Writing Multiprocessor Device Drivers

This chapter addresses questions particular to device drivers that run on multiprocessor workstations. It contains the following sections:

By default, all upper-half device driver functions—**open**(), **close**(), **ioctl**(), **read**(), **write**(), and **strategy**()— and the interrupt function are forced onto processor 0 on a multiprocessor (MP) system. Therefore, a device driver written for a single processor will work unmodified on a multiprocessor system. To avoid context switches to processor 0 for every I/O call, you can modify a device driver to run on any processor. The process of making a device driver MP-safe is often called *semaphoring*, or *multi-threading*, a driver, although the preferred method relies, strictly speaking, on locks rather than on semaphores.

Note: The driver interface now uses the SVR4 MP DDI/DKI interface except for the Silicon Graphics-specific routines, such as **pio_map**() and **dma_map**(). For example, entry points such as **open**(), **close**(), **read**(), and **write**() all have slightly different arguments and, in some cases, different procedure types in 5.x than they had in earlier versions.

Preliminary Considerations

The best way to develop a multiprocessor driver is to follow these steps:

1. Implement and test a single-threaded version of the driver.
2. Make the driver MP-safe with the use of the spinlock and semaphore calls described in this chapter.
3. Test and debug this version of the driver. It must still work perfectly when forced onto processor 0.
4. Add *D_MP* to the *drv***devflag**, recompile the driver, and rebuild the kernel with **lboot**.

   This builds a kernel that no longer forces the upper-half routines onto processor 0; they are allowed to run on the current CPU instead.
5. Run the device driver routines on an arbitrary processor to test and fix any bugs that have been exposed.

Unfortunately, making a device driver MP-safe is not a mechanized procedure, but one that requires a good understanding of the driver and its data structures. When a driver is flagged as *semaphored*, the driver writer must modify the code to prepare for two new scenarios:

- An upper-half routine may be executing on one processor while the interrupt procedure executes on another processor.
- Upper-half routines may run concurrently on different processors. For example, a process on one processor can be executing an **open** procedure while another processor executes the **strategy** function.

Shared Data between Upper-half and Interrupt Routines

If the upper-half and interrupts use **biowait** and **biodone** for synchronization, the driver will perform as desired on both single-processor and multiprocessor systems. This is because these routines have already been made multiprocessor-safe. Often, however, an interrupt routine will synchronize with upper-half procedures by using the **sleep/wakeup** functions and a shared flag word. The following is a common scenario:

Upper-half routine:

```
s = splvme();
flag |= WAITING;
while (flag & WAITING) {
   sleep(&flag, PZERO);
}
splx(s);
```

Interrupt routine:

```
if (flag & WAITING) {
   wakeup(&flag);
   flag &= ~WAITING;
}
```

The **splvme** call is used to protect *flag* from being modified in an interrupt routine. The **splvme** call raises the interrupt priority level only on the current processor and is, thus, insufficient on a multiprocessor. In this case, semaphores can be used for synchronization. When you initialize a semaphore to 0, the first **psema** call puts the calling process to sleep. A subsequent vsema(D3X) call will put the process back on the run queue. See the spl(D3) man page.The following code can replace the upper-half and interrupt scenario above:

Initialization function:

```
initnsema(&driversema, 0, "driver");
```

Upper-half routine:

```
psema(&driversema, PZERO);
```

Interrupt routine:

```
vsema(&driversema);
```

Since the semaphore functions themselves are multiprocessor-safe, no additional locking is necessary.

There may be other cases within the driver where data not specifically pertaining to synchronization is shared between an upper-half routine and an interrupt routine. You can identify these cases easily by searching for **spl***N*/**splx** calls and identifying the data actually being protected against concurrent access. In such cases, it is often useful to employ spinlocks. (They are called spinlocks because the locking functions actually loop until a test-and-set value is **unset**.) Replace the **spl***N* call with a **LOCK**(D3) call, and replace the **splx** call with a corresponding **UNLOCK**(D3) call. These replacements allow the driver to perform as desired on a single processor while providing locking on a multiprocessor. See the spl(D3) man page.

Caution: Data and cache interactions must be considered.Variables that might be in registers must be declared volatile and protected as well in multiprocessor device drivers.

Protecting Shared Data Among Upper-half Routines

Since instances of the device **open**, **close**, **ioctl**, **read**, **write**, and **strategy** functions may execute concurrently on a number of processors, all data that is shared among these routines must be protected. Unfortunately, you need to identify this data by careful examination of the driver code. It is not possible to look for all instances of certain procedure calls, as it is with the interrupt routine.

You may use spinlock calls to protect shared data where the locks are held only for a short period of time. If a lock must be held for a longer period of time, you may use a semaphore initialized to 1. If this is the case, the first call to **psema** is not blocked, but all succeeding calls are. When the lock is to be freed, a vsema(D3X) call allows at most one process waiting for the semaphore to proceed. Semaphores involve slightly more overhead than spinlocks if the lock is free, and a great deal more overhead if the lock is held and the calling process must sleep. This may sound undesirable at first, but keep in mind that waiting on a locked spinlock ties up the processor from other work, while a process that puts itself to sleep allows the CPU to execute other processes. Thus, spinlock calls should be used only in situations where the lock will be held for a short duration.

Semaphore and Spinlock Calls

The remainder of this chapter is a listing of semaphore and spinlock calls. In each case, an example of the call precedes a brief explanation:

semap

```
#include <sys/types.h>
#include <sys/sema.h>
void initnsema(sema_t *semap, int value, char *name);
```

Allocate and initialize a semaphore addressed by **semap**, given *value* and *name* (for debugging).

freesema

```
void freesema(sema_t *semap);
```

Free the semaphore addressed by **semap**.

psema

```
int psema(sema_t *semap, int priority);
```

Decrement the current semaphore value by 1; if the semaphore value becomes less than 0, sleep at the given priority. The priority is the same as that given to **sleep**. The flag bit *PCATCH* may be bit-wise ORed into the priority if the sleep is breakable (greater than PZERO) and it is desired to catch the signal (as is usually the case).

The call may be prefixed with *ap* if the call is to be a NOP on single processors. This is often the case when the semaphore is used for locking.

This function returns 0 in normal operation or -1 if *PCATCH* is specified and a signal interrupted the sleep.

vsema

```
int vsema(sema_t *semap);
```

Increment the current semaphore value by 1; if the result is less than or equal to 0, place a process sleeping on the semaphore onto the run queue. As above, the call may be prefixed with *ap* if the call is to be a NOP on single processors.

This function returns 0 if no process is waiting on the semaphore, or 1 if a process is awakened.

cpsema

```
int cpsema(sema_t *semap);
```

This call conditionally provides the functionality of the **psema** operation. If the semaphore count is already less than 0, the function does not affect the semaphore value and simply returns 0. Otherwise, the semaphore count is decremented.

Note: In no case does the calling process sleep; this function can be useful to test whether a given lock has been acquired.

cvsema

```
int cvsema(sema_t *semap);
```

This function wakes up a process on the semaphore if there is one. More precisely, if the semaphore count is less than 0, it increments the semaphore count, places a process on the run queue, and returns 1; otherwise, the semaphore is unaffected and the function returns 0.

LOCK_ALLOC

```
lock_t *LOCK_ALLOC(uchar_t hierarchy, pl_t min_pl
        lkinfo_t *lkinfop, int flag);
```

This call dynamically allocates and initializes a basic lock. The lock is initialized to the unlocked state. Silicon Graphics does not support the compilation option **_LOCKTEST**, but does provide **splockmeter** for debugging purpose.

LOCK_DEALLOC

```
void LOCK_DEALLOC(lock_t *lockp);
```

This call frees an instance of a basic lock.

LOCK

```
int LOCK(lock_t, lock, int (*splr)());
```

On multiprocessor systems, this call acquires the given spinlock, *lock*. The interrupt priority level is set to at least *splr* while the lock is acquired.

On single processor systems, this calls the **spl** function **splr**.

This function returns the old priority level.

UNLOCK

```
void UNLOCK(lock_t lock, int s);
```

On multiprocessor systems, this call releases the given spinlock **lock** and restores the interrupt priority level to **s**.

On single-processor systems, restore the interrupt priority level to **s**. This is the value returned to **LOCK** above.

SLEEP_LOCK

```
void SLEEP_LOCK(sleep_t *lockp, int priority);
```

This call acquires the sleep lock specified by *lockp*. If the lock is not immediately available, the caller is put to sleep (the caller's execution is suspended and other processes may be scheduled) until the lock becomes available to the caller, at which point the caller wakes up and returns with the lock held.

The caller is not interrupted by signals while sleeping inside **SLEEP_LOCK**. See **psema**(D3X).

SLEEP_LOCK_SIG

```
boolean_t SLEEP_LOCK_SIG(sleep_t *lockp, int priority);
```

This function acquires the sleep lock specified by *lockp*. If the lock is not immediately available, the caller is put to sleep (the caller's execution is suspended and other processes may be scheduled) until the lock becomes available to the caller, at which point the caller wakes up and returns with the lock held.

SLEEP_LOCK_SIG may be interrupted by a signal, in which case it may return early without acquiring the lock.

If the function is interrupted by a job control stop signal (such as **SIGSTOP**, **SIGTSTP, SIGTTIN, SIGTTOU**), which results in the caller entering a stopped state, the **SLEEP_LOCK_SIG** function transparently retries the lock operation upon continuing (the call will not return without the lock).

If the function is interrupted by a signal other than a job control stop signal, or by a job control stop signal that does not result in the caller stopping (because the signal has a non-default disposition), the **SLEEP_LOCK_SIG** call returns early without acquiring the lock.

SLEEP_UNLOCK

```
void SLEEP_UNLOCK(sleep_t *lockp);
```

This function releases the sleep lock specified by *lockp*. If there are processes waiting for the lock, one of the waiting processes is awakened. See vsema(D3X).

SLEEP_TRYLOCK

```
boolean_t SLEEP_TRYLOCK(sleep_t *lockp);
```

This function tries to acquire a sleep lock. See cpsema(D3X).

TRYLOCK

```
int TRYLOCK(lock_t *lockp, pl_t pl);
```

This function tries to acquire a basic lock.

Caution: Drivers that reacquire multiple locks may deadlock when an asynchronous processor obtains a needed lock and does not free it because a lock held by another processor is also looking for a lock.

Multiprocessing STREAMS Drivers

In IRIX, all STREAMS activity is single-threaded through use of a STREAMS monitor. The kernel takes care of acquiring the monitor before running any of the regular STREAMS entry point routines. However, the device driver writer needs to take care of the interrupt entry points (hardware interrupt and timeouts) with streams_interrupt(D3X) and STREAMS_TIMEOUT(D3X). For more detailed information, see the man pages for these two calls.

STREAMS Monitor

The STREAMS monitor ensures mutually exclusive access to STREAMS on multiprocessor systems. The STREAMS put, service, open, and close functions are guaranteed to have the monitor upon entry and, thus run with assured mutual exclusion. The kernel handles all monitor interactions for these procedures, although it does not acquire the monitor for STREAMS driver interrupt routines, which must acquire the monitor explicitly.

All STREAMS drivers must acquire the monitor before performing any interaction with STREAMS from interrupt level, such as **quenable**(), **getq**(), or **putq**(). To obtain the monitor from an interrupt routine, the driver should call:

```
int streams_interrupts (func,arg1,arg2,arg3)
```

This routine either:

1. Acquires the monitor and runs **func** with arguments of *arg1*, *arg2*, and *arg3*, then releases the monitor and returns 1

   or

2. Queues the function on the monitor for execution once the current owner of the monitor releases it, and immediately returns 0. The example below shows how a STREAMS driver could use **streams_interrupt**().

There are additional changes for STREAMS drivers that use calls to **timeout**() and **delay**(), which corrupt the mutual exclusion of the monitor. To make these calls safe, the device driver writer must replace them with macros defined in the include file *sys/strmp.h*.

Replace all calls to **timeout**() with the macro **STREAMS_TIMEOUT**(); replace all calls to **delay**() with the macro **STREAMS_DELAY**(). For example, if the single-processor version of a driver contains the following calls:

```
timeout(watchdog,unit,HZ/10);
```

and

```
delay(100);
```

they should be replaced by:

```
STREAMS_TIMEOUT(watchdog,unit,HZ/10;
```

and

```
STREAMS_DELAY(100);
```

These macros revert to the original **timeout**() and **delay**() calls in the single processor case. The include file *sys/strmp.h* also defines the constant *MP_STREAMS* if the multiprocessor version of STREAMS is in use. This is useful for performing conditional compilation of sections of the STREAMS driver for multiprocessor systems. In any case, the use of these macros and definitions makes the driver machine-dependent.

STREAMS Example

```
#include "sys/strmp.h"

static void interrup_handler;

/* Actual interrupt routine that is called on an interrupt */
driverintr(unit)
int unit:
{
    /* Check to see if interrupt is valid */
    if (driver[unit]->intrmask !=0 {
        /* Call the interrupt handler that interacts
         * with STREAMS */
        streams_interrupt(interrupt_handler,unit);
    }
    else {
        /* Stray interrupt! */
        driver[unit]->stray++;
    }
    return;
}

/* Second-level interrupt handler that interacts with
 *  STREAMS.  This guarantees mutually exclusive access
 *  to STREAMS */
static void
interrupt_handler(unit)
int unit;
{
    register mblk_t *bp;

    if ((bp = allocb(128,BPRI_HI)) ==0) {
        /* Unable to allocate STREAMS block */
        driver[unit]->allocb_fail++;
        return
    }

    /* Copy data into message block */
    bcopy(driver[unit]->data,bp->wptr,128);

    /* Put onto our read queue for additional processing */
    putq(driver[unit]->rq,bp);
    return;
}
```