owner of the mutex and is allowed to proceed. The mutex is then no longer available.
- If a task calls pend when the mutex is not available, the task is suspended.
- If the task that is suspended has a higher priority than that of the task that owns the mutex, the priority of the owning task is raised to the ceiling value of the mutex.
  - Notice that this can happen at most once
  - The task will run at this new priority when it is ready
  - Its priority will be restored to its original priority when it releases the mutex

uC/OS-II mutexes: Pend

- The behaviour of `OSMutexPend()` is quite sophisticated . . .
  - . . . it has features of both the priority inheritance protocol and the priority ceiling protocol described earlier
- If a task calls pend when the mutex is available, it becomes the owner of the mutex and is allowed to proceed. The mutex is then no longer available.
- If a task calls pend when the mutex is not available, the task is suspended.
- If the task that is suspended has a higher priority than that of the task that owns the mutex, the priority of the owning task is raised to the ceiling value of the mutex.
  - Notice that this can happen at most once
  - The task will run at this new priority when it is ready
  - Its priority will be restored to its original priority when it releases the mutex

uC/OS-II mutexes: Pend

- The behaviour of `OSMutexPend()` is quite sophisticated . . .
  - . . . it has features of both the priority inheritance protocol and the priority ceiling protocol described earlier
- If a task calls pend when the mutex is available, it becomes the owner of the mutex and is allowed to proceed. The mutex is then no longer available.
- If a task calls pend when the mutex is not available, the task is suspended.
- If the task that is suspended has a higher priority than that of the task that owns the mutex, the priority of the owning task is raised to the ceiling value of the mutex.
  - Notice that this can happen at most once
  - The task will run at this new priority when it is ready
  - Its priority will be restored to its original priority when it releases the mutex

uC/OS-II mutexes: Pend

- The behaviour of `OSMutexPend()` is quite sophisticated ...
  - ...it has features of both the priority inheritance protocol and the priority ceiling protocol described earlier
- If a task calls pend when the mutex is available, it becomes the owner of the mutex and is allowed to proceed. The mutex is then no longer available.
- If a task calls pend when the mutex is not available, the task is suspended.
- If the task that is suspended has a higher priority than that of the task that owns the mutex, the priority of the owning task is raised to the ceiling value of the mutex.
  - Notice that this can happen at most once
  - The task will run at this new priority when it is ready
  - Its priority will be restored to its original priority when it releases the mutex

uC/OS-II mutexes: Pend

- The behaviour of `OSMutexPend()` is quite sophisticated ...
  - ...it has features of both the priority inheritance protocol and the priority ceiling protocol described earlier
- If a task calls pend when the mutex is available, it becomes the owner of the mutex and is allowed to proceed. The mutex is then no longer available.
- If a task calls pend when the mutex is not available, the task is suspended.
- If the task that is suspended has a higher priority than that of the task that owns the mutex, the priority of the owning task is raised to the ceiling value of the mutex.
  - Notice that this can happen at most once
  - The task will run at this new priority when it is ready
  - Its priority will be restored to its original priority when it releases the mutex

uC/OS-II mutexes: Pend

- The behaviour of `OSMutexPend()` is quite sophisticated . . .
  - . . . it has features of both the priority inheritance protocol and the priority ceiling protocol described earlier
- If a task calls pend when the mutex is available, it becomes the owner of the mutex and is allowed to proceed. The mutex is then no longer available.
- If a task calls pend when the mutex is not available, the task is suspended.
- If the task that is suspended has a higher priority than that of the task that owns the mutex, the priority of the owning task is raised to the ceiling value of the mutex.
  - Notice that this can happen at most once
  - The task will run at this new priority when it is ready
  - Its priority will be restored to its original priority when it releases the mutex

uC/OS-II mutexes: Pend

- The behaviour of `OSMutexPend()` is quite sophisticated …
  - …it has features of both the priority inheritance protocol and the priority ceiling protocol described earlier
- If a task calls pend when the mutex is available, it becomes the owner of the mutex and is allowed to proceed. The mutex is then no longer available.
- If a task calls pend when the mutex is not available, the task is suspended.
- If the task that is suspended has a higher priority than that of the task that owns the mutex, the priority of the owning task is raised to the ceiling value of the mutex.
  - Notice that this can happen at most once
  - The task will run at this new priority when it is ready
  - Its priority will be restored to its original priority when it releases the mutex

uC/OS-II mutexes: Post

- post, release, signal, V(s)

  ```
  INT8U OSMutexPost(OS_EVENT *pevent)
  ```

- Should be called only by the task that holds the mutex
- If priority of owning task has been raised when a higher priority task attempted to acquire mutex, original task priority is restored.
- If one or more tasks are waiting for the mutex, the mutex is given to the highest priority task waiting on the mutex. The scheduler is then called to determine if the awakened task is now the highest priority task ready to run, and if so, a context switch is done to run the readied task.
- If no task is waiting for the mutex, the mutex value is simply set to available (0xFF).

uC/OS-II mutexes: Post

- post, release, signal, V(s)

  ```
  INT8U OSMutexPost(OS_EVENT *pevent)
  ```

- Should be called only by the task that holds the mutex
- If priority of owning task has been raised when a higher priority task attempted to acquire mutex, original task priority is restored.
- If one or more tasks are waiting for the mutex, the mutex is given to the highest priority task waiting on the mutex. The scheduler is then called to determine if the awakened task is now the highest priority task ready to run, and if so, a context switch is done to run the readied task.
- If no task is waiting for the mutex, the mutex value is simply set to available (0xFF).

uC/OS-II mutexes: Post

- post, release, signal, V(s)

```
INT8U OSMutexPost(OS_EVENT *pevent)
```

- Should be called only by the task that holds the mutex
- If priority of owning task has been raised when a higher priority task attempted to acquire mutex, original task priority is restored.
- If one or more tasks are waiting for the mutex, the mutex is given to the highest priority task waiting on the mutex. The scheduler is then called to determine if the awakened task is now the highest priority task ready to run, and if so, a context switch is done to run the readied task.
- If no task is waiting for the mutex, the mutex value is simply set to available (0xFF).

uC/OS-II mutexes: Post

- post, release, signal, V(s)

  ```
  INT8U OSMutexPost(OS_EVENT *pevent)
  ```

- Should be called only by the task that holds the mutex
- If priority of owning task has been raised when a higher priority task attempted to acquire mutex, original task priority is restored.
- If one or more tasks are waiting for the mutex, the mutex is given to the highest priority task waiting on the mutex. The scheduler is then called to determine if the awakened task is now the highest priority task ready to run, and if so, a context switch is done to run the readied task.
- If no task is waiting for the mutex, the mutex value is simply set to available (0xFF).

uC/OS-II mutexes: Post

- post, release, signal, V(s)

```
INT8U OSMutexPost(OS_EVENT *pevent)
```

- Should be called only by the task that holds the mutex
- If priority of owning task has been raised when a higher priority task attempted to acquire mutex, original task priority is restored.
- If one or more tasks are waiting for the mutex, the mutex is given to the highest priority task waiting on the mutex. The scheduler is then called to determine if the awakened task is now the highest priority task ready to run, and if so, a context switch is done to run the readied task.
- If no task is waiting for the mutex, the mutex value is simply set to available (0xFF).

Acknowledgements

- Cooling, N., Mutex vs Semaphores Parts 1 and 2, Sticky Bits Blog, 2009
- Kalinsky, David and Michael Barr. "Priority Inversion," Embedded Systems Programming, April 2002, pp. 55-56.
- Labrosse, J., MicroC/OS-II: The Real-time Kernel, CMP, 2002
- Li, Q. and Yao, C., Real-time concepts for embedded systems, CMP, 2003
- Sha, L. Rajkumar, R. and Lehoczky, J. *Priority Inheritance Protocols: An Approach to to Real-Time Synchronization*. IEEE Transactions on Computers, 39 (9): 1175–1185; September, 1990