Operating systems and concurrency (B09)

David Kendall

Northumbria University

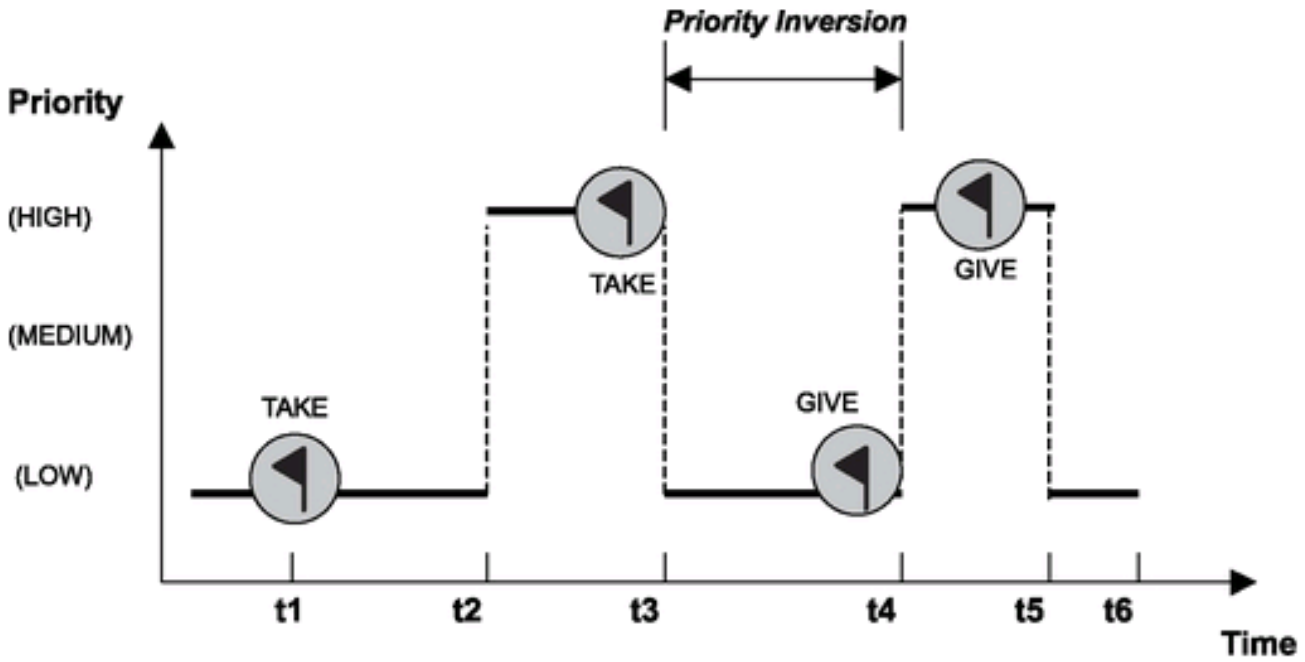
Problems with the use of semaphores

- 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

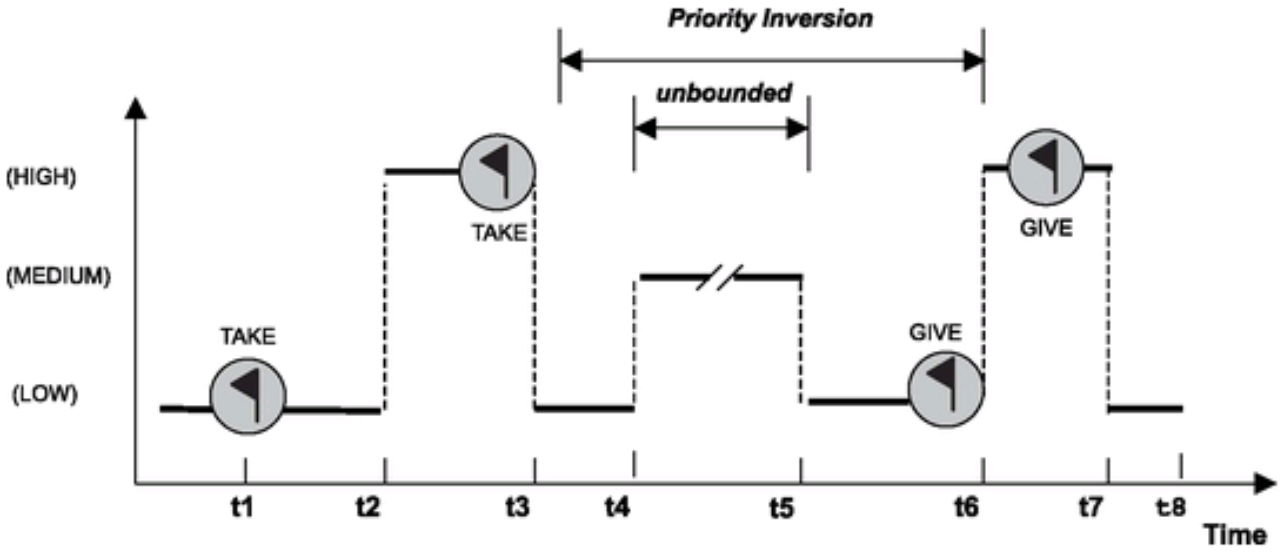
- Fixed priority preemptive OS with blocking on access to shared resources can suffer priority inversion.
- Low priority task is allowed to execute while higher priority task is blocked.
- Look at priority inversion in more detail
- Consider possible solution to the priority inversion problem

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

Solar system's most famous priority inversion

Mars Pathfinder (1997)

- Mars Pathfinder mission to Mars in 1997 almost failed due to a software problem
- NASA engineers attempted to debug the software over a radio link to Mars
- Smallest recorded distance between Earth and Mars is $5.6 \cdot 10^{10}$ m
- Assuming radio waves travel through space at the speed of light, the smallest propagation delay of a message from Earth to Mars is over 3 minutes

Solar system's most famous priority inversion

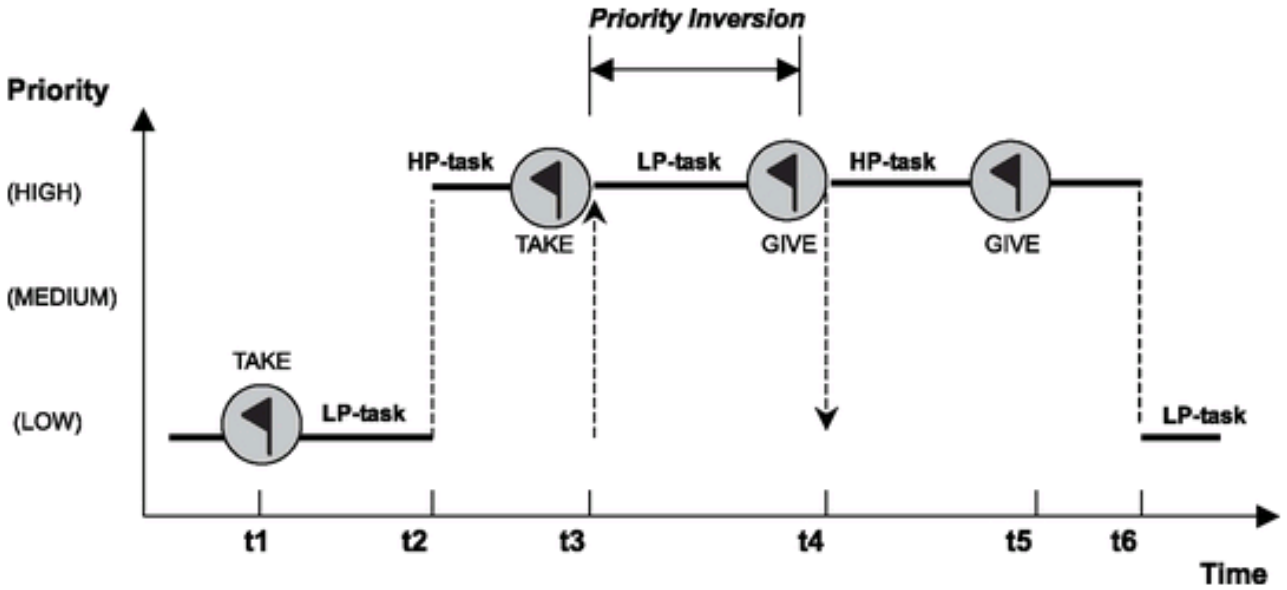
Mars Pathfinder (1997)

- Communicating at these speeds, they managed to work out that the problem was caused by priority inversion
- They transmitted a software upgrade that turned on a priority inheritance protocol in the VxWorks OS.
- The mission continued successfully

What happened on Mars?

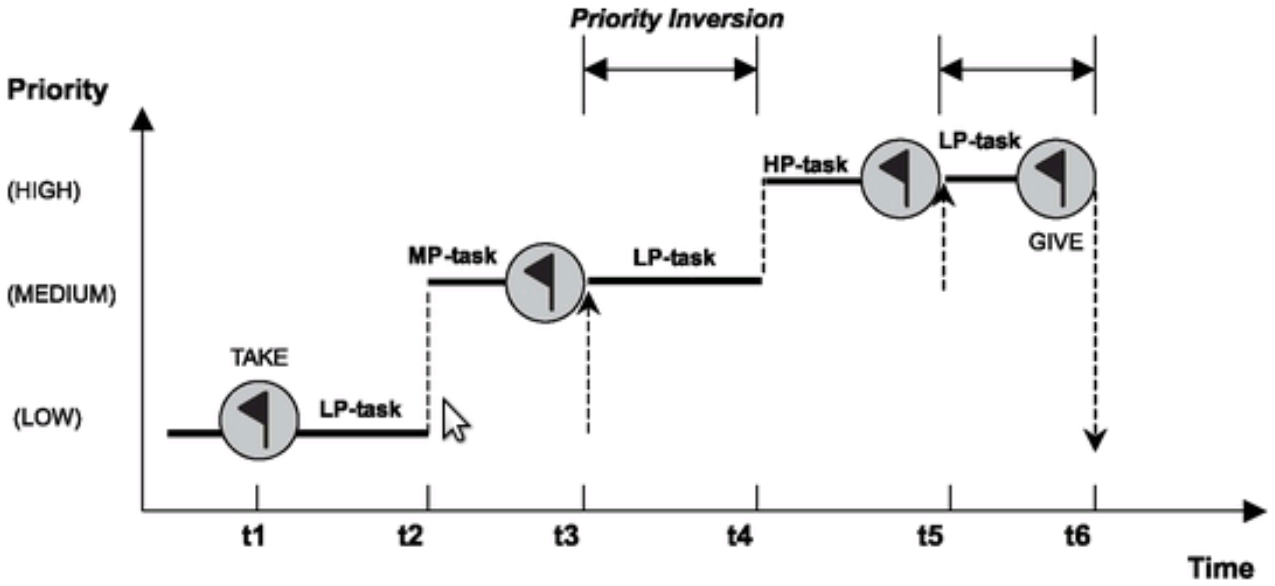
- Various devices communicated over a data bus.
- Activity on bus managed by pair of high-priority tasks, one of which communicated through pipe with low-priority meteorological science task.
- The meteorological task was preempted by medium-priority tasks while it held a mutex related to the pipe.
- While low-priority task preempted, the high-priority bus distribution manager tried to send more data to it over the same pipe.
- Mutex still held by meteorological task, so bus distribution manager made to wait.
- When other bus scheduler active, it noticed that the distribution manager hadn't completed its work for that bus cycle and forced a system reset.

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

Original ceiling priority protocol (OCPP)

- 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 any priority it inherits by blocking higher priority tasks
- A task T can only lock a resource if its dynamic priority is higher than the ceiling of any resources currently locked by other tasks
- If the dynamic priority of T is not high enough, the task is blocked on the resource with the highest priority ceiling of all the resources currently locked by other tasks
- The task holding the resource that is blocking T has its priority raised to the priority of T

Properties of OCPP

- No deadlock
- Each task can be delayed at most once by a lower priority task
- The delay is a function of the time taken by the lower priority task to complete its critical section
- So, the duration of priority inversion is bounded and . . .
- . . . the task has its priority adjusted at most once

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

OCPP v ICPP

- ICPP is easier to implement than the original (OCPP) as blocking relationships need not be monitored
- ICPP leads to fewer context switches as all blocking occurs prior to first execution
- ICPP requires more priority changes as this happens with all resource usage . . .
- . . . whereas OCPP causes a priority change only if blocking actually occurs
- Note: ICPP is also called
  - Priority Protect Protocol (in POSIX)
  - Priority Ceiling Emulation (in Real-Time Java)

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 *osStatus);
```

uC/OS-II mutexes: Create

- Create mutex

```
OS_EVENT *OSMutexCreate(INT8U prio, INT8U *osStatus);
```

- Mutexes must be created before they are used.

uC/OS-II mutexes: Create

- Create mutex

```
OS_EVENT *OSMutexCreate(INT8U prio, INT8U *osStatus);
```

- 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

uC/OS-II mutexes: Create

- Create mutex

```
OS_EVENT *OSMutexCreate(INT8U prio, INT8U *osStatus);
```

- 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

uC/OS-II mutexes: Create

- Create mutex

```
OS_EVENT *OSMutexCreate(INT8U prio, INT8U *osStatus);
```

- 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);
```

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

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.

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.

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

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

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 release the resource as quickly as possible

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

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.

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.

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.

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

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

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)
```

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

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.

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.

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