4 Multiprocessing
HP-UX servers and workstations can be either uniprocessor or multiprocessor systems. Current and new drivers for either servers or workstations must be written to be multiprocessing safe, because they may eventually run on multiprocessor systems. This chapter covers the kernel services that handle synchronization used by drivers on multiprocessor systems.
Comparing Uniprocessing to Multiprocessing

A uniprocessor (UP) system is comprised of exactly one processor. A driver in a UP system may be executing in only one thread at any given time. That thread will either be a kernel thread (the upper half of a driver), or on the interrupt control stack (ICS) in a processor interrupt context (the lower half of a driver). The UP driver synchronization model coordinates execution between the driver’s upper and lower halves.

A multiprocessor (MP) system is comprised of two or more processors. A driver in an MP system may be executing in multiple threads concurrently. For each processor, the driver may be executing in a kernel thread or in the interrupt context of that processor. The MP driver synchronization model coordinates execution among multiple kernel threads as well as between the driver’s upper and lower halves.

HP-UX is a multiprocessing operating system, and as such, drivers must be written for MP systems. The MP synchronization mechanisms provided by HP-UX are spinlocks and beta semaphores. Drivers that use these synchronization mechanisms will work correctly on both MP and UP systems.
Synchronization Mechanisms

Spinlocks are the most heavily used synchronization mechanism in the HP-UX kernel. They are used to protect data accessible from either a kernel thread or an interrupt context. Only one processor is allowed to own a spinlock at any given time. Other processors that attempt to acquire an owned spinlock will spin and wait for the spinlock to be released by the owning processor.

External interrupts for a processor are disabled for the duration when a processor owns a spinlock or when it attempts to acquire a spinlock. Because external interrupts are disabled, spinlocks must not be owned (i.e., locked) for lengthy periods of time. Likewise, spinlocks must not be held across calls to system services that may block (put the thread to sleep).

Semaphores, which are known in the HP-UX kernel as beta semaphores, provide another synchronization mechanism. They are used to protect data that are accessed by a driver's upper half (executing in a kernel thread that may block). They can not be used to protect data that are accessed by a driver's lower half (executing in an interrupt context that can not block).

Beta semaphores provide mutual exclusion where only one kernel thread is allowed to own the semaphore at any given time. Other kernel threads attempting to acquire the semaphore will be blocked until the semaphore is released.

Unlike spinlocks, beta semaphores may be held across calls to system services that may block.

Timing Hazards and Idle Time

Timing hazards, also known as race conditions, can occur on an MP system. Careful regression testing on MP systems is essential to expose timing hazards that may occur in a driver.

Designs where beta semaphores are owned for lengthy periods of time can cause the idle time of the system to increase as kernel threads are forced to block and wait. This situation can be detected with tools such as *top* and *sar* (see *top*(1) and *sar*(1), and the optional HP products *LaserRX* and *Glance/UX*).
Spinlocks

Spinlocks are the basic locking primitive used by the kernel for short-term locks. When a thread acquires a spinlock, the thread's current processor becomes the effective owner until the spinlock is released. Threads (processors) waiting to acquire an owned spinlock will spin while waiting -- they do not block. For the duration that a processor owns a spinlock, external interrupts to the processor are disabled. External interrupts to the processor are disabled to avoid a potential interruption deadlock. Consider the case where driver code is executing on a processor and owns (i.e., has locked) a spinlock. If external interrupts are not disabled, an interrupt from a device may cause the interrupt service routine (ISR) of the driver to be entered on the same processor. If the driver's ISR attempts to lock the same spinlock, a deadlock will occur because the spinlock is already owned by the processor. The ISR will spin and wait forever.

A spinlock that is owned by a processor makes other processors spin and wait if they attempt to acquire the same spinlock. The other processors burn CPU cycles without doing useful work when this occurs. Therefore, drivers should be designed to hold spinlocks for only short periods of time. A general rule of thumb is if a spinlock is held for longer than a few milliseconds, then it is being held too long.

Spinlock Routines

HP-UX provides the following spinlock routines. Refer to the *HP-UX Driver Development Reference* for detailed descriptions.

- alloc_spinlock() - allocate and initialize a spinlock resource.
- cspinlock() - conditionally lock a spinlock if the spinlock is not owned.
- dealloc_spinlock() - deallocate a spinlock resource.
- owns_spinlock() - check if the processor owns a spinlock.
- spinlock() - acquire (lock) a spinlock.
- spinunlock() - release (unlock) a spinlock.
Beta Semaphores

Beta semaphores are mutually-exclusive, blocking semaphores. When a thread acquires a beta semaphore, it is the owning thread until the beta semaphore is released. The owning thread may subsequently block (i.e., sleep) and still keep ownership. Threads waiting to acquire an owned beta semaphore are blocked.

Since blocking may occur, beta semaphores must not be acquired by a driver while executing in the interrupt context of a processor.

Beta Semaphore Routines

HP-UX provides the following beta semaphore routines. Refer to the HP-UX Driver Development Reference for detailed descriptions.

- `b_cpsema()` - conditionally acquire (lock) a beta semaphore if it is not currently locked.
- `b_initsema()` - initialize a beta semaphore.
- `b_owns_sema()` - test whether a beta semaphore is owned by the current thread.
- `b_psema()` - acquire (lock) a beta semaphore.
- `b_vsema()` - release (unlock) a beta semaphore.
Deadlocks

If a driver acquires beta semaphores or spinlocks in an incorrect order, a deadlock may occur.

The classic illustration of a deadlock is the case of processes A and B which both need resources C and D to complete an activity. If process A locks resource C and process B locks resource D, each will be blocked forever waiting for the resource held by the other process.

To avoid deadlocks, each thread must acquire its locks in the same order. In the above example, processes A and B must acquire resource C before they try to acquire resource D.

Rules for Lock Acquisition

Beta semaphores and spinlocks (and the resources they protect) are assigned a lock order, which is used as follows:

- When a thread of execution acquires a spinlock unconditionally, the order of the requested spinlock must be greater than the order of any spinlock the processor already holds.
- When a kernel thread acquires a semaphore unconditionally, the order of the requested semaphore must be greater than the order of any semaphore the kernel thread already holds.
- If the orders of the acquired and held beta semaphores are equal, both beta semaphores must have the deadlock safe option set. This option is set by ORing the order with the SEMA_DEADLOCK_SAFE bit when the semaphore is initialized.
- Spinlocks have the highest order. A thread of execution must acquire all beta semaphores it requires before it acquires a spinlock.

Lock Orders

The header file `<sys/semglobal.h>` contains the lock orders used by HP supplied kernel services. Drivers typically choose a lock order that is low in value so that the driver can hold its own spinlock (or beta semaphore) while calling a kernel service.
In addition to spinlocks and beta semaphores, HP-UX provides another synchronization mechanism using the system services `sleep()` and `wakeup()`. Typically, the upper half of a driver will start an asynchronous activity and wait for the lower half to complete the activity. The system service `sleep()` is called by the driver's upper half to block the kernel thread and put it on a sleep queue. The driver's lower half calls `wakeup()` to take the kernel thread off the sleep queue and to awaken the thread.

A race condition exists between the time a kernel thread calls `sleep()` and the time `wakeup()` is called. The `wakeup()` routine can be called before the kernel thread has been put on a sleep queue. To handle this race condition, a call to `get_sleep_lock()` must be made before calling `sleep()`.

The routine `get_sleep_lock()` acquires a sleep queue spinlock that is later released by `sleep()` after the kernel thread has been put on the sleep queue. The routine `wakeup()` acquires the same sleep queue spinlock before taking the kernel thread off the sleep queue and awakening the sleeping thread. Drivers typically call `get_sleep_lock()`, start an asynchronous activity, then call `sleep()` as shown below:

```c
(void)get_sleep_lock(wait_chan);
start_async_activity();
(void)sleep(wait_chan, PRIBIO);
```

When the asynchronous activity completes, the driver's `async_completion()` routine calls `wakeup()` as follows:

```c
static void
async_completion(void)
{
    wakeup(wait_chan);
}
```

The routine `get_sleep_lock()` may also be used to protect data shared between the kernel thread that will sleep and the driver's `async_completion()` routine. For example, the driver's top half looks like the following:
(void) get_sleep_lock(wait_chan);
start_async_activity();
activity_count++;
(void) sleep(wait_chan, PRIBIO);

Notice that the incrementing of activity_count is protected by a sleep queue spinlock. When the asynchronous activity completes, the driver’s async_completion() routine must call get_sleep_lock() before it decrements activity_count and calls wakeup().

static void async_completion(void)
{
    lock_t * sleep_lock;
    sleep_lock = get_sleep_lock(wait_chan);
    if (activity_count) {
        activity_count--;
        wakeup(wait_chan);
    }
    spinunlock(sleep_lock);
}

The routine wakeup() has a special provision to allow the sleep queue lock to be acquired and held across a call to wakeup(). After the call to wakeup() in the example above, the sleep queue lock must be unlocked.
Multiprocessing

Synchronization Using `sleep()` and `wakeup()`