HP-UX MultiProcessing

White Paper

Version 1.3

5965-4643
Last modified April 7, 1997

Legal Notices

The information contained within this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Hewlett-Packard shall not be liable for errors contained herein nor for incidental consequential damages in connection with the furnishing, performance, or use of this material.

Warranty. A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

Restricted Rights Legend. Use, duplication, or disclosure by the U.S. Government Department is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 for DOD agencies, and subparagraphs (c) (1) and (c) (2) of the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 for other agencies.

First Edition: April 1997 (HP-UX Release 10.30)

MultiProcessing 5

1 MultiProcessing

Objectives

- Grasp the basic concepts of a multiprocessor (MP) system.
- Gain familiarity with MP data structures.
- Understand locking strategies (spinlocks and semaphores) available in kernel for MP.
- Introduce the kernel interface to control MP events.
- Learn about process scheduling and load balancing in an MP environment.

NOTE

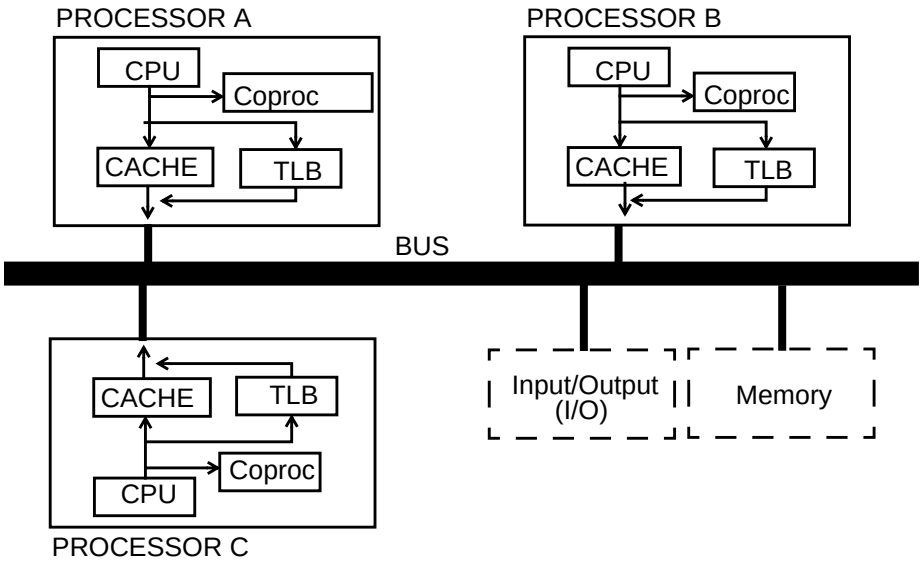
A full study of MP internals is beyond the scope of this text, which does not deal with related issues of interest such as interrupt and powerfail handling, and tuning of other subsystems for a multi-processing environment. Instead, this study introduces multiprocessor data structures and the locking strategies that guarantee consistency across parallel processors.

Throughout this text, multiprocessor systems are referrred to as MP systems. Individual processors will be referred to as both “processors” and SPUs (system processing unit).

MP Overview

A multiprocessor is a system with two or more processing units that act in a controlled and parallel manner to carry out system activity. The figure shows the basic hardware diagram of multiprocessor system with two processors.

Figure 1-1 A sample MP system showing three processors



HP-UX multiprocessing has the following characteristics:

- Two or more processors

  The HP-UX MP implementation supports up to sixteen processors.

- Symmetry

  HP-UX is implemented as a symmetrical multiprocessor operating system. This means that each processor has equal capability to enable any kernel task to execute on any processor in the system. In fact, a thread will often execute on more than one processor during its lifetime. Threads are scheduled in a parallel fashion but this aspect is transparent to users.

- Tight coupling

All processors have uniform access to all of main memory and any I/O device in a shared fashion. This characteristic classifies HP-UX MP as tightly coupled. (By contrast, an implementation where each processor has its own private memory and I/O is known as loosely coupled.)

- Single Integrated Operating System

  A single kernel controls all hardware and software in the HP-UX MP implementation. Locking and synchronization strategies provide the kernel the means of controlling MP events.

- Each processor has its own data structures, including run queues, counters, time-of-day inforamtion, notion of current process and priority.
- Global data structures are protected by semaphores and spinlocks.
- Each processor has its own cache, TLB, registers, interrupts.

NOTE The hardware maintains cache coherency between all processors.

Monarch Selection during Multiprocessor Startup

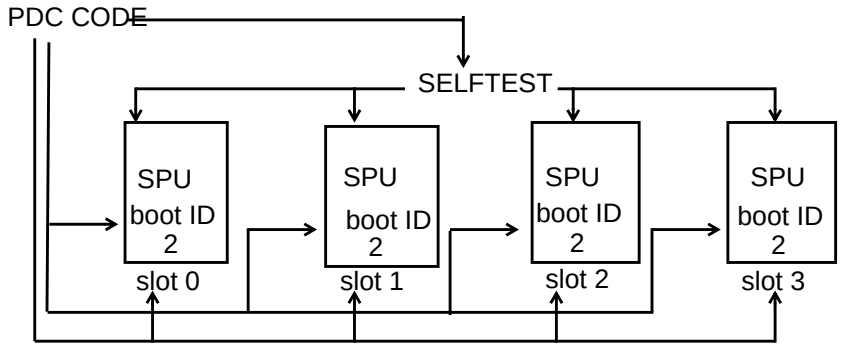
When the system is powered on, after the CPU-level selftests complete, the processor-dependent code (PDC) selects a monarch processor.

As sovereign, the monarch is responsible for all the initial system loader activity; it is the only processor allowed to launch (boot) and enter into the operating system.

The selection of the monarch processor is based on the physical slot location and boot ID. Typically, the processor with the lowest hardware path address (hpa) becomes monarch, although each system has its own arbitration scheme. Later in the initialization process, only the monarch processor walks the bus to determine what other processors are configured and then launches them one at a time to create a multi-processor system.

Figure 1-2, "PDC code selects monarch processor," shows the module layout of a system with four processors, all attached to a central bus. The processor with the highest BOOT_ID value is selected; however as shown, processor BOOT_ID is set to a default value of two by the factory. If more than one processor has the same high BOOT_ID, the processor with the lowest slot number on the bus is selected to be Monarch. In this case, the PDC code is likely to select the processor in slot 0 as the monarch.

Figure 1-2 PDC code selects monarch processor



Monarch selection can be altered by several criteria:

- The PDC routine PDC_CONFIG can configure and deconfigure a given processor based on self-test results. If the processor does not complete self-test, PDC_CONFIG removes it from consideration as potential monarch.
- A user can select a monarch processor using the Boot Console Handler (BCH) code's user interface.
- The Monarch processor takes charge and initializes the bus by sending a CMD.RESET.ST call to each module, thereby excluding the remaining processors.

Once the controlling processor is selected, it invokes its own processor dependent code (PDC) to perform the following:

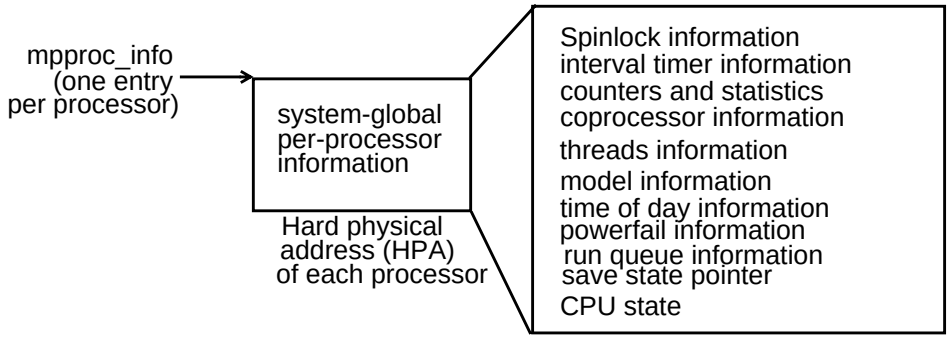
- Initialize all other I/O modules.
- Set up and initialize physical memory for Page 0 of physical memory.
- Load the contents from the PDC ROM to Page 0.
- Use the Boot Console Handler (BCH) to select and initialize the console and boot device.

When the monarch processor is selected, the remaining ("serf") processors go into the "rendezvous code" where all interrupts are cleared. The serf processors wait for a rendezvous interrupt, which happens after the monarch is done with its boot time initialization. If the monarch fails, the serfs can usurp its power (deconfigure the monarch) and force a system reboot, whereupon the arbitration process is repeated and a new monarch selected.

MP Data Structures

The kernel maintains an MP data structure (`typedef struct mpinfo`) that is an array containing system-global per-processor information indexed by SPU number (the hard physical address (HPA) of each processor). The structure and its components are documented in the mp.h header file. The kernel variable `mpproc_info` points to the start of the structure. The kernel variable `mpproc_info[nmpinfo]` points to the end of the array.

Figure 1-3 Scope of information in MP data structures



The general content of each `mpinfo` entry is shown in the table that follows.

Table 1-1 MP information accessed through `mpinfo_t`

MP information	Purpose
spinlock information	Number of spinlocks held, `spl` level at first spinlock taken, pointer to list of spinlocks currently held, current critical spinlock, data on time spent spinning.
Interrupt vector data	Pointer to interrupt vector address (`IVA`), locations of base and top of interrupt stack, pointer to interrupt status word, deferred interrupts.
per-processor counters and statistics `struct mpcntrs`	Array containing: • Numbers of actual reads and writes to file-system blocks, NFS reads and writes, bytes read via NFS, physical reads and writes issued. • Number of times run queue was occupied since bootup; numbers of `execs`, `read/readv()`, `write/writev()`, filename lookups, `inode` fetches, `select()` calls, System V semaphore and message operations, `mux` I/O transfers, raw characters read, characters output since bootup. • Numbers of active process, thread, `inode`, and file entries allocated by the SPU.
coprocessor information `struct coproc_info`	Two 8-bit masks, positioned 0-7; bit 7 corresponds to GR 31. Both elements are `0xC0` if floating point coprocessor is present. • `ccr_present` -to indicate the presence of coprocessor(s). • `ccr_enable` - indicates coprocessor(s) has passed self-test.

MP information	Purpose
Threads information	Current process priority, indication of whether thread is on the processor, pointer to active thread structure, space ID of thread's `uarea`, setting for thread / SPU preemption.
Model information `struct model_info`	Hardware version (CPU type and speed) and ID, software version, ID, and capability, boot ID, architectural revision, potential and current keys. architecture revision (`arch_rev`) identifies PA-RISC level of the CPU: • 0 - PA-RISC 1.0 • 4 - PA-RISC 1.1 • 8 - PA-RISC 2.0
Time of day information `struct tod_info`	Values for normalization and synchronization of interval timer.

MP information	Purpose
Powerfail information `struct pf_info`	Powerfail state, interval timer ticks remaining, and exit state
Run queue information `struct mp_rq`	Includes index into an array of run queue pointers (`bestq`), average run-queue length (`neavg_on_rq`) for load balancing, active locked and unlocked run queues by SPU and type of lock, interval timing and run-queue spinlock pointers.
CPU status	The current state of a processor handling a process is represented by one of the following values: • `MPBLOCK` -- waiting on kernel spinlock • `MPIDLE`-- idle • `MPUSER` -- executing in user mode • `MPSYS` -- executing in system mode • `MPSWAIT` -- waiting on a kernel semaphore

Per-Processor Counters and Statistics

The statistics tracked through the `mpcntrs` structure can be beneficial in comparing the activities of different processors. From this you may be able to determine which processor is handling the majority of NFS traffic or other specific filesystem type activity.

Perhaps the most interesting counters in this structure are the counts for active processes, threads, `inodes`, and files.

Table 1-2 Counters tracked in `struct mpcntrs`

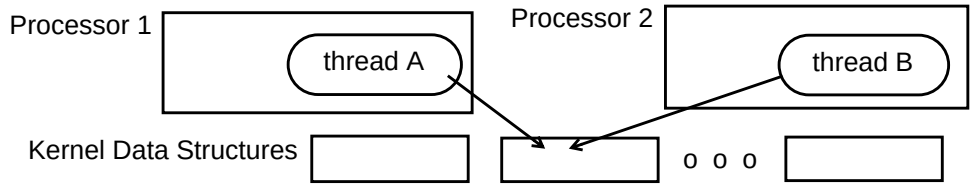
Counter	Purpose
`activeprocs`	Count of the number of processes created by the SPU (number of proc table entries). This count in incremented in `allocproc()` and decremented in `freeproc()`.
`activethreads`	Count of number of threads created by the SPU (number of thread table entries). This count is incremented in `allocthread()` and decremented in `freethread()`.
`activeinodes`	Count of how many `inode`s have been allocated by the SPU (number of `inode` table entries). The count is incremented whenever an inode is removed from the free list by routines such as `ieget()`, and `vx_inoalloc()`.
`activefiles`	Count of the number of file table entries allocated by the SPU. The count is incremented in `falloc()` and decrement whenever a filetable entry is freed by a call to `FPENTRYFREE()`.

These counters track the number of active (in-use) entries for each of the respective kernel tables. These counters must be summed across all running processors to obtain the total number of active entries for each table. The decision as to which processor's `mpinfo` structure to increment or decrement is based on identification of the current processor. If a process is created on SPU A but later terminates while running on SPU B, the `activeprocs` counter will be incremented on SPU A but decremented on SPU B.

Mutual Exclusion for Critical Sections of Code

A principle of synchronization regulates the orderly flow of data into and out of structures and prevents resource contention. Thus, in an MP system, thread A executing on Processor 1 must not contend with thread B executing on Processor 2.

Figure 1-4 Synchronization



Three kinds of critical sections within the kernel require mutual exclusion: *[is it the transition or segue from one to the next or competition of one to the other?]*

- Between two interrupt service routines.
- Between an interrupt service routine and a thread of control.
- Between two threads of control.

In a uniprocessor environment these contentions were easily dealt with: Mutual exclusion was implemented for two interrupt service routines or an interrupt service routine and a thread of control by raising `spl` levels to the highest priority interrupt service routine. To ensure mutual exclusion between threads of control, no thread could be preempted while running in kernel mode.

These protection mechanisms are inadequate for an MP environment. The `spl` routines were local in nature and affected only the interrupt protection level of the calling CPU. Waiting for the current process to reach a safe point, sleep, or exit the kernel failed to give the desired parallelism and made for long, non-preemptable critical durations.

In HP-UX, kernel data structures are protected with software semaphores, locks, and synchronization primitives. Kernel data structures are then divided into sets, with a semaphore or lock guarding

each set. The granularity of the semaphores and locks are empirically determined to minimize blocking of threads of control on the semaphores.

Locking Strategies

Any MP system needs a mechanism for protecting global data structures while allowing multiple processors to execute code concurrently in the system. HP-UX provides for this concurrency through the locking strategies of spinlocks and semaphores.

- Locks provide mutual exclusion in critical sections. Data structures manipulated in these sections are protected by these locks, to prevent errors from occurring if multiple threads of control operate on the data at the same time.

  A lock permits only one thread of control at a time to operate on critical data.

NOTE Every shared kernel data structure is protected by either a spinlock or a semaphore.

- Spinlocks implement a "busy wait condition" for a resource. If a processor attempts to obtain a spinlock being held by another processor, it will wait until the lock is released.

  Spinlocks can be acquired on an interrupt stack. A deadlock can arise, however, if a processor takes an interrupt while holding a spinlock and the interrupt code tries to acquire the same spinlock. To prevent this from occurring, HP-UX requires the spl level to be raised whenever a spinlock is acquired. When the spinlock is released, the prior spl level is reverted to. Once a spinlock is acquired, the spl level should not be lowered within the spinlocked critical section.

  Spinlocks are used to synchronize access to data between multiple processors, and as such, have little value in a uniprocessor system. Within the kernel the `MP_SPINLOCK()` macro checks the uniprocessor flag and returns if not an MP system.

- Kernel semaphores control access through blocking strategies. With blocking semaphores, a processor attempting to acquire a semaphore already held by another processor will put its current thread to sleep and context switch to another task.

  Semaphores are used to provide mutual exclusion or to synchronize access between multiple processes or threads, regardless of how many processors there are.

NOTE Kernel semaphores differ from IPC SystemV semaphores.

In an MP system the decision to use spinlocks or blocking semaphores comes down to a performance issue based on the expected time to busy wait versus the overhead of a process context switch. Additionally, if the lock must be taken while on the Interrrupt Control Stack(ICS), then the process cannot block and must use spinlock. Spinlocks require less overhead than semaphore operations.

Attributes of Spinlocks and Semaphores

The set of data structures protected by a single semaphore or spinlock is defined as a "protection class."

NOTE Every shared kernel data structure is a member of one protection class.

Semaphores have "priority" and "order."

- In this context, priority refers to the scheduling priority to which a process or thread of control is promoted while possessing the semaphore.
- Order refers to a sequential (numeric) arrangement used in detecting and resolving deadlocks.

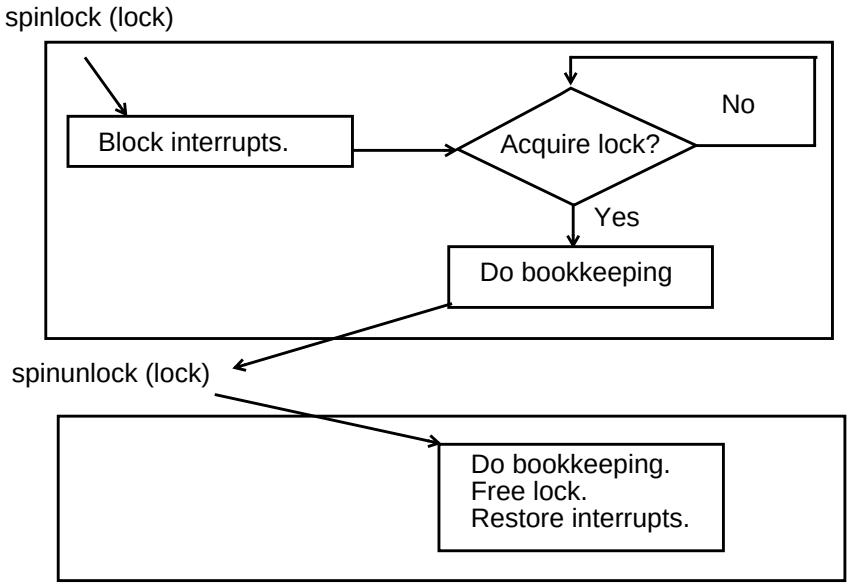
Alpha semaphores (discussed shortly) have an associated lock order (`sa_order`) to prevent a deadlock situation, which can happen if threads on two processors are performing similar operations. The semaphore with the lowest lock order is always locked first. This guarantees that multiple semaphores are locked in the same order by all threads, thus reducing the opportunity for deadlock.

The kernel has assertions to enforce this lock ordering in a debug kernel. Definitions and values of protection class, priority, and order are maintained in the `semglobal.h` header file.

Spinlocks

Spinlocks are at the heart of controlling concurrency within an MP system. Their chief purpose is to protect global data structures by controlling access to critical data. When entering an area of code that modifies a global data structure, the kernel acquires an associated spinlock and then releases it when leaving the affected area of code.

Figure 1-5 Conceptual view of a spinlock



Spinlocks are a more fundamental way of protecting critical sections than semaphores, in that they are used in the construction of the semaphore services; semaphore implementations themselves are critical sections.

```
spinlock (lock);
 [critical section]
spinunlock (lock);
```

The spinlock routines operate on a binary flag of type `lock_t`, to guarantee mutual exclusion of threads of control. The functionality of `spinlock/spinunlock` to raise the `spl` priority to mask out external interrupts and prevent preemption:

```
old_priority = raise_priority (HIGHEST_PRIORITY);
 while (test_and_clear (lock) == 0);
 ...U: lock = 1;
restore_priority (old_priority);
```

To avert deadlocks, spinlock acquisition enforces a simple ordering constraint: Do not attempt to lock a lower or equal-order spinlock to one already held.

Spinlock Rules

The following rules govern use of spinlocks:

- Spinlocks must be held for as short a time as possible (preferably less than the time it takes to make one context switch).
- Spinlocks are a non-blocking primitive. Code protected by spinlocks must not generate traps that can block. (Thus, you may not hold a spinlock across an operation that might take a page fault.)
- Code protected by spinlocks must not cause a context switch. Resources that are never held longer than the time it takes to perform a context switch should be protected with spinlocks. This prevents useless preemption.
- Spinlocks are used to guarantee access to global data structures by a single thread of execution. Thus, they must be acquired prior to the section of code that accesses the global data structures.
- When a lock is unavailable, the spinlock waits until the busy lock is free
- Resources manipulated by an interrupt service routine (ISR) should be protected with a spinlock. ISRs may not block. This applies also to kernel routines that might potentially be called from an ISR.
- Under spinlocks, interrupts are disabled and the thead of control is not allowed to sleep.
- Spinlocks can be acquired on the ICS. It is necessary to prevent interrupts when the top half acquires a spinlock, so that an interrupt does not occur and spin for the same spinlock, thus causing a deadlock.

NOTE

`MP_SPINLOCK` is a macro that checks to see if the code is being executed on an MP system and call `spinlock()` if it is. On a uniprocessor system, there is no need to lock the spinlock since only one thread can execute at a time and it will not sleep until it leaves kernel mode.

Numerous spinlocks are created at the time of kernel initialization with a call to `alloc_spinlock()`, which primarily allocates memory for the spinlock data structure and initializes its fields. The kernel creates these spinlocks from `init_spinlocks()` and by calls to `vm_initlock()` for the VM spinlocks. The table below lists some of the spinlocks allocated at the time of kernel initialization. Other spinlocks are created and destroyed during runtime.

Table 1-3 Spinlocks allocated when kernel is initialized

Type of Spinlock	Names
Process Management	`sched_lock, activeproc_lock, activethread_lock, rpregs_lock, callout_lock, cred_lock`
File System	`file_table_lock, devvp_lock, dnlc_lock, biodone_lock, bbusy_lock, v_count_lock, unrm_table_lock, inode_lock, inode_move_lock, rootvfs_lock, kmio_lock, sysV_msgque_lock, sysV_msghdr_lock, sysV_msgmap_lock, reboot_lock, devices_lock, audit_spinlock`

Type of Spinlock	Names
Networking	`netisr_lock, ntimo_lock, bsdskts_lock, nm_lock`
Virtual Memory Management (VM)	`msem_list_lock, buf_hlist_lock, swap_buf_list_lock, vaslst_lock, text_hash, lost_page_locck, rlistlock, rmap_lock, kmemlock, pswap_lock, rswap_lock, pfdat_lock, pfdat_hash, eq_lock, bcvirt_lock, bcphys_lock, alias_lock, psl_random_lock, mprot_list_lock`
General	`semaphore_log_lock, ioserv_lock, swtrig_lock, time_lock, vmsys_lock, lpmc_log_lock, itmr_sync_lock, itmr_state_lock, pdce_proc_lock, pfail_cntr_lock, printf_lock, io_tree_lock, dma_buflet_lock, space_id_lock, lofs_lo_lock, lofs_lfs_lock, lofs_li_lock`

Spinlock Inlining

To improve the performance of the spinlock code, HP-UX implements a technique called “dynamic inlining.”

A macro is used for select performance-sensitive spinlocks that reserves space for inlining the spinlock instead of simply calling the spinlock function. This is done at compile time. At execution time, if the system has more than one processor, the macro is replaced with inline spinlock code.

For systems with more than one processor, the mutual exclusion algorithm now uses an `LDCW` instruction, which reduces the pathlength of the spinlock routines.

Spinlock Data Structures

HP-UX uses two types of data structures for its spinlock implementation:

- The `lock_t` structure represents a single spinlock.

- Hashed spinlocks are used for locating a spinlock within a pool. (Hashed spinlocks will be explained shortly.)

Spinlock Data Structure (lock_t)

There is one lock_t data structure for every spinlock. The table that follows describes the elements in the structure.

Table 1-4 Elements of the spinlock data structure lock_t

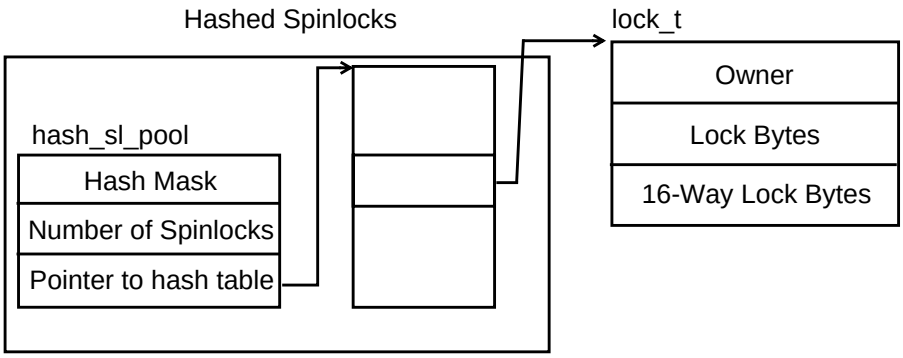
Element	Purpose
sl_lock	Used in the LDCW instruction to acquire the lock. A nonzero value indicates the lock is free.
sl_owner	Pointer to the per-processor data area (&mpinfo[cpunum]) for the processor owning the lock. If the lock is not owned, the value is 0.
sl_flag	A flag that indicates another CPU might want this lock.
sl_next_cpu	The cpu number of the last CPU that acquired the lock under arbitration
sl_pad	Padding to bring lock_t to a reasonable cache line size.

Hashed Spinlocks

A single spinlock works well for a single instance of a global data structure or one that is accessed in synchronously. However, contention occurs when using a single spinlock for a data structure with multiple instances (such as a vnode structure). Conversely, using a single spinlock for each vnode would be overcompensating.

To compromise, HP-UX allocates the capability to use a pool of “hashed spinlocks” that are accessed by a hash function to deal with data structures having multiple instances of individual entries or a group of entries.

Figure 1-6 Hashed spinlocks point to singular spinlock data structures



When developing code, a programmer can choose to do a hash for a hash pool covering any particular requirement (for example, one for `vnodes`, one for `inodes`, etc).

The routine `alloc_h_spinlock_pool()` is used to allocate a pool of hashed spinlocks. From this routine, the kernel calls from `init_hashed_spinlocks()`. A spinlock for a particular instance is then accessed by hashing on the address or some other unique attribute of that instance. You can see spinlocks obtained through this hash pool by a call to `MP_H_SPINLOCK()` in the kernel. Some of the hashed pools currently allocated by the kernel are

- `vnl_h_sl_pool`
- `bio_h_sl_pool`
- `sysv_h_sl_pool`
- `reg_h_sl_pool`
- `io_ports_h_sl_pool`
- `ft_h_sl_pool`

Table 1-5 Key elements in `hash_sl_pool` structure

Hash pool element	Purpose
`n_hash_spinlocks`	Contains pointers to `lock_t` structures
`**hash_sl_table`	Points to an array of size `n_hash_spinlocks`

The hash functions return an index into this array of pointers.

Regardless of whether the spinlock data structure is accessed directly or through a hash table, the acquisition details to be discussed next are the same.

Spinlock Arbitration

To ensure that no processor is kept waiting indefinitely for a spinlock, round-robin arbitration using two modules takes place.

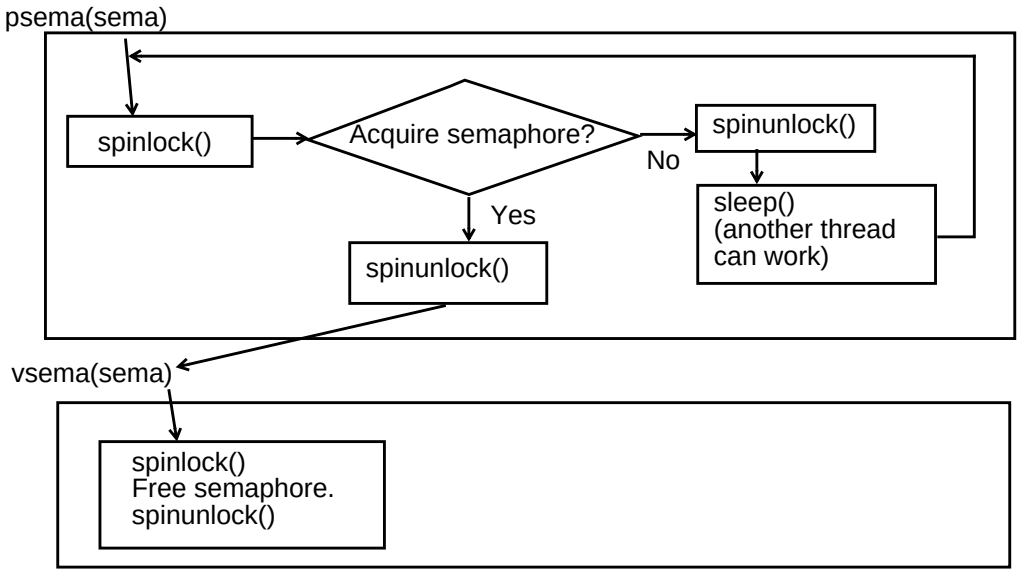
Table 1-6 Modules for spinlock arbitration

Module	Purpose
`wait_for_lock()`	Waits until a spinlock is acquired or a timeout occurs. Puts the lock address into a table indexed by CPU number. Sets a flag to indicate that there are CPUs waiting for the lock.
`su_waiters()`	Called from `spinunlock` when the `sl_flag` is set. Either releases the lock or passes it to another processor.

Semaphores

Semaphores are routines that ensure orderly access to regions of code. Like spinlocks, semaphores guard kernel data structures by controlling access to regions of code associated with a set of data structures. Unlike spinlocks, semaphores require the waiting thread to relinquish the CPU while awaiting the lock. Semaphores are implemented using a `swtch()` to allow another thread to run.

Figure 1-7 Conceptual view of a semaphore



Semaphores serve two functions -- mutual exclusion and synchronization. Mutual-exclusion semaphores protect data and are further classified by their degree of restrictiveness.

Mutual-Exclusion Semaphores

Mutual-exclusion semaphores provide mutually exclusive access to regions of code that are associated with a set of data structures.

In a mutual-exclusion semaphore, a processor attempting to acquire a semaphore already held by another processor puts its current thread of control to sleep and switches to another. It is assumed that the expected time duration the thread will wait while the lock is busy will be much greater than the overhead of a process switch.

The kernel makes available two types of mutual-exclusion semaphores:

- Alpha semaphores, which must be released when a thread of control sleeps.

  The alpha semaphore cannot be held during sleep because it is used to protect data structures that must be consistent at the time of context switch. This applies, for example, to the fields in structures that describe the process state of a thread of control.

  A broadly encompassing alpha semaphore, called an empire semaphore, protects a collection of data structures.

- Beta semaphores, which a thread of control may hold while sleeping.

  A beta semaphore can be held while sleeping because the protected data structures need not be consistent at the time of context switch. An example of this is the page frame lock during a page fault. The resource must remain locked during the resolution of the fault but the thread yields the processor while its page is brought in from memory.

Synchronization Semaphores

Synchronization semaphores signal events rather than block access to data structures and are used when events are awaited. They synchronize a thread with other threads and external events. The table that follows describes some of these differences in practice.

Comparison of Blocking vs Synchronization Semaphore

Data protection (blocking or mutual exclusion) semaphores and synchronization semaphores differ in four ways.

- Locking and unlocking operations
- Signal handling
- Initialization
- Count

Table 1-7 Differences between mutual-exclusion and synchronization semaphores

Mutual-Exclusion (Alpha/Beta)	Synchronization
Lock and unlock operations are always performed by the same thread.	Lock and unlock operations performed by different threads. Synchronization is provided by one thread doing a lock, which causes it to block. While blocked, the thread sleeps until another thread does an unlock, causing the locking thread to awaken. This mechanism enables you to use semaphores to cause a thread to wait on an event from another thread
Thread blocked on semaphore cannot be awakened by a signal. Signals are deferred until the thread acquires the semaphore, on the assumption that semaphores are not held for long durations.	Threads blocked on a synchronization semaphore have three options: • Signals can be caught and handled. • Signals can be deferred until the semaphore operation is complete. • Signals can be handled as though the code were unsemaphored.

Sample Semaphores

The table below shows some of the semaphores the kernel creates at initialization time from `init_semaphores()` and `realmain()`. They are listed only by type and name. You can look at the kernel source to observe how each are used.

Table 1-8 A Sampling of semaphores

Type of semaphore	Sample

Alpha Semaphores	`filesys_sema`
	`pm_sema`
	`up_io_sema`
	`mdisc2_sema`
	`vmsys_sema`

Beta Semaphores	`msem_betasem` `iomap_betasem`
Synchronization Semaphores	`runin` `runout`

Empire Semaphores

Some alpha semaphores are classified as empire semaphores, because they protect data structures for an entire subsystem. Empire semaphores are locked when any of the structures within the set must be modified. Because they control access to an entire subsystem, empire semaphores are used to serialize operations within the subsystem. For example, the filesystem empire (`filesys_sema`) is locked when calling `sync()`, to prevent other threads from invalidating pages of data that are being flushed from cache.

Empire semaphores are acquired and released with `pxsema()/vxsema()` calls.

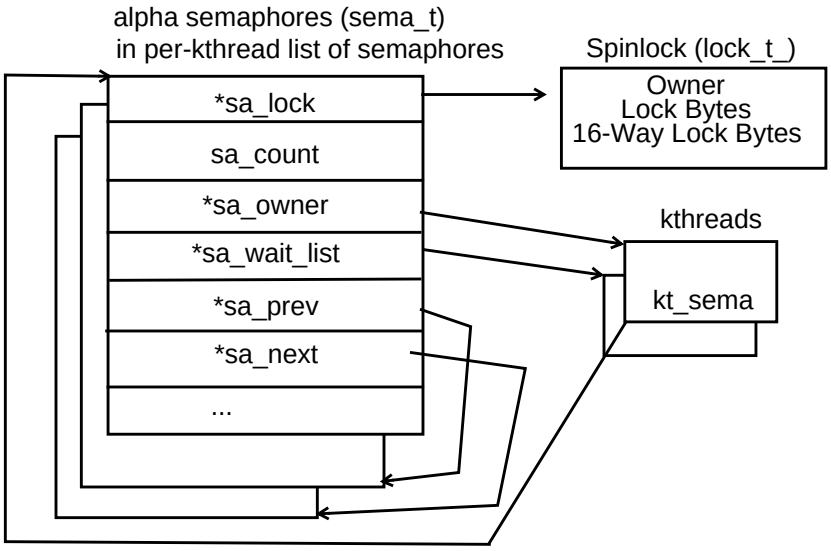
MP Safety

The `up_io_sema` empire semaphore provides “single threading” (access control) for I/O drivers that are not MP safe. An MP-safe driver is one that synchronizes multiple accesses to code and structures, so that more than one instance of the driver may be active at any given time without contention.

Alpha Semaphore Structures

Alpha semaphores are defined as `type sema_t`. The following figure shows its principal elements and how it is implemented in relation with other kernel structures.

Figure 1-8 Alpha semaphore vs-a-vs spinlock and kthreads



A spinlock is used to protect the data structures that implement the semaphore.

Table 1-9 Principal elements of `struct sema (sema_t)`

Element	Purpose
`*sa_lock`	Pointer to spinlock protecting the semaphore.
`sa_count`	Value of semaphore count, which indicates whether the semaphore is available.
`*sa_owner`	Pointer to kernel thread that owns a lock on the semaphore.
`*sa_wait_list`	Pointer to head of `kthread` waiting on the semaphore.

Element	Purpose
`*sa_prev, *sa_next`	Previous and next semaphore in per-kthread list; used to link semaphores owned by a given thread together.
`sa_order`	Deadlock protocol order for semaphore.
`sa_priority`	Priority of a mutual exclusion semaphore.
`sa_missers`	Array of processors used for semaphore arbitration.

Performance is a key consideration for use of alpha semaphores. To prevent "starvation" of code, the following algorithm governs their use:

When a CPU misses on an alpha semaphore, the CPU's number is put in an array (`sa_missers`), indexed according to the priority of that processor for the semaphore. The processor with the lowest entry in the array is favored. This arrangement ensures fair access to the semaphore.

A value called `asema_max_ignore` limits the number of times a semaphore is checked and found unavailable. Once this value is exceeded, arbitration code (`asema_available()`) ensures that the CPU does not get starved for a semaphore.

Alpha Semaphore Services

The kernel uses several different kinds of routines to manage alpha semaphores:

- Initialize an alpha semaphore.
- Acquire/release a semphore while adjust priority.
- Acquire/release semaphore across empires.
- Bind/unbind semaphore to a kernel thread.
- Test for whether a kernel thread owns semaphores.
- Arbitrate for an alpha semaphore.

NOTE The services to acquire a semaphore begin with the letter P; those to release a semaphore begin with the letter V. These derive from the Dutch words Proberen, meaning “to test” and Verhogen, meaning “to increment.”

Acquire and Release an Alpha Semaphore

Table 1-10 Acquisition and release of an alpha semaphore

External Interface	Purpose
`initsema (semaphore,value,priority, order)`	Initialize a mutual-exclusion semaphore • Must be called before a semaphore is used. • Must not be called when a semaphore is actively being used by the kernel.

`psema(semaphore)`
`vsema(semaphore)`
	Acquire, release a mutual exclusion semaphore.
	• `psema()` acquires the semaphore by decrementing the semaphore count.
	  • If the count is non-negative, the thread acquires the semaphore.
	  • If the count is negative, the priority of the calling thread is raised and the thread blocks until the semaphore is available.
	• `vsema()` releases the semaphore; it does not adjust the priority, but delays this until the process leaves the kernel.

`pxsema(semphore, save)` `vxsema(semaphore, save)`	Acquire, release semaphores when crossing into and out of another empire.

Bind and Unbind a Semaphore to a Kernel Thread

These routines serve primarily to maintain the `kt_sema` field in the `kthread` structure. This field keeps track of currently held alpha semaphores.

As a thread acquires a semaphore, `sema_add()` links semaphores together through these field. You can obtain all of the semaphores owned by a thread by following `kthread->kt_sema->sa_next`.

Semaphores are bound to threads to maintain the list of semaphores held when a thread goes to sleep. All bound semaphores are released at that time and by following this list, they can be reaquired when awakened.

Table 1-11 Bind and unbind an alpha semaphore

Internal function	Purpose
`sema_add (kthread, semaphore)`	Add a reference to a newly acquired semaphore into the thread's `kthread` structure. Update the `kthread`'s priority.
`sema_delete (kthread, semaphore)`	Remove a reference to a thread's `kthread` structure and recompute the `kthread`'s priority.

Test for Ownership of Semaphore

Table 1-12 Tests for ownership of an alpha semaphore

Function	Purpose
`owns_sema (semaphore)`	Returns true if the current thread owns the semaphore. The routine compares `semaphore->sa_owner` with `u.u_kthreadp`.
`kthread_owns_semas (*kthread,sema)`	Returns true if a `kthread` owns one or more semaphores; otherwise returns false.

Wait for an Alpha Semaphore

Numerous routines govern the kernel's decision about whether to switch to a thread of control that needs a semaphore.

Table 1-13 Tests for whether to switch thread of control

Function	Purpose
`asema_available (semaphore)`	Routine determines whether the kernel should switch to a process that needs a semaphore, based on performance and priority.
`asema_miss_ins (semaphore,CPU)`	Called after a `psema` miss to insert the CPU number into the miss table.
`asema_miss_del (semaphore,CPU)`	Remove the CPU entry from the miss table, recompute priorities.
`asema_miss_pri (semaphore,CPU)`	Find prority of CPU's earliest miss.
`psema_choose_turn (semaphore)`	Determine if CPU deserves to take its turn.
`psema_spin_[1|n] (semaphore)`	Wait on a locked semaphore. The thread spins cycles depending on whether it is the CPU's turn.
`psema_switch_[1|n] (caller)`	Spin for a semaphore without arbitrating.

Beta Semaphores

In some instances the rules governing alpha semaphores are too strict to meet the needs of the kernel. Another class of semaphores exist, known as beta semaphores.

NOTE Unlike alpha semaphores, beta semaphores can be held while a process sleeps.

Beta semaphores are created in the kernel by a call to `b_initsema()`. Beta semaphores have services similar to alpha semaphore services. The following table describes the principal kernel interface routines for beta semaphore operations.

Table 1-14 Interface routines used for beta semaphore operations

Routine	Purpose
`b_initsema()`	Create a beta semaphore, add it to the hash table, link it to the global list of semaphores.
`b_termsema()`	Unlink beta semaphore from hash chain,
`b_psema()`	Acquire the semaphore and possibly sleep if not available. Operative assertions: beta semaphore is valid, no spinlocks are held, interrupts are disabled, not in interrupt context.

Routine	Purpose
`b_cpsema()`	Acquire semaphore and return 0 if available. If not available fail and return 1. Operative assertions: Beta semaphore is valid, interrupts are enabledwhen not on the boot path.
`b_vsema()`	Release the semaphore. Operative assertions: beta semaphore is valid, interrupts are enabled, semaphore is locked and allowed to unlock.
`b_disowns_sema()`	Returns true if current `kthread` does not own the specified beta semaphore.

Beta semaphores use a hash table to access the associated spinlock and wait list information (linked list of `kthreads`).

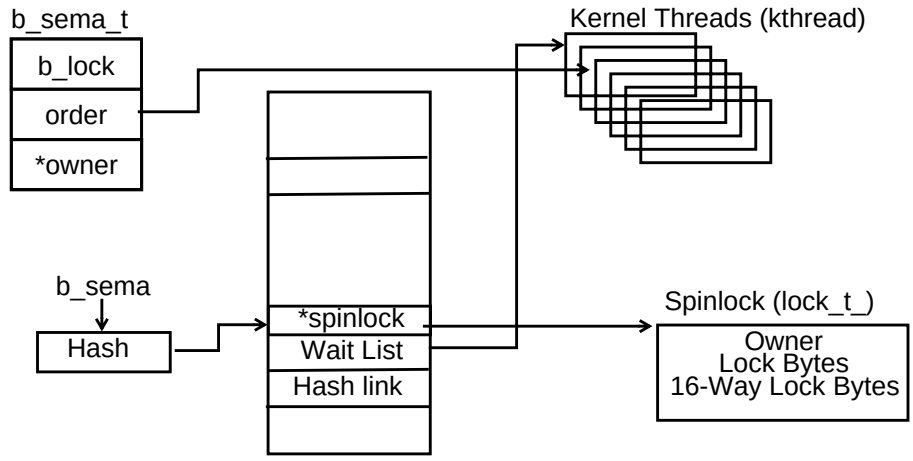
A `kthread` at the head of a semaphore's wait queue is allowed to be awakened and yet miss the semaphore a maximum of `BETA_MISS_LIMIT` times. Other executing code is allowed to acquire the semaphore between the time the semaphore is unlocked by the V operation and the time the awakened `kthread` can execute and lock it. If the miss limit is reached, the semaphore is passed to the waiting `kthread`.

The number of misses the `kthread` at the head of the semaphore's wait queue has taken is maintained in the `kthread`'s `proc` structure. Each V operation on the semaphore will awaken the `kthread` at the head of the wait queue and unlock the semaphore if the miss limit has not been reached. If the miss limit is reached, the V operation will awaken the `kthread` at the end of the wait queue but will not unlock the semaphore, preventing other code from acquiring the semaphore. The awaken `kthread` notices that the semaphore ownership has been passed to it. This is indicated by the miss count being equal to `BETA_MISS_MAX`.

Beta Semaphore Structures

The beta semaphore itself contains only the lock and owner information. Both beta semaphore and its hash table are definedin `sem_beta.h`.

Figure 1-9 Beta semaphore structures



Beta Semaphore Type Definition

The beta semaphore is defined as `typedef b_sema_t` (also defined as `vm_sema_t`) and consists of the three fields.

Table 1-15 elements in `struct b_sema`

Element	Purpose
`b_lock`	Lock state for the semaphore, which may have one for the following values: • 0 = Available • 1 = `SEM_LOCKED` (Semaphore is locked) • 2 = `SEM_WANT` (Semaphore is locked but a thread is waiting on it).
`b_order`	Indicator of what order the semaphore should be locking in.
`*b_owner`	Pointer to the `kthread` structure of the thread of control claiming the semaphore. (This is the only thread information in the beta semaphore structure.)

Note, the `b_lock` field is not a spinlock. The spinlock guarding the beta semaphore is in the hash table.

Beta Semaphore Hash Table

The address of the beta semaphore is indexed into the beta semaphore hash table (`bh_sema_t`) to obtain the spinlock and waiter information.

Table 1-16 elements in `struct bh_sema_t`

Element	Purpose
`*beta_spinlock`	Pointer to the spinlock (`type lock_t`)
`*fp, *bp`	Pointers to the `struct kthread` that comprise a wait list for the beta semaphore. The waiters are linked together using the `kthread.kt_wait_list` and `kthread.kt_rwait_list` fields in the thread structure.
`*link`	Pointer to the link list of beta semaphores.

Performance Considerations and Locking

Consider the following when designing your code to run on a multiprocessing system:

- Spinlocks execute faster than semaphores when they do get the lock.
- Spinlocks waste CPU time by spinning if they cannot get the lock.
- There is a trade-off of efficiency when using semaphores, depending on how long a lock is held before you get it:
  - Semaphores might waste CPU time by switching to another process if they cannot get the lock, because if the lock had been free, the switch would have been unnecessary.
  - Semaphores might save CPU time by switching to another process if they cannot get the lock, because one process can do useful work while the process is waiting for the lock.

  If the lock will be held for a long time (compared to a context switch), switching is preferable; but if held briefly, spinning might be better.
- Because spinlocks are busy waiting, they can immediately get the lock when it comes free.
- With semaphores the waiting process must be context-switched in its sleep state. This represents a high latency in getting the lock.

Deadlocks

Consider the following example:

Table 1-17 Sample deadlock situation

Processor 0	Processor 1
`spinlock(lockA)`	`spinlock(lockB)`
`spinlock(lockB)`	`spinlock(lockA)`

Processor 0	Processor 1
[do work]	[do work]
`spinunlock(lockB);`	`spinunlock(lockA);`
`spinunlock(lockA);`	`spinunlock(lockB);`

Deadlocks occur when two processors (or processes or threads of control) have locked resources in different orders, and each has something needed by the other. As a result, they wait for each other to relinquish what they need. There can be complex chains of these dependencies amongs multiple processors and processes.

The sample code works most of the time. But when both processors fall through their respective code at the same time, a problem occurs. When machines execute 100 million instructions per second (or more), such coincidences happen all too frequently, however.

Ordering Strategy for Deadlock Avoidance

- Locks are always locked in the same order.
- Each lock is given its own order (a positive integer).
- Instrumented kernels are run to ensure that locks are always taken in the correct order.

Maintaining an ordering strategy guarantees that each locking sequence is done in just one order, no matter where the code is executing.

Processor Scheduling

One of the biggest challenges in a multiprocessing environment is to distribute evenly the work across available processors. When a process is created, it is set to run initially on the same SPU as the parent, because a forked process is likely to use some of the same context as the parent. By launching on the same processor, the system takes advantage of previously cached data and avoids cache coherency performance issues.

In a multiprocessor environment, each SPU has a separate run queue. Once a thread is put on a run queue (with `setrq()`) for a certain processor it remains there until removed with `remrq()`. When a process is ready to run, the processor to which it is scheduled is based on the `kthread.kt_spu_wanted` field.

Of major concern is to keep the relative load balanced among processors. To do this, each iteration of `schedcpu()` calls the routine `mp_spu_balance()`.

Additionally, any spu in an idle state may attempt to steal threads from other processors. This is done by the kernel routine `find_thread_other_spu()`.