Porting Applications to the AIX 4.1 OS SMP Environment

By Debora Blakely-Fogel

This article highlights the changes needed in AIX 4.1 operating system uniprocessor applications so that they also function correctly on AIX 4.1 SMP systems. It is written for AIX developers who are moving applications from AIX 3.2.5 to AIX 4.1. It addresses only the porting issues directly attributed to MP. The reader should have a thorough understanding of AIX device drivers.

AIX 4.1 supports Symmetric Multiprocessor (SMP) systems based on the PowerPC family of processors. The programming model for AIX 4.1 uniprocessor systems is slightly different from the programming model for AIX 4.1 SMP systems.

Background of the PowerPC Architecture

To make an existing AIX application run correctly on an AIX Version 4 SMP system, it helps to first understand two key features of the PowerPC Architecture™: its weakly ordered memory and its out-of-order instruction execution.

Weakly Ordered Memory

The PowerPC Architecture specifies a memory model with relaxed instruction ordering requirements. Execution of a load or store instruction is considered complete when the associated address translation completes. At this point, the instruction has completed to the point that it will not generate an internal exception, although a subsequent read or write operation can still generate an external exception.

Load and store instructions are always issued and translated in program order with respect to other load and store instructions. However, a load or store operation that hits in the cache can complete ahead of those that miss in the cache. In addition, loads and stores that miss in the cache can be reordered as they arbitrate for the system bus.

The PowerPC 601™ processor ensures memory consistency by comparing target addresses and prohibiting instructions from completing out of order if an address matches. Load and store operations can be forced to execute in strict program order.

Debora Blakely-Fogel

Out-Of-Order Instruction Execution

In the PowerPC family of processors, instructions are not necessarily executed in sequential order. An instruction in the instruction cache can be sent to an idle execution unit for execution, even though it is not the next sequential instruction. For example, if an integer instruction is executing and the floating-point unit is not busy, a floating-point instruction can be sent to a floating-point unit to execute out of order while the integer unit is executing the integer instruction. Instructions that depend on the result of previous instructions will not execute out of order.

SMP Implications of Binary Compatibility

In general, applications that execute correctly on a uniprocessor RISC System/6000 running AIX 4.1 will also run correctly on an SMP RISC System/6000 running AIX 4.1. The exceptions are applications that serialize shared memory access among multiple running processes or threads and those that share memory with I/O. Typically, these applications use semaphores such as "compare and swap" to synchronize access to a particular segment of memory, and use a simple assignment to release the lock. While the following process works in AIX 3.2, it is important to

note that this will also work on an AIX 4.1 uniprocessor system. It is not sufficient for a PowerPC-based SMP system.

- Obtain the lock for the data, which can be done by calling `cs()` on AIX 3.2.
- Use the data secured by the lock.
- Release the lock with a simple assignment (such as `lock_word = 0;`).

Because of its weakly ordered memory model, the PowerPC can allow access to the application's critical data before obtaining the lock and also release the lock before the changed data is visible to the other processors. The following sequence of events will prevent these problems:

- Obtain the lock and synchronize the instruction caches.
- Use the data secured by the lock.
- Synchronize the buffers and release the lock.

Locking and unlocking for serializing access to shared memory does not require synchronization on a uniprocessor system, but it does on an MP system. AIX 4.1 operating system services can perform the extra required processing, making it easier for application developers. With AIX 4.1, locking is provided by the `_check_lock()` function and unlocking is provided by the `_clear_lock()` function.

The _check_lock() Primitive

Import and export fences are special processor-dependent synchronization instructions. They temporarily block other reads and writes to these locations. They protect against concurrent access by several processors, and against the read and write reordering performed by the PowerPC family of processors.

The `_check_lock()` function is an AIX 4.1 service similar to the compare and swap `cs()` service in AIX 3.2, with the addition of an appropriate import fence when the swap is successful. It differs from the `cs()` service in that it is restricted to word-aligned accesses only, as shown in Figure 1.

The _clear_lock() Primitive

This service is an atomic store with the appropriate export fence, as shown in Figure 2.

For uniprocessor systems, this routine is a simple store. For PowerPC SMP systems, it is preceded by the appropriate export fence. This routine does not need to execute disabled. The export fence guarantees that the data being protected is visible to all other processors prior to the lock word being released.

```
#include <sys/atomic_op.h>
boolean_t _check_lock(atomic_pword, int old_value, int
   new_value)
```

Figure 1. The _check_lock() service

```
#include <sys/atomic_op.h>
void _clear_lock(atomic_p word, int value)
```

Figure 2. The _clear_lock service

In addition to the synchronization changes required to make user applications run correctly on SMP systems, there are changes necessary for AIX device drivers to run correctly on SMP systems.

Applications that execute correctly on a uniprocessor RS/6000 running AIX 4.1 will also run correctly on an SMP RS/6000 running AIX 4.1.

Device Drivers

The AIX device driver models differ for multiprocessors and uniprocessors. Device drivers, as kernel extensions, have the same multiprocessor requirements as other kernel extensions. New features in AIX 4.1 that provide compatibility with AIX Version 3 uniprocessor device drivers also allow the programmer to tell the system the type of device driver when the device driver is registered. AIX 4.1 SMP systems support three types of device drivers:

- **Funneled**: Funneled device drivers run only on the master processor; therefore, the current uniprocessor serialization is sufficient. These device drivers are intended to support low-throughput devices. The base kernel provides binary compatibility for these device drivers.
- **MP Safe**: These device drivers run on any processor and contain modified code for adding a code lock to serialize the device driver's execution. MP Safe device drivers are intended for medium-throughput devices.
- **MP Efficient**: MP Efficient device drivers run on any processor. Their code has been modified to add data locks to serialize the device

driver's accesses to devices and data. MP Efficient device drivers, intended for high-throughput devices, are MP Safe device drivers that have been tuned for performance and efficiency.

The `devswadd()` kernel service defines a device driver to the AIX kernel. A field added to the device switch table indicates whether a device driver is to be funneled, MP Safe, or MP Efficient. To the base kernel, MP Safe and MP Efficient are the same. The base kernel assumes that the device driver is a funneled device driver unless the `devsw` structure's new `d_opts` field is modified to specify the `DEV_MPSAFE` flag before registering the device driver with the call to `devswadd()`. The device driver will be treated as an MP Safe device driver if `DEV_MPSAFE` is specified.

The AIX device driver models differ for multiprocessors and uniprocessors.

Top Half Considerations

A device driver's *top half* is the part that executes at the process level. It is not part of an interrupt-thread critical section, which serializes the execution of code that executes either on an interrupt level or at the process level with code that executes on an interrupt level. An interrupt-interrupt critical section is a special case of this type of critical section. The device driver's top half can be part of a thread-thread critical section, one that serializes the execution of code executing at the process level with other code executing at the process level.

The problem of concurrent access to global data is not specific to MP systems nor is it new to AIX 4.1. AIX Version 3 is a preemptable kernel, requiring that device drivers synchronize access to critical sections of code in their top halves. AIX Version 3 provided the `lockl()` and `unlockl()` kernel services for this serialization, which are also available with AIX Version 4.

In addition, two new types of locks can be used for thread-thread serialization: simple locks and complex locks. There are at least two reasons to consider replacing existing `lockl()` and `unlockl()` calls with these new services. The services for these new locks have instrumentation built into them that can be used to detect bottlenecks and to help in debugging deadlocks. During the development of AIX Version 4, these new lock services were tuned to provide efficient use of the system's resources and will often provide better performance than the `lockl()` and `unlockl()` locks.

Converting `lockl` locks to either simple or complex locks is not always straightforward. The simple locks, the preferred lock type, are functionally closer to `lockl` locks; however, `lockl` locks, unlike simple locks, allow nesting. Simple locks are recommended when `lockl` locks are replaced except when the work to remove nesting is too expensive or another feature of complex locks would help simplify the code. For example, complex locks can synchronize access to code that is rarely modified but frequently read.

The logical filesystem provides several forms of serialization for device driver top half routines:

- Ensures that the device driver's close routine is not called while it is performing a `read`, `write`, or `ioctl`.
- Serializes calls to the device driver's configuration routine (`sysconfig SYS_CFGDD`) with calls to the open and close routines. This serialization is only a code lock to ensure that the two sets of routines are not called simultaneously.

The device driver ensures that the call is valid for the current state. For example, the device driver must return an error when a request is made to unconfigure an open device.

Bottom Half Considerations

A device driver's *bottom half* is the part that executes in an interrupt-thread critical section at the process level, and all code that executes at the interrupt level.

There are many more considerations for making a device driver's bottom half MP Safe or MP Efficient than there are for the top half. The primary considerations for making the device driver MP Safe relate to serialization. In addition, there are some required interface changes for MP Safe device drivers.

Serialization Considerations

AIX Version 3 provides the `i_disable()` and `i_enable()` kernel services to mask interrupts or disable the processor. Although this disabling of interrupts provides adequate serialization on a uniprocessor system, it is not sufficient to serialize an MP Safe device driver's bottom half. Since the I/O is symmetric in an SMP and interrupts can be routed to any of the processors in the complex, masking interrupts will not prevent the interrupt routine from executing simultaneously on another processor. Serialization in an SMP requires the use of MP locks.

Not all device driver code disables interrupts to protect global data. In some cases, careful

ordering is used instead of disabling interrupts. This coding style will not work on a multiprocessor. Consider an example in which entries are added to and removed from a singly linked list only in a process environment. Updating the list can be done carefully to keep the list consistent. For example, an element might be added to the list with a simple assignment instruction only after all of its data fields are updated. Or the fields of an element might be modified only after it is removed from the list with a simple assignment instruction.

On a uniprocessor, this allows an interrupt handler to safely scan the list. This depends on process-level code never running simultaneously with the interrupt-level code. Although this is a correct assumption for a uniprocessor, it is incorrect for a multiprocessor. MP locks must be used to provide proper serialization.

Changing a device driver's bottom half serialization to make it MP Safe involves carefully replacing the uniprocessor interrupt locking with multiprocessor spin locking. This work consists of the following:

- Replacing `i_disable()` and `i_enable()` calls with calls to `disable_lock()` and `unlock_enable()`
- Adding serialization when disabled
- Removing nested locking
- Ensuring that interrupt-thread locks are never held across a sleep

AIX 4.1 provides the `disable_lock()` and `unlock_enable()` MP lock services for serializing interrupt-thread critical sections of a device driver's code. The `disable_lock()` service provides a spin lock that disables interrupts on the processor from which it is called. It also blocks the execution of code that tries to attain the same spin lock on another processor. The purpose of the `unlock_enable()` service is to release this spin lock and reenable the interrupts on the processor from which it is called.

Device Drivers Disabled When Called

Several device driver bottom half routines are disabled by the base kernel before they are called. These routines might not have serialization logic to prevent concurrent access to global data. To make these routines MP Safe, MP locks (`disable_lock()` and `unlock_enable()`) must be added to provide proper serialization. Figure 3 shows the device driver routines that are called disabled by the base kernel.

Routine	Location	Called at Interrupt Priority
EPOW handler	`intr.handler`	`INTEPOW`
Interrupt handler	`intr.handler`	`INTCLASS0 - INTCLASS3 (intr.priority)`
Timeout routine	`trb.func`	`INTCLASS0 - INTCLASS3 (trb.ipri)`
Watchdog	`watchdog.func`	`INTTIMER`
Off-level	`intr.handler`	`INTOFFL0 - INTOFFL3 (intr.priority)`
Iodone routine	`buf.b_iodone`	`INTIODONE`

Figure 3. Device driver routines disabled by base kernel

Uniprocessor Device Driver

```
oldpri = i_disable(my_intpri);
while ( io_not_done )
    e_sleep(&my_event);
i_enable(oldpri);
```

SMP Device Driver

```
oldpri = disable_lock(my_intpri, &my_lock);
while ( io_not_done )
    e_sleep_thread(&my_event, &my_lock,
    LOCK_HANDLER);
unlock_enable(oldpri, &my_lock);
```

Figure 4. Replacing MP Safe and MP Efficient device driver calls

Interrupt-Thread Locks Across Sleeps

Interrupts are automatically enabled when a process is put to sleep, then automatically disabled when the process is dispatched after wakeup. This is not true for multiprocessor spin locks; therefore, both MP Safe and MP Efficient device drivers must be changed to ensure that they never hold an interrupt-thread lock across a sleep. In general, this consists of replacing the device driver's calls to `e_sleep()` with calls to `e_sleep_thread()`. Figure 4 shows a typical example in which uniprocessor code would be changed to multiprocessor code.

The `LOCK_HANDLER` flag tells `e_sleep_thread()` that this simple lock is an interrupt-thread spin lock and not a thread-thread wait lock. This flag must always be specified for an interrupt-thread spin lock.

In addition, MP Safe and MP Efficient device drivers must be careful to release their multiprocessor spin locks when they call outside routines that might sleep. Although this is not a

```
if (disable_count == 0)
    oldpri = i_disable(my_intpri);
disable_count++;
    do_critical_section;
disable_count--;
if (disable_count == 0)
    i_enable(oldpri);
```

Figure 5. Device driver uses a variable to save path length

concern for the bottom half routines that run only on an interrupt level, it is a concern for the routines that can run at the process level. It is not necessary for locks to be released and reacquired when calling a base kernel service that can be called from an interrupt environment.

The order in which locks are acquired and the type of lock used are very important.

This discussion of explicit sleep calls also applies to implicit sleeps, such as page faults. An MP Safe or MP Efficient device driver cannot touch data that is not pinned while holding an interrupt-thread lock.

Removing Lock Nesting

The `i_disable()` service fully supports nesting, and many device drivers have nested calls to `i_disable()`. Since the `disable_lock()` service does not support nesting, all nested locking must be removed from a device driver to make it either MP Safe or MP Efficient. The best way to do this is to restructure the device driver so that nesting is not needed.

One successful technique to remove nesting is to have two names for each routine that can be called from either outside or inside a critical section. The first name is a small routine that is called from outside a critical section. This routine acquires the lock, calls the other routine to do the work, then releases the lock. The other routine assumes that the lock is already acquired and performs the function for which it has been called.

As shown in Figure 5, some device drivers may keep a variable to prevent nested calls to `i_disable` to save path length.

This technique cannot be used on a multiprocessor, even after the `i_disable()` and `i_enable()` calls have been changed to `disable_lock()` and `unlock_enable()`. That is because the initial test of `disable_count` is a critical section and could run concurrently on more than one processor. Since the AIX 4.1 base kernel does not provide an equivalent test that works for a multiprocessor, you must prevent nesting in a different way.

Preventing Deadlocks

The order in which locks are acquired and the type of lock used are very important. System deadlocks can occur when locks are not acquired and released in the correct order, or when an inappropriate type of lock is used.

One type of lock deadlock can occur when an interrupt-thread simple lock is acquired in a process environment without interrupts disabled. The system will deadlock if that thread is interrupted for any reason, and another instance of that device driver is called. The device driver will attempt to get its spin lock and will spin waiting on the preempted process to release the simple lock.

This can happen only when an interrupt-thread simple lock calls `simple_lock()` when interrupts are not disabled. It cannot happen when `disable_lock()` is called—yet another reason why `simple_lock()` should never be used directly for interrupt-thread critical section locks. The uniprocessor base kernel saves path length by not acquiring or releasing the simple lock when `disable_lock()` and `unlock_enable()` are called.

A more complex lock problem can occur when more than one lock must be acquired to perform an operation. A strict lock order must be defined to prevent deadlocks.

Consider the example illustrated in Figure 6 in which the different processes, pl and p2, need to acquire two different locks, lock1 and lock2. Assume the code being executed by process pl acquires lock1 before acquiring lock2, and the code being executed by process p2 acquires the locks in the opposite order. Assume also that between the time that p1 successfully acquires lock1 and the time that p1 tries to acquire lock2, process p2 runs and successfully acquires lock2. This results in a deadlock because process p1 will never release lock1 for use by process p2 until it has acquired lock2, and process p2 will never release lock2 for process p1 to use until it has acquired lock1. The locks must always be acquired in the same order and released in the inverse order.

Interface Changes for MP Safe and MP Efficient Device Drivers

Lock ordering problems are the reason for most changes in kernel interfaces for MP Safe and MP

Efficient device drivers. The base kernel either has to release its interrupt-thread locks before calling a device driver routine, which opens a serialization window, or it has to hold its interrupt-thread locks while calling a device driver routine, resulting in a possible deadlock. The changes in the base kernel interfaces are designed to prevent the problems that result from either of these situations.

MP Safe Registration

Since the kernel is designed to support binary compatibility for funneled device drivers, MP Safe and MP Efficient device drivers must inform the kernel that the extra funneling serialization does not apply to them. This is generally done when the `devswadd()` call registers the device driver; however, the kernel cannot relate certain device driver routines to the device switch entry of the device driver.

A device driver's strategy routine is defined to the base kernel via the `devswadd()` kernel service along with the device driver's top half routines. The `devstrat()` service will assume that the device driver is funneled unless the `d_opts` field of the `devsw` structure is modified to specify the `DEV_MPSAFE` flag prior to calling `devswadd()`. The device driver will be treated as an MP Safe or MP Efficient device driver if `DEV_MPSAFE` is specified. Strategy routine calls for funneled device drivers will be queued to be run on the master processor if the strategy routine is for a funneled device driver. Otherwise, the strategy routine will just be called. It may or may not have been called when `devstrat()` returns.

The `buf` structure's `b_iodone` field defines a device driver's `iodone` routine to the base kernel. The `iodone()` service assumes that the device driver is funneled unless the `dev` structure's `b_flags` field specifies `B_MPSAFE`. An MP Safe or MP Efficient device driver must set this flag in the `buf` structure before calling `devstrat()`. Otherwise, `iodone()` will run the device driver's `b_iodone` routine on the master processor.

The `i_init()` kernel service defines an interrupt handler to the base kernel. This service assumes a funneled device driver unless the `intr` structure's `flags` field is modified to specify the `INTR_MPSAFE` flag prior to calling `i_init()`. The device driver is treated as an MP Safe or MP Efficient device driver if `INTR_MPSAFE` is specified. The `INTR_MPSAFE` flag should also be specified for EPOW interrupt handlers and off-level

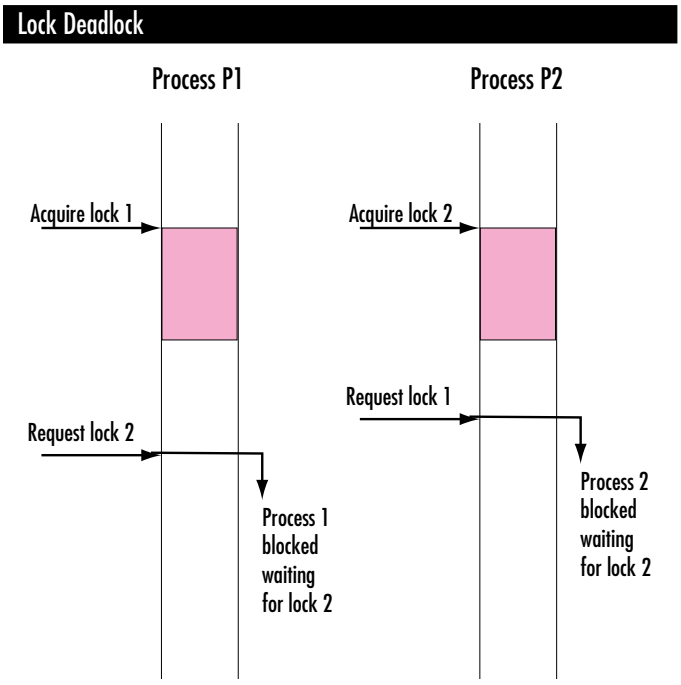

Figure 6. Lock deadlock

interrupt handlers. Although the initial multiprocessor base kernel does not require these flags to be set properly, setting them allows for future base kernel design changes. Similarly, the `T_MPSAFE` flag should be set in the flags field of the `trb` structure for MP Safe and MP Efficient device drivers.

Since there is no flags field in the watchdog handler's `watchdog` structure, watchdog handlers are architected to run only on the master processor or only on the processor from which `w_init()` is called.

Interrupt Handlers

The `i_init()` and `i_clear()` kernel services are called to add or remove an interrupt handler from the list of interrupt handlers known to the base kernel. The base kernel can always succeed at adding or removing the specified interrupt handler on a uniprocessor. It may not be successful on a multiprocessor if interrupts are being processed on another processor. Therefore, neither `i_init()` nor `i_clear()` can be called while holding an interrupt-thread lock. Calling either of these services while holding an interrupt-thread lock can result in a deadlock.

Timeout Request Handlers

The `tstop()` kernel service is called to remove a timeout request from the list of timeout requests known to the base kernel. Although the base kernel can always succeed at removing the specified timeout request on a uniprocessor, it may not be successful on a multiprocessor if timeout requests are being processed on another processor. A return value of 0 has been added to `tstop()` if the timeout request is successfully removed; otherwise it is -1. The `tstop()` calls must be changed in an MP Safe or MP Efficient device driver.

If an interrupt-thread lock was held when `tstop()` was called, the lock must be released and reacquired before retrying the call to `tstop()`. This allows the requests on the other processors to complete. Some delay may be required between releasing the interrupt-thread lock and reacquiring it. The `tstop()` service may fail even if an interrupt-thread lock is not held when it is called. In this case, the caller does not have to release and then reacquire the lock, but only retry the `tstop()` call.

Watchdog Timers

The `w_init()` and `w_clear()` kernel services add or remove a watchdog timer handler from the list of watchdog timer handlers known to the base kernel. The base kernel can always succeed at adding or removing the specified watchdog timer handler on a uniprocessor. It may not be successful on a multiprocessor if watchdog timers are being processed on another processor; therefore, a return value of 0 has been added to `w_init()` and `w_clear()` if the watchdog timer handler is successfully removed; otherwise it is -1. The `w_init()` and `w_clear()` calls in MP Safe or MP Efficient device drivers must be changed in a similar way to the calls to `tstop()`.

If an interrupt-thread lock was held when `w_init()` or `w_clear()` was called, the lock must be released and reacquired before retrying the call to `w_init()` or `w_clear()`. This allows the requests on the other processors to complete. Some delay may be required between releasing the interrupt-thread lock and reacquiring it. The `w_init()` or `w_clear()` service can fail even if an interrupt-thread lock is not held when it is called. In this case, the caller does not have to release and then reacquire the lock, but just retry `w_init()` or `w_clear()`.

The device driver's watchdog timer routine might be called even after the device driver calls `w_stop()` to disable the watchdog timer. This occurs only in a very small window when the kernel is in the process of calling the watchdog timer and the device driver calls `w_stop()` because of an interrupt that completes the I/O operation. This window exists in the current uniprocessor versions of AIX and will continue to occur in future versions of AIX. The device driver may need to use some form of state information between its interrupt handler and its watchdog timer routine to prevent accidental timeouts because of this.

I/O Processing

The base kernel currently sets `B_DONE` before calling the buffer's `b_iodone` routine. It will continue to do this for funneled device drivers, but it will not set `B_DONE` for MP Safe or MP Efficient device drivers. The MP Safe or MP Efficient device driver's `b_iodone` routine must set `B_DONE` within its critical section while holding its interrupt-thread lock. Otherwise, the setting of `B_DONE` and the wakeup of any process waiting on this I/O are not serialized correctly. This also implies that MP Safe and MP Efficient device drivers cannot call `iowait()`. Instead, they must provide a routine that sleeps on the buf's `b_event` field when `B_DONE` is not set. In addition, after setting `B_DONE`, the device driver must awaken processes that are waiting for the I/O to complete.

Dump Routines

The dump routine for an MP Safe or MP Efficient device driver should be coded to assume the worst case. The dump routine should assume that one critical section of the device driver was active on another processor when the dump was initiated, and the device driver's hardware may be in an indeterminate state. The dump routine should first establish a well-defined state at the start of the first dump write.

The dump routine may call parts of the device driver or kernel that have interrupt-thread lock calls. The kernel will ensure that no deadlock occurs due to these calls.

Debora Blakely-Fogel, *IBM Corporation, RISC System/6000 Division, 11400 Burnet Road, Austin, TX 78758. Ms. Blakely-Fogel is an advisory programmer responsible for providing technical assistance to software vendors. She has a BS in Mathematics from the University of Massachusetts at Lowell and an MS in Computer Science from New Mexico State University.*