NAME

rwlock – readers writer locks, **rwcreate, rwdelete, rdlock, wrlock, rdunlock, wrunlock**

SYNOPSIS

#include <rwlock.h>

rwlock_t *rwcreate(int *flags***);**

STATUS rwdelete(rwlock_t ***rwp*);**

STATUS rdlock(rwlock_t ***rwp*, int** *timeout***);**

STATUS wrlock(rwlock_t ***rwp*, int** *timeout***);**

STATUS rdunlock(rwlock_t ***rwp*);**

STATUS wrunlock(rwlock_t ***rwp*);**

DESCRIPTION

Readers writer locks are used for synchronization of critical sections of code. They are frequently used in scenarios where a data structure is frequently search but only occasionally modified. Readers writer locks, as the name implies, allow many concurrent readers or a single writer access to critical sections.

To create a **rwlock** (readers writer lock) use **rwcreate()**. **rwcreate()** allocates memory from the default system memory partition for the rwlock_t and returns a pointer to it. The value of *flags* can be the bitwise OR of the following values.

RW_WRQ_FIFO:

This is the default if neither **RW_WRQ_FIFO** nor **RW_WRQ_PRIO** is set in *flags*. The default behavior ensures that writers will not starve. When **RW_WRQ_FIFO** is set then tasks that block while taking a write lock are queued in first in first out order.

RW_WRQ_PRIO:

When **RW_WRQ_PRIO** is set then tasks that come off of the waiting writers queue do so in priority order. This allows for the highest priority waiting writer to run prior to lower priority writers that are also waiting. Writer starvation is possible when **RW_WRQ_PRIO** is set in the *flags* argument.

After **rwcreate()** returns the readers writer lock is ready to be used. Do not call **rwcreate()** multiple times for the same **rwlock**. Be sure to initialize any lock before using it with a call to **rwcreate()**. Make sure that no task calls any of the other readers writer lock functions until the lock has been successfully initialized.

To destroy a readers writer lock use **rwdelete()**. The **rwdelete()** function frees the memory allocated for the **rwlock** by **rwcreate()**. Make sure that no other task calls any of the other readers writers lock functions while the lock is being destroyed. Make sure that no task calls any of the readers writers lock functions on *rwp* after the lock has been destroyed.

To acquire a read lock to the **rwlock** pointed to by *rwp*, use **rdlock()**. If the **rwlock** is currently locked for writing by any other task or there are tasks waiting to acquire a write lock, the read lock will not be granted at that time. In such a case, the task will either block or return immediately with an error depending upon the value of *timeout* specified. Otherwise, the task will be granted a read lock. At any moment numerous tasks may hold a read lock to any **rwlock**. At no time will a task hold a read lock to a **rwlock** while another task holds a write lock to the **rwlock**. It is an error for a task to call **rdlock()** on a **rwlock** that it currently holds a read or write lock on.

To acquire a write lock to the **rwlock** pointed to by *rwp*, use **wrlock()**. If the **rwlock** is currently locked for writing by any other task or there is at least one task holding a read lock to the **rwlock**, the write lock will not be granted at that time. In such a case, the task will either block or return immediately with an error depending upon the value of *timeout* specified. Otherwise, the task will be granted a write lock. At any moment only one task at a time may hold a write lock to the **rwlock**. At no time will a task hold a write lock to a **rwlock** while any other task holds a read lock to the **rwlock**. It is an error for a task to call **wrlock()** on a **rwlock** that it currently holds a read or write lock on. If the write lock cannot be acquired immediately and the task blocks, the order in which tasks are awakened when the write lock becomes available depends on the whether **RW_WRQ_FIFO** or **RW_WRQ_PRIO** was specified in *flags* when the **rwlock** was created.

A *timeout* can be used with **rdlock()** and **wrlock()**. If the lock cannot be acquired within the requested *timeout* the function will return and indicate the appropriate error. The *timeout* is specified in units of system clock ticks. Use **sysClkRateGet()** to avoid making assumptions about the clock rate. For example, setting the *timeout* to **sysClkRateGet()** will have the task block for a maximum of one second waiting to acquire the lock. A *timeout* value of **WAIT_FOREVER** can be used to have the calling task block indefinitely until the lock becomes available. A *timeout* of **NO_WAIT** can be used to have the function never block and return immediately with an error if the lock cannot be immediately acquired.

To release a read lock to a **rwlock** acquired from a call to **rdlock()** call **rdunlock()**. It is an error to call **rdunlock()** if the calling task does not currently hold a read lock to the **rwlock**. In particular it is an error to call **rdunlock()** if the **rwlock** is currently write locked by the calling task or is not read locked by the calling task, even if another task holds a read lock to the **rwlock**.

To release a write lock to a **rwlock** acquired from a call to **wrlock()** call **wrunlock()**. It is an error to call **wrunlock()** if the calling task does not currently hold a write lock to the **rwlock**. In particular it is an error to call **wrunlock()** if the **rwlock** is currently read locked by the calling task or is not write locked by the calling task, even if another task holds a write lock to the **rwlock**.

USAGE

You need to load the readers writer lock library into a VxWorks node before using any of the **rwlock** functions documented here. The library can be loaded from your target specific module directory on fecode−bd. An example target is PPC604. You can load either a specific version by specifying a version number or the most recent version by omitting a version number. For example to load version 1.0 of the readers writer lock library on a PPC604 node:

ld < vxworks_boot/module/PPC604/rwlock-1.0.out

RETURN VALUES

If **rwcreate()** is successful it returns a pointer to an initialized readers writer lock. Otherwise it returns **NULL**.

For all the other functions in the **rwlock** library **OK** is returned to indicate success and **ERROR** for failure. When a function fails, errno is set to an appropriate error code.

ERRORS

When an error occurs errno is set to indicate the error. These error codes are defined in *objLib.h*.

S_objLib_OBJ_TIMEOUT
A call to **rdlock()** or **wrlock()** with a *timeout* other than **WAIT_FOREVER** or **NO_WAIT** timed-out.

S_objLib_OBJ_ID_ERROR
The readers writer lock contains an invalid semaphore possibly because the **rwlock** has been destroyed.

S_objLib_OBJ_UNAVAILABLE
A call to **rdlock()** or **wrlock()** with a *timeout* of **NO_WAIT** was made and the lock was not immediately available.

S_intLib_NOT_ISR_CALLABLE
The readers writer lock functions cannot be called from interrupt context.

NOTES

This readers writer lock implementation is intended to be fair for both readers and writers. It can guarantee that no readers or writers starve meaning that once a task has tried to acquire a read or write lock it will eventually succeed in the lock operation or timeout.

Synchronization is fundamentally at odds with priority, a lower priority task can prevent a higher priority task from running by locking out access to a critical section. The nature of readers writer locks is to have numerous concurrent readers or a single writer at any one time. These properties can be used to the advantage of a real time system. On the other hand, used carelessly readers writer locks can exacerbate the problems inherent between synchronization and priority. The property that many readers are allowed concurrently can be a very valuable tool. High priority tasks can run query types of operations concurrently with other lower priority tasks also running queries. At the other extreme, the same property can be deadly if a high priority writer needs to run but there are many lower priority readers that currently have the critical section read locked.

As with all other synchronization protocols, priority inversion is a factor. The classic example of this with a readers writer lock is when a low priority task holds a read lock to a **rwlock** and a high priority task tries to acquire a write lock to the same **rwlock**. The high priority task blocks, effectively having the low priority of the task that holds the read lock for the duration that the read lock is held by the lower priority task. This priority inversion can be for an unbounded duration because any unrelated task with priority between that of the low and high priority tasks can run during the priority inversion thus preventing the low priority task from releasing the read lock and allowing the high priority task to run.

A way to deal with this scenario is to use an ad hoc priority ceiling protocol. Before a task takes a read or write lock, raise the priority of that task to the maximum priority of all tasks that lock the **rwlock**. After releasing the lock, lower the priority to the original value. In VxWorks taskPrioritySet() is useful for this approach. With this approach it is clear that priority ordering on the wait queues is pointless so there is no need to use **RW_WRQ_PRIO**. This is a good approach for managing priority inversion problems and guaranteeing neither readers nor writers starve when tasks of medium priority cannot be avoided.

With care a band of priorities can be used for tasks that that use a shared readers writer lock. In this situation no other tasks run in this range of priorities. With no tasks unrelated to the **rwlock** of medium priority the priority inversion is bounded. This approach can be used to give different priorities to various tasks that acquire read and write locks to the same **rwlock**.

In both the priority ceiling and priority band approaches described above it is important to take into account situations when tasks may have their priority temporarily adjusted. In particular these two approaches do not mix well with the priority inheritance protocol which VxWorks implements for its mutual exclusion semaphores. The problem is that the use of a mutual exclusion semaphore with priority inheritance can be buried under a seemingly unrelated API. For example the default memory partition in VxWorks is a shared resource and thus API functions such as malloc() and free() are likely to use mutual exclusion semaphores with priority inheritance within their implementation. It is a good idea to only call functions which VxWorks allows to be called from interrupt context and those guaranteed not to block within critical regions when using the priority ceiling or priority band approaches.

In the VxWorks priority inheritance protocol, the priority of a task that holds a mutual exclusion semaphore with priority inheritance is temporarily increased to the priority of a higher priority task that blocks while attempting to take the same semaphore. If the task that holds the semaphore is blocked on another mutual exclusion semaphore with priority inheritance, the priority of the task holding that semaphore will be elevated as well. This elevation of priority follows the entire chain of tasks blocked on mutual exclusion semaphores taken and shared between the involved tasks. Thus VxWorks uses chaining in its implementation of a priority inheritance protocol. For all the tasks with elevated priority, their priority is only restored when the task releases all of the mutual exclusion semaphores it has taken. The priority will remain elevated when the task releases the semaphore which caused the priority inheritance to take place, if the task with an elevated priority continues to hold other mutual exclusion semaphores. It can be said the priority inheritance protocol of VxWorks exhibits unbounded priority inheritance because of this behavior. Thus, when considering the affects of the priority inheritance protocol in VxWorks to task priority, care must be taken to consider the effects of nested taking of mutual exclusion semaphores especially carefully.

Another approach to use to mitigate the problems of **rwlock** synchronization is to make use of the *timeout* to the **rdlock()** and **wrlock()** functions. If the lock cannot be acquired by your task within the *timeout*, then that task can take appropriate action such as logging an error and dropping the data. With care a deadline based approach can be a very useful approach in real time applications. For example, if a high priority writer timed-out waiting to acquire a write lock, it could raise the priority of all tasks possibly holding read locks and try to acquire the write lock again. Then when it has acquired the write lock it restores the priorities of the other tasks to their original values. As with all real time applications that use a deadline approach, be aware that the *timeout* is from when the task blocks to the time the task is unblocked and placed on the ready queue. Other tasks and system interrupts can greatly affect scheduling latency both before the task blocks and after the task unblocks but before it actually runs. This is a particularly important consideration when using readers writer locks because all waiting readers are placed on the ready queue simultaneously and run in priority order. Thus the time-outs that are chosen must budget for all other possible running tasks and still meet the hard deadlines.

If you really have a system where priority inversion is not a problem, then you might as well run all of your tasks at the same priority and use **kernelTimeSlice()**. Because there is a cost to every context switch on any real hardware, tasks should only be of differing priorities when there is good reason for doing so.

The readers writer lock functions are multi-task safe, multiple tasks may call these functions concurrently within the limits spelled out in the **DESCRIPTION** section. It is not safe to call any of the readers writer lock functions from interrupt context or from a signal handler. Technically you can call the **rwlock** functions from a signal handler but the code is not reentrant from the same task. A signal handler could interrupt one of these functions and if that signal handler called any of the **rwlock** functions, this could corrupt the internals of the **rwlock** itself. There is little incentive for the readers writer lock functions to be reentrant. For example take the case of a task that has called **rdlock()** on a **rwlock**. While the lock is held, the same task receives a signal and the signal handler performs a **rdlock()** on the same **rwlock**. This is clearly a programmer logic error because it is always an error to lock a **rwlock** that is currently locked. If you do call these functions from a signal handler, only do so in a context when you know that the current task cannot also be calling these functions. Be aware that many of the **rwlock** functions can block. There is very little good reason and many problems with calling functions that can potentially block from a signal handler. Please fully understand the consequences and restrictions of calling these functions from a signal handler if you choose to do so.

rdlock() and **wrlock()** exhibit interesting behavior when a *timeout* is used and a signal is received. In such a case, the call to **rdlock()** or **wrlock()** is interrupted, the signal handler is executed, and the **rdlock()** or **wrlock()** is restarted with the original *timeout*. This can cause unexpectedly long delays and is caused by a limitation of VxWorks where all functions are restarted, with their original time-outs, on your behalf after a signal handler is executed. Probably for this reason alone it is not worth while to use signal handlers in a real time VxWorks application save for handling of nonrecoverable errors.

Task deletion safety is limited in VxWorks. You may be interested in using **taskSafe()** and **taskUnSafe()** where appropriate to give some protection to critical regions in your code synchronized with readers writer locks. The **rwlock** functions themselves provide no task deletion safety themselves. If you wish for the limited protection to task deletion that VxWorks can provide to be used for the calls to the **rwlock** functions themselves, use **taskSafe()** and **taskUnSafe()** around all of the calls to the **rwlock** functions made.

If you follow the convention of first obtaining a write lock to a readers writer lock before calling **rwdelete()**, do not misunderstand how this can be useful and how it can be misused. It is an error to call any of the readers writer lock functions on a **rwlock** that has been or is being destroyed. Imagine that a task is in the midst of acquiring a read lock to a **rwlock** when a higher priority task interrupts and acquires a write lock and then destroys the **rwlock**. Later the lower priority task will run and dereference a pointer to the now destroyed readers writer lock. Here a number of things can take place all of them bad and in the category of errors where resources are used after they have been returned to the resource pool. If in the meantime the address pointed to by *rwp* has not been reallocated, a stale memory location will be used. Alternatively, *rwp* may point into a block of memory that has at this point been allocated for some other use. In this case some other data will be corrupted. This data could even be a new readers writer lock that has been created in the meantime. All of these errors are very difficult to locate and it beneficial to understand and avoid them. What this technique is useful for is that it can be used to sequence the destruction of a **rwlock**. All of the other tasks that block while trying to acquire locks to the **rwlock** after the write lock has been acquired will return with an error when the task that acquired the write lock destroys that **rwlock**.

HISTORY

The first implementation of readers writer locks in VxWorks at Fermilab was written by Mike Sliczniak to be used in Rich Neswold’s ErrLog module. This original implementation was based on the multireader locks described in: UNIX Systems for Modern Architectures, written by Curt Schimmel. Later an improved version was written by Mike Sliczniak to support time-outs and provide a simplified API.

AUTHOR

Mike Sliczniak

SEE ALSO

kernelTimeSlice, sysClkRateGet, taskPrioritySet, taskSafe, taskUnSafe,
VxWorks Programmer's Guide, chapter 2: Basic OS