

# **Common Misconfigured HP-UX Resources**

**Mark Ray  
Steven Albert  
Jan Weaver**  
Hewlett Packard Company

## **Common Misconfigured HP-UX Resources and How they Impact Your System**

Physical memory is a finite resource. It is also a shared resource, with many processes attempting to access this finite resource. Not only do processes need memory in order to run, the HP-UX operating system (or kernel) also needs spaces for its critical resources and tables. Some of these resources are static (do not change in size) and some are dynamic. Many of these resources can be configured to be a certain size or configured to be limited by a certain value.

From time to time Hewlett Packard support receives customer calls regarding the amount of memory the system is using to perform system related tasks. While there are many possible system resources that can take up memory, this presentation attempts to identify some of the common misconfigured HP-UX resources and how they impact your system.

There are many reasons why a HP-UX resource may be misconfigured. The most common reason is that customer environments are unique. There is no one set of tunables that is best for all systems. Understanding how these resources are managed and how they impact memory utilization are keys a successful configuration.

The resources covered are:

- The HFS Inode Cache
- The HP-UX Buffer Cache
- The JFS Inode Cache
- The JFS 3.5 Metadata Buffer Cache
- Semaphores Tables

The presentation will answer some of the following questions regarding the resources:

- What is the purpose of the resource?
- How is the resource managed?
- How much memory does each resource require?
- How can the resource be tuned?
- Are there any guidelines to consider when configuring the resource?

# **The HFS Inode Cache**

**Mark Ray**

Global Solutions Engineering  
Hewlett Packard Company

## The HFS Inode Cache

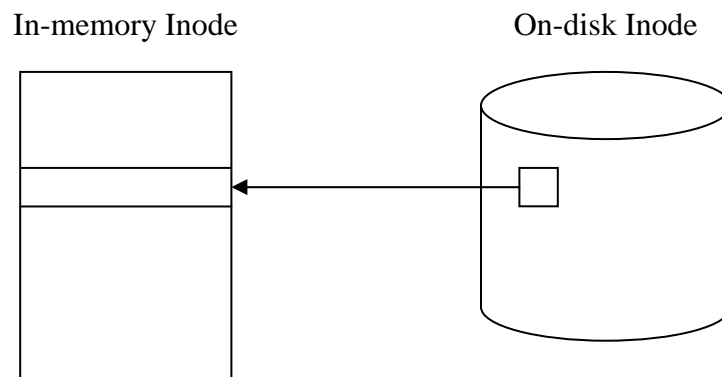
With the introduction of JFS, many systems use HFS for the boot file system (/stand) only. Since the JFS inode cache is managed separately from the HFS inode cache, you may need to adjust the size of your HFS inode cache.

This paper will address the following questions regarding the HFS inode cache:

1. What is an inode cache?
2. How is the HFS inode cache managed?
3. How much memory is required for the HFS inode cache?
4. What dependencies are there on the HFS inode cache?
5. Are there any guidelines for configuring the HFS inode cache?

### What is an Inode Cache?

An inode cache is simply a holding location for inodes from disk. Each inode in memory is a superset of data which contains the inode from disk. The disk inode stores information for each file such as the file type, permissions, timestamps, size of file, number of blocks, and block map. The in-memory inode stores the on-disk inode information along with overhead used to manage the inode in memory. This information includes pointers to other structures, pointers used to maintain linked lists, the inode number, lock primitives, etc



The inode cache is simply the collection of the in-memory inodes with its various linked lists used to manage the inodes. Once the inode is brought into memory, subsequent access to the inode can be done through memory without having to read or write it to disk.

One inode cache entry must exist for every file that is opened on the system. If the inode table fills up with active inodes, the following error will be seen on the console and in the syslog file and the `open()` system call will fail:

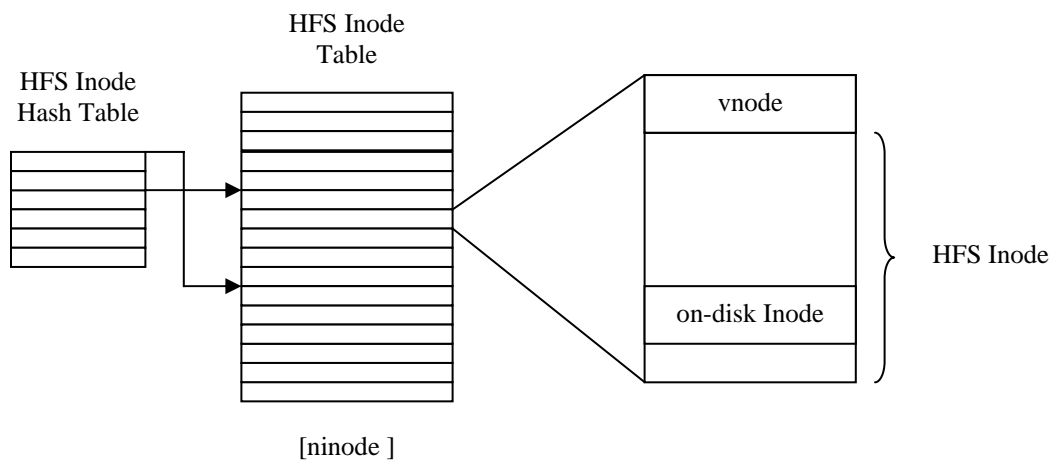
```
inode table is full
```

Once the last close is done on a file, the inode will be put on a free list. It is not necessarily removed from the cache. So the inode cache will likely contain some files that are closed, so that if the file is re-opened, a disk access will not occur as the inode is already in memory.

## The HFS Inode Cache is a Static Cache

The HFS inode cache is a statically sized table built during bootup. It is simply an array of in-memory HFS inodes, which is hashed for quick lookups into the table. Inodes are either in use, or on the free list. It is common for the HFS inode cache to appear full, since even closed files are maintained in the HFS inode cache.

The size of the cache is determined by the tunable **ninode**. Note that the tunable only sizes the HFS inode cache, and does not affect the JFS inode cache. The tunable also sizes the HFS inode hash table, which is the previous “power of 2” based on **ninode**. For example, if **ninode** is configured for 1500 inodes, then the hash table will have 1024 hash entries (since 1024 is the previous power of 2 to 1500).



When an inode lookup is performed, the device and inode number are used to hash into the inode hash table. From the hash header, a linked list of inodes that hash to the same hash header is analyzed to see if the desired inode is found. If the desired inode is found, we simply use the inode in memory.

If the desired inode for the file is not found, we simply reuse the least recently used inode, which is the first inode on the free list. The inode information on disk is then read into this newly obtained inode. If there are no inodes on the free list, the system call will fail with the “inode table is full” message.

## Determining the Memory Cost of the HFS Inode Cache

We can determine the cost of the HFS inode cache table by identifying the size of the inode hash table, and the actual cache of inodes. Note the inode cache is simply an array of vnode/inode structures (the vnode is the Virtual File System layer part). So both the size of the vnode and the

size of the inode must be considered. The table below can be used to identify the number of bytes needed for each structure:

|            | 10.20 | 11.0 32-bit | 11.0 64-bit | 11.11 32-bit | 11.11 64-bit |
|------------|-------|-------------|-------------|--------------|--------------|
| vnode      | 100   | 100         | 176         | 100          | 176          |
| inode      | 336   | 336         | 488         | 367          | 496          |
| hash entry | 8     | 8           | 16          | 8            | 16           |

Table 1: Number of bytes per entry

Using the table above, the memory cost of the HFS inode cache can be calculated. For example, if `ninode` is configured at 10,000 on an 11.11 64-bit system, the system would use 6563 Kb for the HFS inode table  $((496+176)*10000)$ , and 128 Kb for the HFS hash table (previous power of 2  $(8192) * 16$ ).

## The HFS Inode Cache and the Directory Name Lookup Cache (DNLC)

The `ninode` tunable used to configure the HFS inode cache size can also impact the size of the Directory Name Lookup Cache (DNLC). The size of the DNLC used to be entirely dependent on the value of `ninode`. However, this caused some problems since the DNLC was used by HFS and non-HFS file systems (such as JFS). For example, even if the `/stand` file system was the only HFS file system, a customer would have to configure `ninode` very large in order to get a large enough DNLC to handle the JFS files.

The dependency on the `ninode` tunable is reduced with the introduction of 2 tunables:

**`ncsize`** – Introduced with `PHKL_18335` on 10.20. Determines the size of the Directory Name Lookup Cache independent of `ninode`.

**`vx_ncsize`** – Introduced in 11.0. Used with `ncsize` to determine the overall size of the Directory Name Lookup Cache .

While you can tune `ncsize` independently of `ninode`, the default value is still dependent on `ninode` and is calculated as follows:

$$(NINODE+VX_NCSIZE) + (8*DNLC_HASH_LOCKS)$$

Beginning with JFS 3.5 on 11.11, the DNLC entries for JFS files are maintained in a separate JFS DNLC.

## Configuring your HFS Inode Cache

The default size of the HFS inode cache is based on a number of tunables, most notably, nproc:

$$((NPROC+16+MAXUSERS)+32+(2*NPTY))$$

However, some systems will need to configure a larger HFS inode cache, and some systems will need to configure a smaller cache. The first thing you need to remember is how many HFS file systems you presently have. If the boot filesystem (/stand) is your only HFS file system, then ninode can be configured very low (maybe 200). If you configure a small HFS inode cache, be sure the DNLC is configured appropriately for other file system types.

At a minimum, the HFS inode cache needs to be configured to be large enough to hold all the HFS files that are open at any given instance in time. For example, you may have 200,000 HFS inodes, but only 1000 are simultaneously opened at the same time.

If you use mostly HFS files systems, the default value of ninode is good for many systems. However, if the system is used as a file server, with random files opened repeatedly (for example, a web server or NFS server), then you may consider configuring a larger ninode value (perhaps 40,000-80,000). When configuring a large ninode cache, remember the memory resources that will need to be allocated.

# **The HP-UX Buffer Cache**

**Jan Weaver**  
Hewlett Packard Company

## The HP-UX Buffer Cache

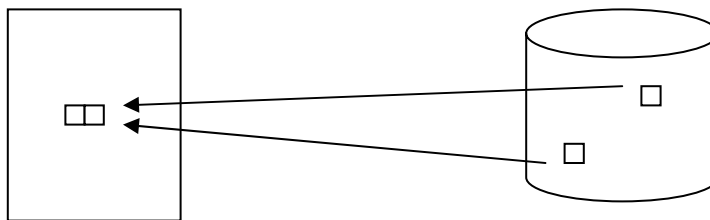
The HP-UX buffer cache configuration can be confusing and the HP-UX buffer cache is frequently over or under configured. Understanding how the HP-UX buffer cache is maintained and used can help to determine the proper configuration in your application environment.

This paper will address the following questions regarding the HP-UX buffer cache:

1. What is the buffer cache?
2. How does a static buffer cache differ from a dynamic buffer cache?
3. How does the buffer cache work?
4. How much memory is required for the buffer cache and its related structures?
5. What are the advantages and disadvantages of using the buffer cache?
6. Can the buffer cache be bypassed?
7. Are there any guidelines for configuring the buffer cache?

### What is the Buffer Cache?

The buffer cache is an area of memory where pages from the secondary storage devices are stored. The buffer cache is used to reduce access to the secondary storage devices by storing frequently accessed pages in memory.



Once the file data is in memory, subsequent access can be performed in memory, without the need to access the secondary storage device.

### Static Versus Dynamic Buffer Cache

By default, the buffer cache is dynamic, as it can expand or shrink in size over time. A dynamic buffer cache is tuned by setting the kernel tunable parameters **nbuf** and **bufpages** to zero, and by setting the minimum and maximum ranges as a percentage of memory, **dbc\_min\_pct** and **dbc\_max\_pct** respectively. The default values are:

|                    |           |
|--------------------|-----------|
| <b>dbc_max_pct</b> | <b>50</b> |
| <b>dbc_min_pct</b> | <b>5</b>  |

Note that the **dbc\_min\_pct** cannot be less than 2 and **dbc\_max\_pct** cannot be greater than 90.

When the system is initially booted, the system allocates `dbc_min_pct` (default 5%) of memory for buffer pages (each page is 4096 bytes). The system also allocates 1 buffer header for every 2 buffer pages. The size of the buffer cache will grow as new pages are brought in from disk. The buffer cache can expand very rapidly, so that it uses the maximum percentage of memory specified by `dbc_max_pct`. A large file copy and backups are operations that can cause the buffer cache to quickly reach its maximum size. While the buffer cache expands quickly, it decreases in size only when there is memory pressure.

A static buffer cache can be configured to a fixed size by setting either `nbuf` or `bufpages`. Setting `nbuf` specifies the number of buffer headers that should be allocated. Two buffer pages are allocated for each buffer header for a total of `nbuf*2` buffer pages. If the `bufpages` kernel parameter is set and `nbuf` is 0, then the number of buffer pages is set to `bufpages` and one buffer header is allocated for every 2 buffer pages for a total of `bufpages/2` buffer headers. If both `nbuf` and `bufpages` are set, `nbuf` is used to size the buffer cache.

A static buffer cache can also be configured by setting the `dbc_min_pct` and `dbc_max_pct` to the same value.

Trade-offs are associated with either a static or dynamic buffer cache. If memory pressure exists, a static buffer cache cannot be reduced, potentially causing more important pages to be swapped out or processes deactivated. In contrast, some overhead exists in managing the dynamic buffer cache, such as the dynamic allocation of the buffers and managing the buffer cache address map or buffer cache virtual bitmap (the `bufmap` and `bcvmap` will be discussed later in more detail). Also, a dynamic buffer cache expands very rapidly, but contracts very slowly and only when memory pressure exists.

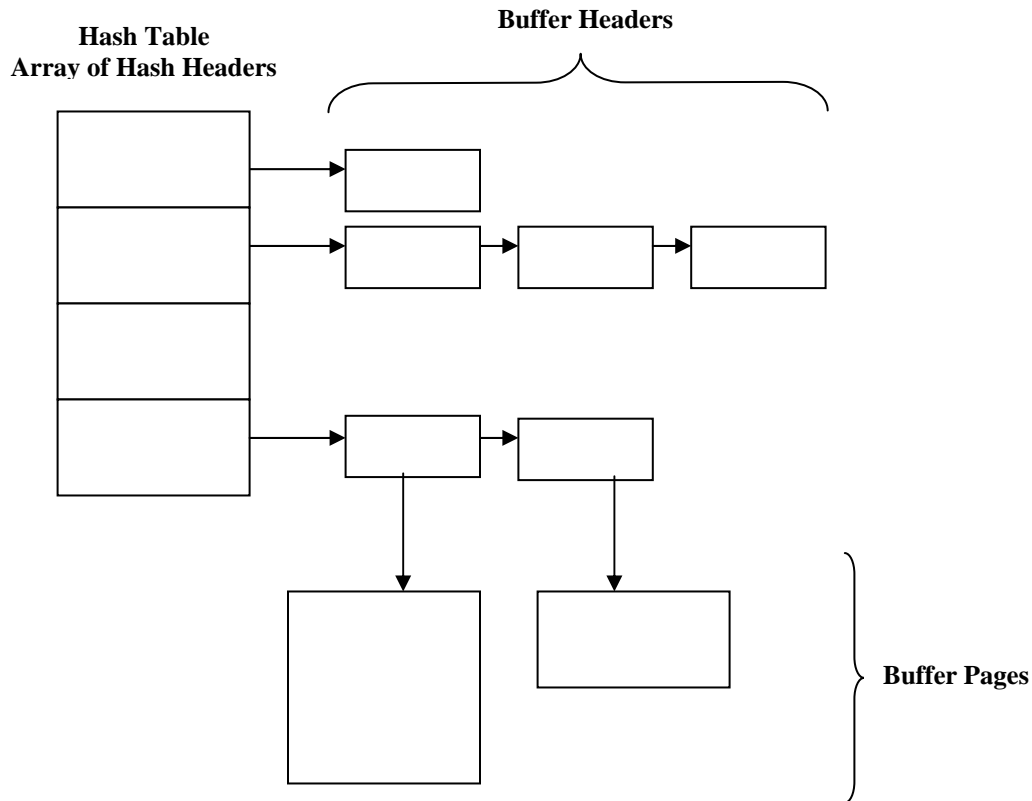
## How the Buffer Cache Works

The main parts of the buffer cache are the Buffer Cache Hash Table, Buffer Headers, and Buffer Pages. At a minimum, we allocate 1 page per buffer header for the actual buffer, even if the buffer is only 1kb in size. The maximum buffer size is 64kb. When data from disk is needed, we use the block device (specifically the vnode address of the block device) and the block number to calculate a hash index into the Buffer Cache Hash Table, which is an array of Buffer Cache Hash Headers. The Buffer Cache Hash Header will point to a linked list of buffers whose block device and block number hash to the same hash header.

If an attempt to access a block from a device and it does not exist in the appropriate hash chain, a buffer cache miss occurs and one of 2 actions will occur:

1. A new buffer is allocated (if a dynamic buffer cache is used). Data must be read in from disk.
2. An existing buffer is reused (if a static buffer cache is used or the buffer cache is already at the `dbc_max_pct`) and data must be read in from disk. The buffer reused is either buffer that has been invalidated (i.e. file has been removed or file system has been unmounted), or the buffer has not been accessed recently.

However, if the desired buffer is found in the buffer cache, the data can be accessed without accessing the disk.



**Diagram: The HP-UX Buffer Cache**

Note that buffers remain in the buffer cache even after a file is closed. Thus, if a file is re-opened a short time later, the buffers may still be available in the buffer cache. For example, if your buffer cache is 500 MB in size, and you perform a `grep(1)` on a 100MB file, each data block will need to be read into the buffer cache as the file is scanned. However, if you perform a subsequent `grep(1)` on the same file, the file should be accessed via the buffer cache without accessing the disk device even though the file was closed after the first `grep(1)`.

## Buffer Cache and Memory

So how much memory does the buffer cache take? This seems like a simple question. If you configure your buffer cache to a dynamic maximum of 10% of physical memory on a system with 12 GB of memory, then the maximum size of the buffer cache is 1.2GB of memory. However, this only represents the Buffer Pages. Other structures used to manage the buffer cache are not accounted for in this total. These other structures include:

- Buffer Headers.
- Buffer Cache Hash Table
- Buffer Hash Locks
- Buffer Cache Address Map / Buffer Cache Virtual Map

## Buffer Headers

Each buffer in the buffer cache needs a header structure that defines what the block represents, how it can be used and how it is linked. Buffer headers are approximately 600 bytes in length. If the buffer cache is fixed, nbuf buffer headers are allocated at system initialization. If the buffer cache is dynamic, buffer headers are allocated dynamically as needed. When more buffer headers are needed a page of memory is allocated and carved up into as many buffer headers as will fit in one 4K page.

## Buffer Cache Hash Table

Blocks in the buffer cache are hashed so that they can be accessed quickly. The number of hash entries is computed at boot time and is  $\frac{1}{4}$  of the number of free memory pages rounded up to the nearest power of two. So a system with 12 GB of memory will have approximately 1 million hash table entries regardless of the buffer cache configuration. A page is 4096 bytes, so 12GB represents 3145728 pages. One quarter of that is 786432 rounded up to the next power of two is 1,048,576. Each hash header is 40 bytes, so a 12GB system would have 40MB of memory allocated to Buffer Cache Hash Table.

## Buffer Cache Hash Locks

Instead of having a lock for each Buffer Cache Hash Header, a single lock may be used for multiple Hash Headers. This reduces the amount of memory needed for hash locks. Table 2 below specifies how many hash chains each hash lock covers.

## Buffer Cache Address Map / Buffer Cache Virtual Map

In HP-UX 11.00 and prior releases, the Buffer Cache Address map (there are actually 2 maps - **bufmap/bufmap2**. Further references to bufmap refer to both maps) is a resource map used to keep track of virtual addresses used by the buffer cache. The bufmap contains entries for each address range that is free and available for use by the buffer cache. The bufmap varies in size based on the memory size. It takes approximately 1% of memory on a 32-bit system and 2% of memory on a 64-bit OS.

The Buffer Cache Virtual Map (**bvmap**) is a bit map introduced at HP-UX 11.11 that represents pages in the buffer cache. Since it is a bitmap, its memory requirements are smaller than the bufmap. By default, the bitmap is sized (in bits) to number of physical memory pages \* the kernel tunable **bvmap\_size\_factor** (default 2). There is some overhead to manage the bitmap groups, but over all the memory usage is insignificant.

The default **bvmap\_size\_factor** of 2 is fine for many systems, especially those that use a **dbc\_max\_pct** of 20 or less. However, when a dynamic buffer cache is used, buffers of varying sizes can be allocated and de-allocated over time. The allocation and de-allocation

of variable sized buffers can fragment the bcvmmap. If there are not any bitmap areas available to represent the size of the buffers needed, the system may thrash. This is more common with systems where dbc\_max\_pct is configured at 50% or more or where different sized buffers are used. Buffers of different sizes may be used when multiple HFS file systems of different block sizes are used, or multiple JFS file systems are used that where some file systems have **max\_buf\_data\_size** set to 8 KB and others files systems have max\_buf\_data\_size set to 64 KB. In such caches, bcvmmap\_size\_factor should be increased to at least 16.

See vxtunefs(1M) for more information on max\_buf\_data\_size.

Note that the bcvmmap\_size\_factor parameter is only available on 64-bit 11.11 systems that have PHKL\_27808 installed.

The following tables provide some examples of memory requirements for systems of different memory sizes with different tunable values.

| <b>System Memory Size</b> | <b>10%<br/>BufPages/Buf Headers</b> | <b>20%<br/>BufPages/Buf Headers</b> | <b>50%<br/>BufPages/Buf Headers</b> |
|---------------------------|-------------------------------------|-------------------------------------|-------------------------------------|
| <b>1GB</b>                | <b>100MB/7.3MB</b>                  | <b>200MB/15MB</b>                   | <b>500MB/36MB</b>                   |
| <b>2GB</b>                | <b>200MB/15MB</b>                   | <b>400MB/30MB</b>                   | <b>1GB/75MB</b>                     |
| <b>4GB</b>                | <b>400MB/30MB</b>                   | <b>800MB/60MB</b>                   | <b>2GB/150MB</b>                    |
| <b>8GB</b>                | <b>800MB/60MB</b>                   | <b>1.6GB/120MB</b>                  | <b>4GB/300MB</b>                    |
| <b>12GB</b>               | <b>1.2GB/90MB</b>                   | <b>2.4GB/180MB</b>                  | <b>6GB/450MB</b>                    |
| <b>32GB</b>               | <b>3.2GB/240MB</b>                  | <b>6.4GB/480MB</b>                  | <b>16GB/1.2GB</b>                   |
| <b>256GB</b>              | <b>25.6GB/2GB</b>                   | <b>51GB/4GB</b>                     | <b>128GB/9.6GB</b>                  |

**Table 1 : Memory Usage of Buffer Cache Headers and Pages**

| System Memory Size | Hash Table Size | Hash Entries Per Lock | Hash Table Mem Requirements |
|--------------------|-----------------|-----------------------|-----------------------------|
| 1GB                | 65536           | 512                   | 2.5MB                       |
| 2GB                | 131072          | 1024                  | 5MB                         |
| 4GB                | 262144          | 2048                  | 10MB                        |
| 8GB                | 524288          | 4096                  | 20MB                        |
| 12GB               | 1048576         | 8192                  | 40MB                        |
| 32GB               | 2097152         | 16384                 | 80MB                        |
| 256GB              | 16777216        | 131072                | 640MB                       |

**Table 2 : Memory Usage of Buffer Cache Hash Table and Hash Locks**

## Advantages of the Buffer Cache

There are several advantages to using buffer cache.

1. **Small sequential I/O.** Applications read data from the file system in various size requests, which may not line up to the actual file system block size. Without the buffer cache, each request would have to go to the physical disk for the entire disk block, even though only a few bytes might be needed. If the next read is from the same physical disk block, the block would have to be read in again since it was not saved. However, with the buffer cache, the first read causes a physical I/O to the disk, but subsequent reads in the same block are satisfied through the buffer cache.
2. **Readahead.** If file system access is sequential, the buffer cache provides enhanced performance via readahead. When the file system detects sequential access to a file it begins doing asynchronous reads on subsequent blocks (also called prefetching) so that the data can potentially be available (or in transit) in the buffer cache when the application requests it.

For HFS file systems, general sequential reads are configured via the system tunable **hfs\_ra\_per\_disk**. If using LVM striping, we multiple the hfs\_ra\_per\_disk value by the number of stripes.

For JFS 3.1 file systems, sequential readahead starts out small and as the sequential access continues, JFS reads ahead more aggressively.

For JFS 3.3/3.5 the readahead range is the product of the file system tunables **read\_pref\_io** and **read\_nstream**. When sequential access is first detected, 4 ranges

are read into the buffer cache ( $4 * \text{read\_pref\_io} * \text{read\_nstream}$ ). When an application finishes reading in a range, a subsequent range is prefetched asynchronously. The readahead can greatly benefit sequential file access. However, applications that generally do random I/O may inadvertently trigger the large readahead by occasionally reading sequential blocks. This readahead will likely be unused due to the overall random nature of the reads.

For JFS 3.3/3.5 the size of the readahead can be controlled by the vxtunefs parameters `read_nstream` and `read_pref_io` and for JFS3.5 readahead can be turned off by setting the vxtunefs parameter `read_ahead` to 0. For JFS 3.1 the size of the readahead is not tunable.

3. **Hot Blocks.** If a file system block is repeatedly accessed by the application (either a single process or multiple processes), the block will stay in the buffer cache and can be used without having to go to the disk each time the data is needed. The buffer cache is particularly helpful when the application repeatedly searches a large directory, perhaps to create a temporary file. The directory blocks will likely be in buffer cache if they are frequently used and physical disk access will not be required.
4. **Delayed writes.** The buffer cache allows for applications to perform delayed or asynchronous writes. An application can simply write the data to the buffer cache and the system call will return without waiting for the I/O to disk to complete. The dirty buffer will be flushed to disk later via operations such as `syncer(1M)`, `sync(1M)`, or `fsync(2)`. Performing delayed writes is sometimes referred to as flush behind.

## Disadvantages of using Buffer Cache

While it may seem that every application would benefit from using buffer cache, using the buffer cache does have some costs.

1. **Memory.** Depending on how it is configured, the buffer cache may be the largest single user of memory. By default, a system with 8 GB of memory may use as much as 4 GB for buffer cache pages alone (with `dbc_max_pct` set to 50). Even with a dynamic buffer cache, a large buffer cache can contribute to overall memory pressure. Remember, the buffer cache will not return the buffer pages unless there is memory pressure. Once memory pressure is present, buffer pages are aged and stolen just as other user pages, so it contracts much slower than it expands.
2. **Flushing the Buffer Cache.**

**Syncer** - The `syncer(1M)` is the process that flushes delayed write buffers to the physical disk. Naturally, the larger the buffer cache, the more potential work performed by the `syncer`. The 11.0 `syncer` is single threaded. It wakes up periodically and sequentially scans the entire buffer cache for blocks that need to be written to the physical device. The default `syncer` interval is 30 seconds, which means that every 30 seconds the entire buffer cache is scanned for delayed write

blocks. The syncer actually runs 5 times during the syncer interval, scanning one fifth of the buffer cache each time.

HP-UX 11.11 is more efficient at flushing the dirty buffer. The 11.11 syncer is multi-threaded, with one thread per CPU. Each CPU has a dirty list and each syncer thread is responsible for flushing buffers from its respective dirty list. This improves buffer cache scaling, in that only dirty buffers are scanned and each thread has its own list of dirty buffers preventing contention around a single list.

**Other Sync Operations** - Various system operations require that dirty blocks in the buffer cache be written to disk before the operation completes. The last close of a file, an unmount of a file system, or a sync system call are examples of such operations. These operations are independent of the syncer and must traverse the entire buffer cache looking for blocks that need to be written to the device. For unmounts, once the operation completes, it must traverse the buffer cache again, invalidating the buffers that were flushed. These traversals of the buffer cache can take time, particularly if contention around the hash locks exists.

3. **IO Throttling** - Besides just walking the hash chains and locking/unlocking the hash locks, the larger the buffer cache, the larger the number of dirty buffers that are likely to be in the cache needing to be flushed to the disk. This can cause large amounts of write I/O to be queued to the disk during sync operations. A read request or synchronous write request could get delayed behind the writes and cause an application delay. Flushes of the buffer cache can be throttled by limiting the number of buffers that can be queued to a disk at one time. By default, throttling is turned off. For JFS 3.1 throttling can be enabled if PHKL\_27070 is installed by setting vx\_nothrottle to 0. This alleviates read starvation at the cost of sync operations such as unmounting a file system. For JFS 3.3/3.5 the amount of data flushed from a single file to a disk can be controlled via the vxtunefs **max\_diskq** parameter.

**Write Throttling** – Setting **max\_diskq** to throttle the flushing of dirty buffers has a disadvantage. Processes that perform sync operations, such as umount(1M) or bdf(1M) can stall since the writes are throttled. Setting max\_diskq does not prevent applications from continuing to perform asynchronous writes. If writes to large files are being done, it's possible to exhaust all the buffers with dirty buffers, and can delay reads or writes from other critical applications.

With JFS 3.5, a new tunable was introduced - **write\_throttle**. This controls the number of dirty buffers a single file can have outstanding. If an application attempts to write faster than the data can be written to disk and the write\_throttle amount has been reached, the application will wait until the some of the data is written to disk and the amount of data in the dirty buffers falls back below the write\_throttle amount.

Setting write\_throttle can alleviate the long sync times as well as keep the disk I/O queues manageable. However, the application writing to disk can experience delays as it writes the data.

4. **Large I/O.** The maximum size of a buffer is 64 Kb. For I/Os larger than 64 Kb, the request must be broken down into multiple 64 Kb I/Os. So reading 256 Kb from disk

may require 4 I/Os. However, if the buffer cache is bypassed, a single 256 Kb direct I/O could potentially be performed.

5. **Data accessed once.** Management of the buffer cache does require additional code and processing. For data that is accessed only once, the buffer cache does not provide any benefit for keeping the data in the cache. In fact, by caching data that is accessed only once, the system may need to remove buffer pages that are more frequently accessed.
6. **System crash.** Since many writes are delayed, the system may have many “dirty” buffers in the buffer cache that need to be posted to disk when the system crashes. Data in the buffer cache that is not flushed before the system comes down is lost.

## **Bypassing the buffer cache**

There are several ways that I/O can avoid using the buffer cache altogether. Bypassing the buffer cache is known as direct I/O. For the JFS features mentioned below, the HP OnlineJFS license is needed to perform direct I/O.

### **- mincache=direct, convosync=direct**

For JFS file systems, if the file system is mounted with the mincache=direct option and/or the convosync=direct option, reads and writes to the buffer cache are bypassed – see the manpage for mount\_vxfs(1M) for more details. If a file system is mounted with these options all I/O to the file system bypasses the buffer cache. Readahead and write behind are not available when bypassing the buffer cache since there is no intermediate holding area.

### **- ioctl (fd,VX\_SETCACHE, VX\_DIRECT)**

For JFS file systems, this ioctl call will set the access for the file referenced by fd as direct and will bypass the buffer cache. This call applies only to the instance of the file represented by fd. Other applications opening the same file are not affected. See the vxfsio(7) manpage for more information.

### **-discovered direct IO**

JFS provides a feature called discovered direct I/O, where I/Os larger than a certain size are done using direct I/O. Large I/Os are typically performed by applications that read data once, such as backup or copy operations. Since data is accessed once, there is no benefit to caching the data. Caching the data may even be detrimental as more useful buffers may get reused to cache the once accessed data. Therefore large I/Os on JFS file systems are performed “direct” and bypass the buffer cache. For JFS 3.1, the discovered direct I/O size is fixed at 128 Kb. For JFS 3.3/3.5 the default discovered direct IO size is 256 Kb, but can set with vxtunefs(1M) by setting the **discovered\_direct\_iosz** tunable.

## **-raw io**

If access to the data is through the raw device file, then buffer cache is not used.

## **-async driver**

Some databases use async driver (/dev/async), which performs asynchronous I/O, but bypasses the buffer cache and reads directly into shared memory segments.

## **Guidelines to Tuning the HP-UX Buffer Cache.**

Providing guidelines for tuning the buffer cache is very difficult. So much depends on the application mix that is running on the system. However, some generalizations can be made.

If using a database, the database buffering will likely be more efficient than the system buffering. The database is more likely to understand the I/O patterns and therefore keep relevant buffers in memory. So given a choice, memory should be assigned to the database global area rather than the system buffer cache.

As we have seen HP-UX 11.11 is more efficient at handling large buffer caches than HP-UX 11.0 . The term “large” is relative, but for this discussion consider a buffer cache larger than 1 Gb or greater than 50% of memory to be large. In general the buffer cache on 11.0 should be configured to 1GB or less due to scaling issues with large caches, but can be larger on 11.11. However, if you are using a large buffer cache on 11.11 you should have PHKL\_27808 or superceding patch installed to increase the buffer cache virtual map size.

The larger the buffer cache, the longer sync operations will take. This particularly affects file system mount and unmount times. If file systems need to be mounted or unmounted quickly (for example during SG package switch) , then a smaller buffer cache may be desired.

If the buffer cache is configured too small, the system could be constantly searching for available buffers. The buffer cache should probably be configured to a minimum of 200MB on most systems.

Applications that benefit most from large caches are often file servers, such as NFS or web servers, where large amounts of data are frequently accessed. Some database applications that do not manage their own file access may also fall into this category. Please check with your application vendor for any vendor specific recommendations.

# **The JFS Inode Cache**

**Mark Ray**

Global Solutions Engineering  
Hewlett Packard Company

## The JFS Inode Cache

HP-UX has long maintained a static cache in physical memory for storing HFS file information (inodes). The Veritas File System<sup>1</sup> (HP OnlineJFS/JFS) manages its own cache of file system inodes which will be referred to here as the JFS inode cache.<sup>2</sup> The JFS inode cache is managed much differently than the HFS inode cache. Understanding how the JFS inode cache is managed is key to understanding how to best tune the JFS inode cache for your unique environment. Improper tuning of the JFS inode cache can affect memory usage or inode lookup performance.

This paper will address the following questions regarding the JFS inode cache.

1. What is an inode cache?
2. What is the maximum size of the inode cache?
3. How can I determine the number of inodes in the cache?
4. How can I determine the number of inodes in use?
5. How does JFS manage the inode cache?
6. How much memory is required for the JFS inode cache?
7. Are there any guidelines for configuring the JFS inode cache?

### What is an Inode Cache?

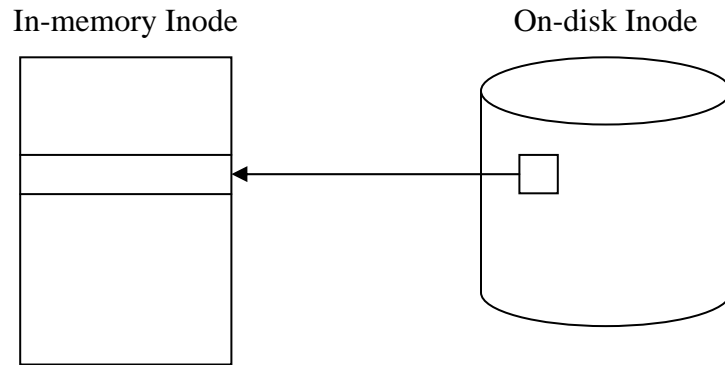
An inode cache is simply a holding location for inodes from disk. Each inode in memory is a superset of data, which contains the inode from disk. The disk inode stores information for each file such as the file type, permissions, timestamps, size of file, number of blocks, and extent map. The in-memory inode stores the on-disk inode information along with overhead used to manage the inode in memory. This information includes pointers to other structures, pointers used to maintain linked lists, the inode number, lock primitives, etc

Once the inode is brought into memory, subsequent access to the inode can be done through memory without having to read or write it to disk.

---

1. VERITAS is a registered trademark of VERITAS Software Corporation

2. JFS will be used to refer to the HP products HP OnlineJFS/JFS, while VxFS will be used to refer to the filesystem type used by the Veritas Filesystem.



One inode cache entry must exist for every file that is opened on the system. If the inode table fills up with active inodes, the following error will be seen on the console and in the syslog file and the `open()` system call will fail:

```
vx_iget - inode table overflow
```

Once the last close is done on a file, the inode will be put on a free list. It is not necessarily removed from the cache. So the inode cache will likely contain some files that are closed, so if the file is re-opened, a disk access will not occur as the inode is already in memory.

## The JFS Inode Cache is a Dynamic Cache

Unlike HFS, the JFS inode cache is a dynamic cache. A dynamic cache is a cache that grows and shrinks based on need. As files are opened, the number of inodes in the JFS inode cache grows. As files are closed, they are moved to a free list and can be re-used at a later time. However, if the inode is inactive for a certain period of time, the inode is freed and space is return to the kernel memory allocator. So over time, the numbers of inodes in the inode cache will grow and shrink.

## Maximum Inodes in the JFS Inode Cache

While the JFS inode cache is dynamically sized, there is still an absolute maximum number of inodes that can be maintained in the inode cache. The table below shows the default maximum number of inodes in the JFS inode cache<sup>3</sup>:

---

3. Currently, JFS 3.1 is supported on 11.00, JFS 3.3 is supported on 11.00, 11.11, 11.20, and 11.22, and JFS 3.5 is supported on 11.11 and 11.23.

| Physical Memory | JFS 3.1 | JFS 3.3/<br>JFS 3.5 |
|-----------------|---------|---------------------|
| 128 Mb          | 9333    | 8000                |
| 256 Mb          | 18666   | 16000               |
| 512 Mb          | 37333   | 32000               |
| 1 Gb            | 74666   | 64000               |
| 2 Gb            | 149333  | 128000              |
| 8 Gb            | 149333  | 256000              |
| 32 Gb           | 149333  | 512000              |
| 128 Gb          | 149333  | 1024000             |

**Table 1: Default Values for Maximum Inodes in JFS Inode Cache**

Specific values exist for memory sizes less than 128 Mb, but the sizes are not mentioned here since most HP-UX systems should be using 256 Mb of memory or more.

If the size of memory falls in between the memory sizes listed above, a value proportional to the 2 surrounding values is used. For example, a system using JFS 3.3 with 5 gigabytes of memory would have a default maximum number of inodes in the inode cache of 192000.

For JFS 3.1, any system with 2 gigabytes or more of memory will have a maximum of 149333 inodes in the JFS inode cache. However, for JFS 3.3 and JFS 3.5, the maximum continues to scale based on the last 2 entries.

Note the default maximum number of inodes seems very high. Remember that it is a maximum, and the JFS inode cache is dynamic, as it can shrink and grow as inodes are opened and closed. The maximum size must be large enough to handle the maximum number of concurrently opened files at any given time, or the “vx\_iget - inode table overflow” error will occur.

You can verify the maximum size of the JFS inode cache by using the following adb commands<sup>4</sup> prior to JFS 3.5:

|                  |                   |
|------------------|-------------------|
| JFS 3.1 (32-bit) | vxfs_fshead+0x8/D |
| JFS 3.1 (64-bit) | vxfs_fshead+0xc/D |
| JFS 3.3          | vxfs_ninode/D     |

---

4. To start adb, type “adb -k /stand/vmunix /dev/mem”

Using JFS 3.5, the `vxfssstat(1M)` command can be used to display the maximum number of inodes in the inode cache:

```
# vxfssstat / | grep inodes
    3087 inodes current      128002 peak                128000 maximum
    255019 inodes allocated  251932 freed

# vxfssstat -v / | grep maxino
vxi_icache_maxino          128000      vxi_icache_peakino          128002
```

Note in the above example, the inode cache can handle a maximum of 128,000 inodes.

## Determining the Current Number of Inodes in the JFS Inode Cache

Determining how many inodes currently in the inode cache is difficult on JFS versions prior to JFS 3.5 as existing user tools do not give this information. The following `adb` commands can be used to display the current number of inodes in the JFS inode cache:

|                  |                                 |
|------------------|---------------------------------|
| JFS 3.1 (32-bit) | <code>vxfss_fshead+010/D</code> |
| JFS 3.1 (64-bit) | <code>vxfss_fshead+014/D</code> |
| JFS 3.3          | <code>vx_cur_inodes/D</code>    |

Using JFS 3.5, the `vxfssstat(1M)` command can be used to display the current number of inodes in the inode cache:

```
# vxfssstat / | grep inodes
    3087 inodes current      128002 peak                128000 maximum
    255019 inodes allocated  251932 freed

# vxfssstat -v / | grep curino
vxi_icache_curino          3087      vxi_icache_inuseino          635
```

Note from the above output, the current number of inodes in the cache is 3087.

## Determining the Number of Active JFS Inodes in Use

While a number of the JFS inodes exist in the cache, remember that not all of the inodes are actually in use as inactive inodes exist in the cache. For 11.0, JFS 3.1, you can find the number of active inodes actually in use as follows:

|                  |                                 |
|------------------|---------------------------------|
| JFS 3.1 (32-bit) | <code>vxfss_fshead+014/D</code> |
| JFS 3.1 (64-bit) | <code>vxfss_fshead+018/D</code> |

For JFS 3.3, there is no easy method to determine the actual inodes in use. The structures were changed in JFS 3.3 and the `vxfsstat(1M)` command is not available until JFS 3.5. For JFS 3.5, we can again use `vxfsstat(1M)` to determine the actual number of JFS inodes that are in use:

```
# vxfsstat -v / | grep inuse
vxi_icache_curino          128001      vxi_icache_inuseino      635
```

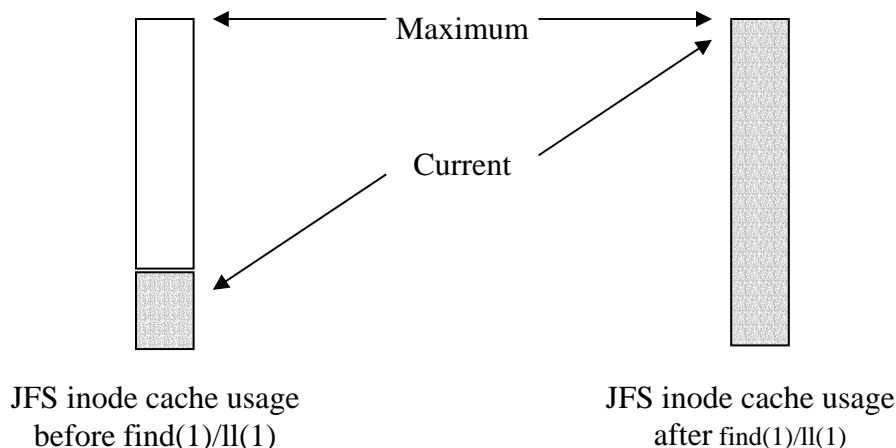
So while the inode cache is full with 128001 inodes, only 635 are actually in use. The remaining inodes are actually inactive, and if they remain inactive, one of the `vxfsd` daemon threads will start freeing the inodes after a certain period of time.

Note that the current number of inodes (128001) is greater than the maximum number of inodes (128000). This behavior is normal, as there are few exceptions that allow allocating a few additional inodes in the cache.

## Growing the JFS Inode Cache

When a file is opened and does not already exist in the cache, its inode must be brought in from disk into the JFS inode cache. So JFS must make a decision as to whether to use one of the existing inodes on a free list, or to allocate a new inode if we have not allocated the maximum number of inodes. For JFS 3.1, an inode must be on one of the free lists for 2 minutes before it is reused. For JFS 3.3 and JFS 3.5, an inode must be on a free list for 3 minutes before it is reused. If the inodes on the free list have been there for less than 2 or 3 minutes, then JFS will allocate more inodes from the kernel memory allocator as necessary.

Now, consider an application that does a `stat(2)` system call on many files in a very short timeframe. Example applications or processes are the `find(1)`, `ls(1)`, `ll(1)` or `backup` commands. These commands traverse through the file systems at a very rapid pace. If you have enough files on your system, you can easily fill up the JFS inode cache in a very short time. However, the files are usually not kept open. So if the JFS inode cache fills up, an inode is claimed off the free list even though it has not been on the free list for the appropriate amount of time.



Note that maintaining all these inodes may or may not be helpful. For example, if the inode cache can hold 128,000 inodes, and the find(1) command traverses 129,000 inodes, then only the last 128,000 inodes would be in the cache. The first 1,000 inodes would have been reused when reading in the last 1000 inodes. So if the find(1) command were performed again, then it would have to re-cache all 129,000 inodes.

However, if the find(1) command only traversed through 127,000 inodes, then all of the inodes would be in the cache for the 2nd find(1) command, which would then run much faster.

As an example, consider an HP-UX 11.11 system using JFS 3.5. Prior to a find(1) command, the vxfsstat(1M) command shows the current number of inodes to be 3087.

```
# vxfsstat / | grep inodes
  3087 inodes current      128002 peak              128000 maximum
 255019 inodes allocated  251932 freed
```

When the find(1) command is performed, the inode cache will fill up if it traverses enough files:

```
# find / -name testfile
# vxfsstat / | grep inodes
 128001 inodes current    128002 peak              128000 maximum
 379936 inodes allocated  251935 freed
```

After the find(1), the number of inodes currently in the cache jumped to 128,001.

The “peak” value of 128002 represents the highest value that the “inodes current” has been since the system was booted. Again, its normal for the “current” and “peak” counts to be greater than the “maximum” value by a few inodes.

## Shrinking the JFS Inode Cache

Periodically, one of the JFS daemon (vxfsd) threads runs to scan the free lists to see if any inodes have been inactive for a period of time. If so, the daemon thread begins to free the inodes back to the kernel memory allocator so the memory can be used in future kernel allocations. The length of time an inode can stay on the free list before it is freed back to the kernel allocator and the rate at which the inodes are freed varies depending on the JFS version.

|  | JFS 3.1                               | JFS 3.3 | JFS 3.5 |
|--|---------------------------------------|---------|---------|
| Minimum time on free list before being freed (seconds) | 300                                   | 500     | 1800    |
| Maximum inodes to free per second                      | 1/300 <sup>th</sup> of current inodes | 50      | 1 - 25  |

**Table 2: JFS Inode free rate**

As an example, JFS 3.3 will take approximately 2000 seconds or 33 minutes to free up 100,000 inodes.

With JFS 3.5, you can actually see the minimum time on the free list before the inode is a candidate to free using `vxfsstat(1M)`:

```
# vxfsstat -i / | grep "sec free"
    1800 sec free age
```

This value is also considered the `ifree_timelag`, and can also be displayed with `vxfsstat(1M)` on JFS 3.5:

```
# vxfsstat -v / | grep ifree
vxi_icache_recycleage          1035    vxi_ifree_timelag          1800
```

Consider the JFS 3.5 case again, the system begins to free the inodes if they have been inactive for 30 minutes (1800 seconds) or more. So about 30 minutes after the `find(1)` command, the inodes start to free up:

```
# date; vxfsstat -v / | grep -i curino
Thu May  8 16:34:43 MDT 2003
vxi_icache_curino          127526    vxi_icache_inuseino          635
```

The `vxfsstat(1M)` is executed again 134 seconds later:

```
# date; vxfsstat -v / | grep -i curino
Thu May  8 16:36:57 MDT 2003
vxi_icache_curino          127101    vxi_icache_inuseino          635
```

Note that 425 inodes were freed in 134 seconds, about 3 inodes per second.

After being idle all evening, the next day, the number of inodes in the inode cache were reduced down again as all the inactive files were freed:

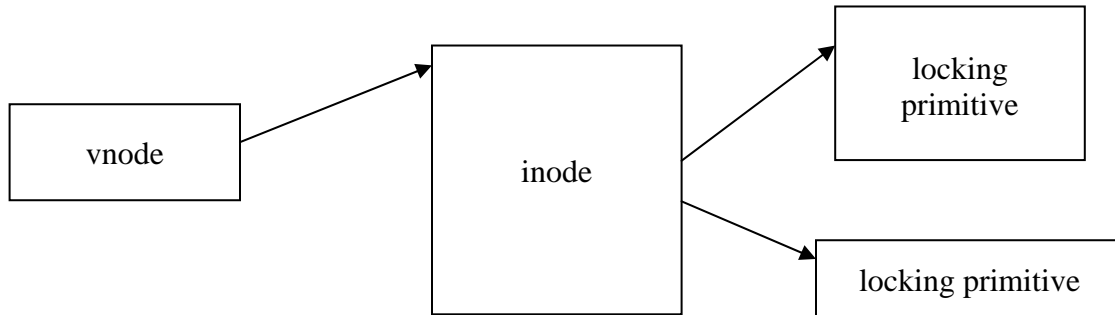
```
# date; vxfsstat -v / | grep -i curino
Fri May  9 14:45:31 MDT 2003
vxi_icache_curino          3011     vxi_icache_inuseino          636
```

## Determining the Memory Cost Associated with each JFS Inode

The size of the individual inodes varies depending on the release. Prior to JFS 3.3, the vnode (virtual file system node) and inode were allocated together as a single resource. On JFS 3.3 and later, the vnode is allocated separately from the inode. On a 32-bit OS, the inode is smaller as the pointer fields are only 4 bytes. On a 64-bit OS, the pointer fields are 8 bytes.

The kernel memory allocator also impacts the amount of space used when allocating the inodes and associated data structures. We only allocate memory using certain sizes. For example, if the kernel tries to allocate an 80 byte structure, the allocator may round the size up to 128 bytes, so that all allocations in the same memory page are of the same size.

For each inode, we also need to account for locking structures that are also allocated separately from the inode.



The table below can be used to estimate the memory cost of each JFS inode in the inode cache. Each items reflects the size as allocated by the kernel memory allocator.

| Structures      | JFS 3.1<br>11.00<br>32-bit | JFS 3.1<br>11.00<br>64-bit | JFS 3.3<br>11.00<br>32-bit | JFS 3.3<br>11.00<br>64-bit | JFS 3.3<br>11.11<br>32-bit | JFS 3.3<br>11.11<br>64-bit | JFS 3.5<br>11.11<br>64-bit |
|-----------------|----------------------------|----------------------------|----------------------------|----------------------------|----------------------------|----------------------------|----------------------------|
| inode           | 1024                       | 2048                       | 1024                       | 1024                       | 1024                       | 1364                       | 1364                       |
| vnode           | *                          | *                          | 128                        | 256                        | 128                        | 184                        | 184                        |
| lock structures | 196                        | 196                        | 352                        | 352                        | 272                        | 384                        | 352                        |
| Total           | 1220                       | 2244                       | 1494                       | 1632                       | 1352                       | 1902                       | 1850                       |

**Table 3: Estimated Memory Cost per Inode (in bytes)**

Note that the above is a minimal set of memory requirements. There are also other supporting structures, such as hash headers and free list headers. Other features may use more memory. For example, using Fancy Readahead on JFS 3.3 on a file will consume approximately 1024 additional bytes per inode. Access Control Lists (ACLs) and quotas can also take up additional space.

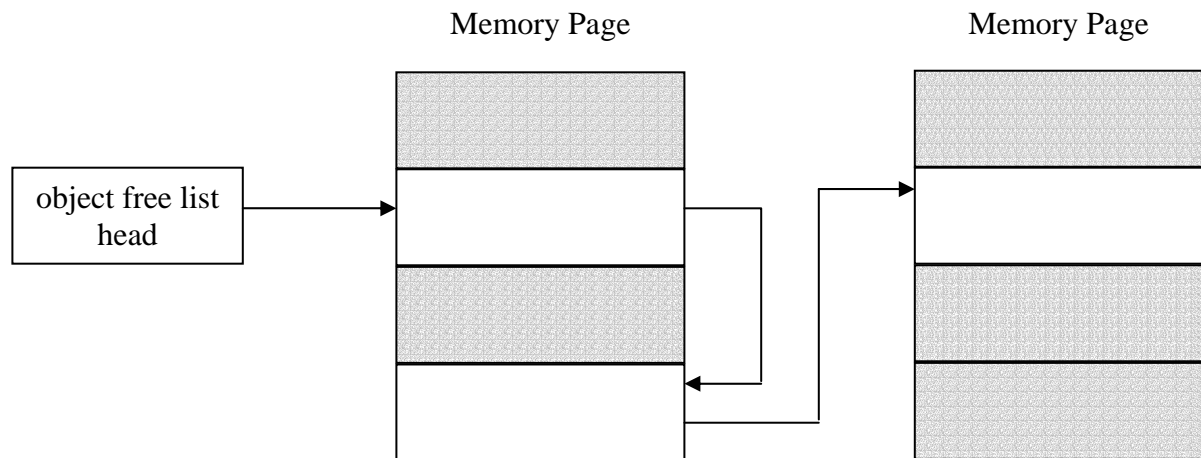
As an example, consider an HP-UX 11.0 (64-bit) system using JFS 3.1 that has 2GB of memory. If an ll(1) or find(1) command is done on a file system with a large number of files (>149333), then the inode cache is likely to fill up. Based on the default JFS Inode Cache size of 149333, the minimum memory cost would be approximately 319 MB or 15% of total memory.

Now, consider what happens if the system is upgraded to use JFS 3.3 on the same 11.0 system. The default size of the inode cache is now 128,000 and the memory cost per inode is 1632. Thus the minimum memory cost if the inode cache fills up is approximately 200 MB or 10% of total memory. By allocating the vnode separately in JFS 3.3 and using a smaller default JFS inode cache size, the memory usage actually declined.

However, if you then add more memory so the system has 8 Gb of memory instead of 2 Gb, then the memory cost for the JFS inode will increase to approximately 400 MB. While the total memory cost increases, the percentage of overall memory used for the JFS inode cache drops to 5%.

## Effects of the Kernel Memory Allocator

The internals of the kernel memory allocator have changed over time, but the concept remains the same. When the kernel requests dynamic memory, it allocates an entire page (4096 bytes) and then subdivides the memory into equal sized chunks known as “objects”. The kernel does allocate pages using different sized objects. The allocator then maintains free lists based on the object size.



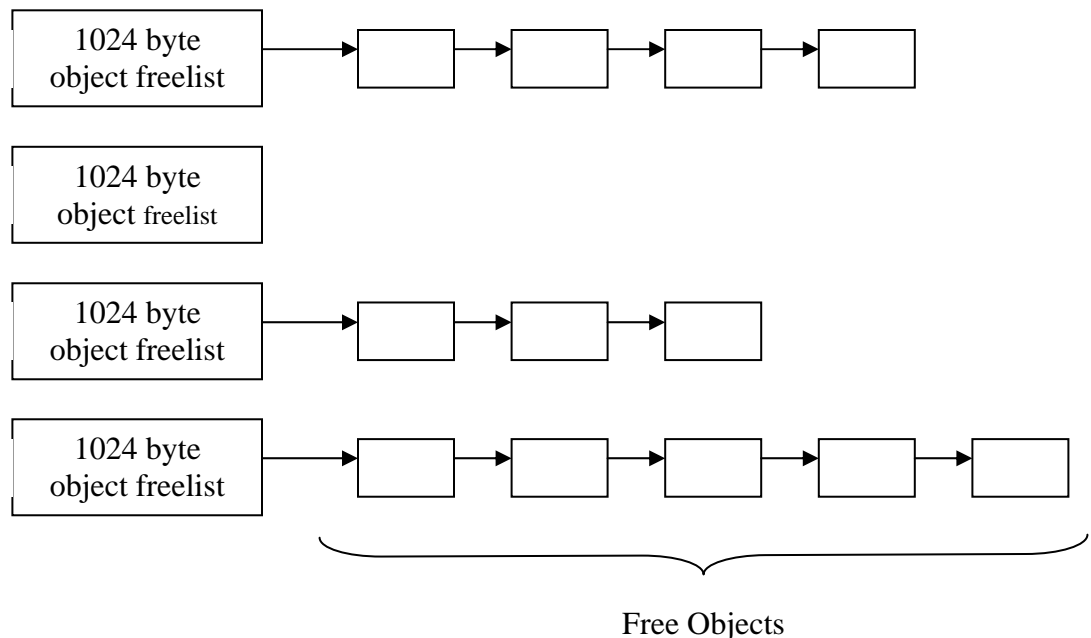
In the example above, we have memory pages divided into 4 objects. In some implementations, there is some page overhead and object overhead associated with the object. For this example, assume the size of each object is 1024 bytes. Each page may have both used and freed objects associated with it. All of the free pages are linked to a linked list pointed to by a object free list head. There is typically one object free list head associated with each CPU on the system for each object size. So CPU 0 can have a object free list for the 32-byte objects, the 64-byte objects, etc. CPU 1 would also have corresponding object free lists.

There are multiple object sizes available. However, not all sizes are represented. For example, on 11.0 there are pages that use the 1024-byte object and the 2048-byte object, but nothing in

between. So if JFS requests 1040 bytes of memory, an entire 2048-byte object is taken from a page divided into two 2048 byte objects.

As inodes are allocated, the kernel will allocate memory pages and divide it up into objects, which are then used by the JFS subsystem for inodes and associated data structures. In order to allocate 128000 inodes which take 1024 bytes each, the system would need to allocate 32000 pages (4 inodes per page).

As discussed earlier, the JFS inode cache is dynamic. Inodes that are not accessed are eventually freed. These freed inodes go back to the object freelist. If an ll(1) or find(1) is done to access 128,000 inodes, and then the inodes are unreferenced for some time, then large object freelist chains can form as the JFS daemon thread starts to free the inodes. This can occur for each CPU object freelist since inodes are freed to the CPU's object chain where they were allocated.



Note that in the above diagram, if CPU 3 needed a new inode, the kernel memory allocator would simply return the 1st object from the object free list chain. However, the object freelist chain for CPU 1 is empty. If an ll(1) or find(1) is done, new inodes will be needed in the JFS inode cache. Thus new memory pages will be allocated and divided up into objects and placed in the object freelist for CPU 1. The kernel memory allocator will not steal free objects from another CPU's free object chain. By not stealing objects, the system realizes a performance gain by reducing contention on the object chain locks. However, when an inode is freed, the kernel memory allocator will place the freed area on the free object chain for CPU 1 to be used for a subsequent allocation. The freeing of the inodes creates potentially large free object chains for each CPU as the JFS inode cache shrinks.

While the object freelists can be used for other 1024-byte allocations, they cannot be used for other sized allocations as all the objects allocated from a page must be the same size. If you

have 4 CPUs, then you can potentially consume memory for freed inodes on the object freelist as well as used inodes in the JFS inode cache.

The HP-UX kernel does perform “object coalescing”, such that if all the objects in a given page are freed, then the memory page can be returned to the free pool and allocated for use in other areas (process memory, different size memory objects, etc.). However, the “object coalescing” only occurs if there is memory pressure present and there is some overhead in performing the coalescing.

## **Tuning the Maximum Size of the JFS Inode Cache**

Every customer environment is unique. The advantage to a large inode cache is that we do not have to access the disk every time the inode is needed. If you are doing continuous random lookups on a directory with 64,000 files in it (such as opening a file or performing a stat(2) system call), then having a large cache is helpful. However, if you are doing a find(1) or ll(1) command from time to time on a set of 512,000 inodes, having 128,000 inodes in the cache will not help. You need to understand how your files are accessed to know whether or not a large inode cache will be helpful. The memory cost associated with each inode should be taken into account as well.

You can tune the maximum size of the JFS inode cache using the **vx\_ninode** tunable. This tunable was introduced on 10.20 with kernel patches PHKL\_23835 (s700) and PHKL\_23836 (s800). The tunable is available on 11.0 and 11.11 (with PHKL\_24783), and on subsequent releases.

At a minimum, you must have at least one JFS inode cache entry for each file that is opened at any given time on your system. If you are concerned about the amount of memory that JFS can potentially take, then try to tune vx\_ninode down so that the cache only takes ~1-2% of overall memory. Most systems will work fine with vx\_ninode tuned to 8,000 - 20,000. However, you need to consider how many processes are running on the system and how many files each process will have open on average. Systems used as file servers and web servers may have performance benefits from using a large JFS inode cache and the defaults are sufficient.

Note that tuning ninode does not affect the JFS inode cache as the JFS inode cache is maintained separately from the HFS inode cache. If your only HFS file system is /stand, then ninode can usually be tuned to a low value (for example, 400).

## **Tuning Your System to Use a Static JFS Inode Cache**

By default, the JFS inode cache is dynamic in size. It grows and shrinks as needed. However, since the inodes are freed to the kernel memory allocator’s free object chain, the memory may not be available for use for other reasons (except for other same-sized memory allocations). The freed inodes on the object free lists are still considered “used” system memory. Also, the massive kernel memory allocations and subsequent frees add additional overhead to the kernel.

The dynamic nature of the JFS inode cache does not provide much benefit. This may change in the future as the kernel memory allocator continues to evolve. However, using a statically sized JFS inode cache has the advantage of keeping inodes in the cache longer, and reducing overhead of continued kernel memory allocations and frees. Instead, the unused inodes are retained on the JFS inode cache free list chains. If we need to bring a new JFS inode in from disk we simply use the oldest inactive inode. Using a static JFS inode cache also avoids the long kernel memory object free chains for each CPU.

A static JFS inode cache can be tuned by setting the `vx_noifree` kernel tunable. The tunable was introduced on HP-UX 10.20 with kernel patches PHKL\_23835 (s700) and PHKL\_23836 (s800). It is also available using JFS 3.1 on 11.0. The `vx_noifree` tunable is not available on JFS 3.3 and above at the present time.

## Summary

Deciding whether or not to tune the JFS Inode Cache depends on how you plan to use your system. Memory is a finite resource, and a system manager needs to decide how much of the system's memory needs to be spent on a specific resource.

By default, the JFS inode cache is configured to be very large. You should understand the advantages and disadvantages of using the default JFS inode cache sizes and behavior. The type of access to the files on the system and the memory cost associated with the JFS inode cache should be considered when deciding whether or not to tune the JFS inode cache.

For most customers, tuning the JFS inode cache down would save memory and potentially enhance performance by leaving more memory for other operations. With the introduction of `vxfsstat(1M)` on JFS 3.5, you can determine how many inodes are actually in use at a given time to judge how large your inode cache should be. Most systems do not need more than 20,000 inodes, so reducing the size of the JFS inode cache can reduce system memory utilization.

Since the JFS inode cache often expands to the maximum amount, using a fixed sized inode cache will help the kernel memory allocator from managing large per-cpu freelists. Therefore, tuning `vx_noifree` can also reduce overall system memory utilization.

However, if the primary use of the system is a fileserver, where random file lookups are being performed constantly or sequential lookups are done and the working set is less than the size of the inode cache, then using the default JFS inode cache sizes is probably best. Application performance could be degraded if the application relies on having a larger working set of inodes in the cache.

# **The JFS 3.5 Metadata Buffer Cache**

**Mark Ray**  
Global Solutions Engineering  
Hewlett Packard Company

## The JFS 3.5 Metadata Buffer Cache

In past releases of the Veritas File System<sup>5</sup> (HP OnlineJFS/JFS), the metadata of a file system was cached in the standard HP-UX Buffer Cache with all of the user file data. Beginning with JFS 3.5 introduced on HP-UX 11.11, the JFS metadata was moved to a special buffer cache known as the JFS Metadata Buffer Cache (or metadata cache). This cache is managed separately from the HP-UX Buffer Cache. This metadata cache serves the same purpose as the HP-UX buffer cache, but enhancements were made to increase performance due to the unique ways the metadata is accessed.

This paper will address the following questions regarding the metadata cache:

1. What is metadata?
2. Is the metadata cache static or dynamic?
3. How much memory is required for the metadata cache?
4. How can the metadata cache be tuned?
5. Are there any guidelines for configuring the metadata cache?

### What is Metadata?

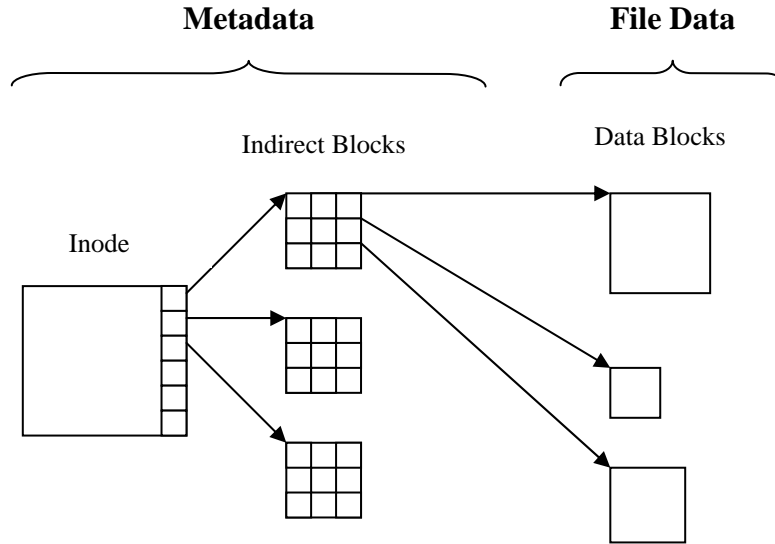
Metadata is structural information from disk such as inodes, indirect block maps, bitmaps, and summaries.

If you consider an actual file on disk, it is made up of the inode and data blocks, and potentially indirect blocks. The inode contains an extent map to either the data blocks or other extent maps known as indirect blocks

When inodes are first read in from disk, the file system reads in an entire block of inodes from disk into the metadata cache (similar to reading a data file). Then, the inodes that are actually being accessed will be brought into the JFS inode cache. Note the difference between inodes in the metadata cache which contains the disk copy only and the inode in the JFS inode cache which contains the linked lists for hashing and free lists, the vnode, locking structures, and the on-disk copy of the inode.

---

<sup>5</sup> VERITAS is a registered trademark of VERITAS Software Corporation.



## The Metadata Cache – dynamic or static?

The metadata cache is a dynamic buffer cache, which means it can expand and shrink over time. It normally expands during periods of heavy metadata activity, especially with operations that traverse a large number of inodes, such as a `find(1M)` or backup. Simply reading a large file may fill up the HP-UX buffer cache, but not the metadata cache.

Unlike the HP-UX buffer cache, which contracts only when memory pressure is present, the metadata cache contracts after the metadata buffers have been inactive for a period of time (1 hour). While the metadata cache contracts, it contracts at a slow rate, such that the size often remains at or near the maximum size.

Also, during bootup, the JFS allocates all the buffers up to the maximum buffers allowable. So the cache starts out full and contracts slowly. So while the cache is dynamic, it may appear to be static unless monitored over a long period of time.

The current size of the buffer cache and the maximum size of the buffer cache can be viewed using the `vxfstat(1M)` command:

```
# vxfstat -b /

12:55:26.640 Thu Jul 3 2003 -- absolute sample

buffer cache statistics
348416 Kbyte current          356040 maximum
122861 lookups                98.78% hit rate
2428 sec recycle age [not limited by maximum]
```

## The Metadata Cache and Memory

The default maximum size of the metadata cache varies depending on the amount of physical memory in the system according to the table below:

| Memory Size (Mb) | JFS Metadata Cache (Kb) | JFS Metadata Cache as a percent of memory |
|------------------|-------------------------|---|
| 256              | 32000                   | 12.2%                                     |
| 512              | 64000                   | 12.2%                                     |
| 1024             | 128000                  | 12.2%                                     |
| 2048             | 256000                  | 12.2%                                     |
| 8192             | 512000                  | 6.1%                                      |
| 32768            | 1024000                 | 3.0%                                      |
| 131072           | 2048000                 | 1.5%                                      |

**Table 1: Size of JFS Metadata Cache**

If the size of memory falls between 2 ranges, the maximum sizes is calculated in proportion to the 2 neighboring memory ranges. For example, if the system has 4 GB of memory (4096 Mb), then the calculated maximum metadata cache size is 356040 Kb, a value proportional to the 2 GB and 8 GB range.

Note that the table represents the *maximum* size of the metadata cache. And while the cache is dynamic, it does not contract very fast. However, it is possible that there is not enough metadata to fill the cache, even if all the metadata is brought into the cache. The `vxfstat(1M)` command should be used to see how much metadata is in the cache.

However, the memory usage from the Table 1 accounts for only the buffer pages. It does not include other overhead for managing the metadata cache, such as the buffer headers, hash headers and free list headers. The buffer headers are dynamically allocated and deleted as needed. The size of the buffers may vary, so the actual number of buffer headers needed will vary as well.

For estimation purposes, we will calculate the space needed for the buffer headers based on an average buffer size between 4096 and 8192. Each buffer header will take approximately 820 bytes. The actual amount of memory allocated to the buffer headers will likely be somewhere in between.

| Memory Size (Mb) | JFS Metadata Cache (Kb) | Memory for buffer headers (4kb per buffer) | Memory for buffer headers (8kb per buffer) |
|------------------|-------------------------|--|--|
| 256              | 32000                   | 6.25 Mb                                    | 3.13 Mb                                    |
| 512              | 64000                   | 12.5 Mb                                    | 6.25 Mb                                    |
| 1024             | 128000                  | 25 Mb                                      | 12.5 Mb                                    |
| 2048             | 256000                  | 50 Mb                                      | 25 Mb                                      |
| 8192             | 512000                  | 100 Mb                                     | 50 Mb                                      |
| 32768            | 1024000                 | 200 Mb                                     | 100 Mb                                     |
| 131072           | 2048000                 | 400 Mb                                     | 200 Mb                                     |

**Table 2: Memory used by buffer headers**

Note the buffer headers will take an estimate of 0.1% to 2.4% of memory, depending on the memory size, the buffer sizes, and the number of buffers actually in the cache.

## Tuning the Metadata Cache

The kernel tunable `vx_bc_bufhwm` specifies the maximum amount of memory in kilobytes (or high water mark) to allow for the buffer pages. This value can be set with `sam(1M)` or `kmtune(1M)`, or in the `/stand/system` file. By default, `vx_bc_bufhwm` is set to zero, which means to use the default maximum sized based on the physical memory size (see Table 1).

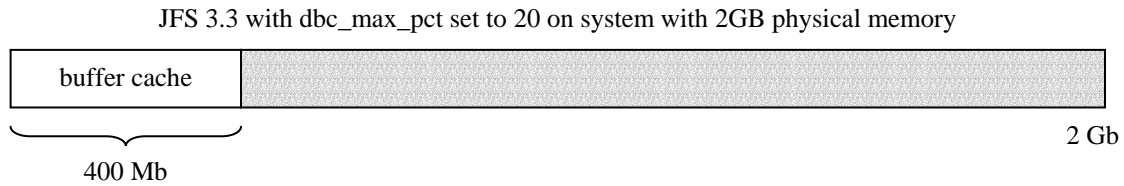
The system must be rebooted after changing `vx_bc_bufhwm` for the new value to take effect.

## Recommended Guidelines for tuning the JFS Metadata Buffer Cache

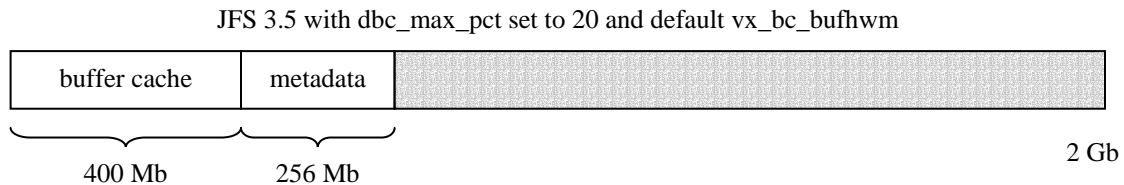
If you upgrade from JFS 3.3 to JFS 3.5 on 11.11, the metadata cache can take up to 15% of memory (depending on memory size) above what was taken by JFS 3.1 since the metadata on JFS 3.3 was included in the HP-UX buffer cache. This can potentially increase performance of metadata intensive applications (for example, applications that perform a high number of file creations/deletions or those that use large directories).

However, the memory cost must be considered. If the system is already running close to the low memory threshold, the increased memory usage can consume memory that could potentially be used for other applications, potentially degrading the performance of other applications.

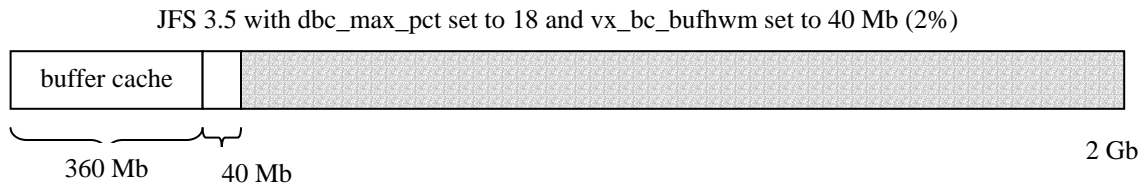
As an example, consider a 2GB system with `dbc_max_pct` set to 20%, running JFS 3.3 on 11.11.



The JFS file system is then upgraded to JFS 3.5. Now, `dbc_max_pct` is still 20%, but the metadata buffer cache is an additional 12% of memory (plus overhead for the hash headers, free lists headers, etc). Thus less space exists for other applications. If you still desire to use only 20% for buffer cache for both data and metadata, the tunables must be evaluated and changed.



As a suggestion, consider the maximum amount both data and metadata that is desired, and then consider the amount of metadata as a percentage of overall data. Going back to the example above, if the system should use a maximum of 20% of memory for both data and metadata, and you desire a 90/10 ratio of data to metadata, change `dbc_max_pct` to 18 and `vx_bc_bufhwm` to 40000 (or 40 Mb which is 2% of the 2 Gb physical memory).



Note that this example uses a 90/10 ratio of data to metadata. The ratio you choose may be different depending on your application usage. The 90/10 ratio is probably good for applications that use large files, such as database applications. Applications that use lots of small files with frequent file creations/deletions or large directories, such as file servers may need more space for metadata, so a 60/40 or 50/50 ratio may be appropriate.

# **Semaphore Tables**

**Steve Albert**  
Hewlett Packard Company

## Semaphore Tables

Many third party applications, databases in particular, make heavy use of the semaphores (commonly referred to as System V IPC) available in HP-UX. The installation guides often recommend changing the tunables associated with semaphores. The purpose of this paper is to describe the memory impacts that are caused by changing these tunables.

This paper will address the following questions regarding semaphores:

1. What is a semaphore?
2. What interfaces are used to access semaphores?
3. What tunables affect semaphore memory usage?
4. How much memory is used to manage the semaphore tables?
5. Are there any guidelines for configuring the semaphore tables?

### What is a Semaphore?

Although this paper will not go into the details of implementation, it is important to understand what a semaphore is, how it is used, and what the various tunables actually control.

In simple terms, a semaphore is a construct that is used to control access to a set of resources. It has a count associated with it that determines how many resources are available. Resources are allocated by decrementing the count and returned by incrementing the count. Of course, all of the updates to the semaphore are done atomically through system calls. When all of the resources have been given out, the next process that attempts to allocate one will go to sleep until a resource is freed, at which point it will be awakened and granted access.

A simple example will suffice to make this clear. Suppose we have three stations that can serve us. We can control access by having 3 tokens. The first person takes a token, then the second person, then the third person. Each of these people can be served simultaneously. If a fourth person wishes to be served, he/she must wait until a token becomes available. When any person finishes, the token is returned and the next person in line can take it. Of course, the degenerate case is when the count is set to 1, which is how one could implement exclusive access. This is the basic idea and is the mechanism databases frequently use to control concurrent access.

### Interfaces

There are three main interfaces that are used to manage semaphores – **semget()**, **semctl()** and **semop()**. **semget()** is the routine used to allocate a set of semaphores. Note that you can allocate more than one semaphore at a time and operations can be conducted on these sets. A successful call to **semget()** will return a semaphore identifier. **semop()** is the routine that is used to operate on a set of semaphores. Finally, **semctl()** is the routine that is used to perform control operations on the semaphore set. For more details, the reader should look at the manual pages for these calls.

## Tunables

Next, let's look at all of the existing tunables related to semaphores. Much of this information is available on the Hewlett-Packard documentation website at <http://www.docs.hp.com>. Simply follow the links to dynamic kernel tunables. There are 9 tunables related to semaphores, which are listed below along with a brief description of what each one does:

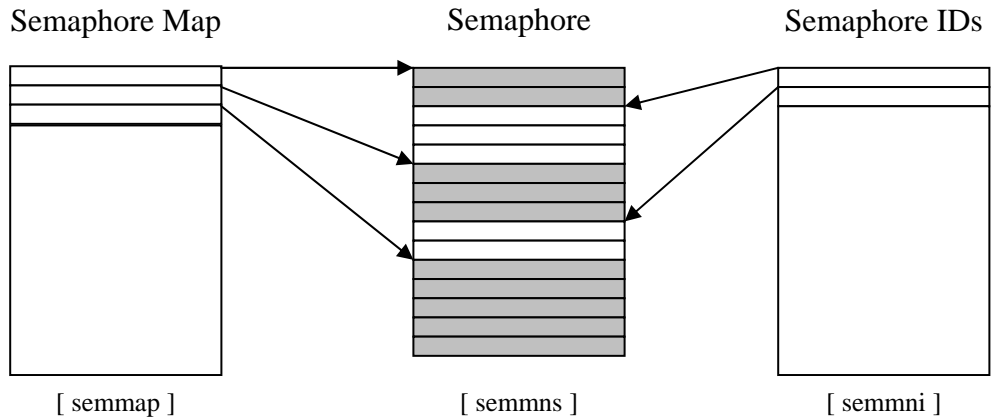
|               |   |
|---------------|---|
| <b>sema</b>   | enable or disable System V IPC semaphores at boot time            |
| <b>semaem</b> | maximum cumulative value changes per System V IPC semop() call    |
| <b>semmap</b> | number of entries in a System V IPC semaphore map                 |
| <b>semmni</b> | number of System V IPC system-wide semaphore identifiers          |
| <b>semmns</b> | number of System V IPC system-wide semaphores                     |
| <b>semmnu</b> | maximum number of processes that can have undo operations pending |
| <b>semmsl</b> | maximum number of System V IPC semaphores per identifier          |
| <b>semume</b> | maximum number of System V IPC undo entries per process           |
| <b>semvmx</b> | maximum value of any single System V IPC semaphore                |

Of the tunables listed above, **sema** simply enables/disables the semaphore feature and **semaem** and **semvmx** represent limits associated with the values in the semaphores. These tunables do not affect semaphore memory usage. That leaves the following list for consideration: **semmap**, **semmni**, **semmns**, **semmnu**, **semmsl**, and **semume**.

## Static Semaphore Tables

Until recently, the semaphore tables were statically sized, meaning the entire tables were allocated in memory when the system booted. Semaphores are allocated in groups of 1 or more from the Semaphore Table using the **semget()** system call. The **semget()** system call returns a semaphore ID from the Semaphore ID Table which is used to access the defined group of semaphores. The semaphores in a group are in contiguous memory, so there is also a Semaphore Map that describes which ranges of semaphores are currently free. The semaphore undo structures will be described later.

Below is a simple diagram to show the relationship between the Semaphore Map table, the Semaphore ID table, and the Semaphore Map table. Note that the grayed out semaphores are free and available for use:



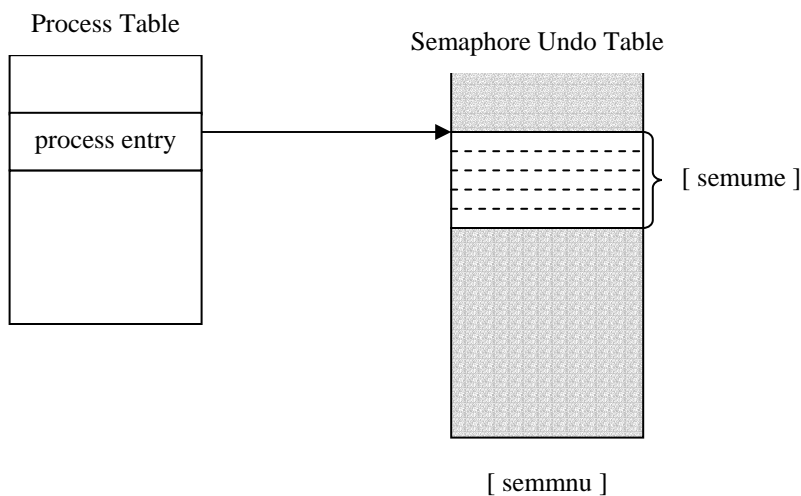
The **semmap** tunable is used to size the Semaphore Map table. Each entry in the array is 8 bytes, so each increment to **semmap** will increase the boot time kernel size by 8 bytes.

The **semmni** tunable is used to size the Semaphore ID table that maintains the system wide semaphore identifiers. Each entry in the array is 96 bytes, so each increment to **semmni** will increase the boot time kernel size by 96 bytes.

The **semmns** tunable is used to size the Semaphore Table that keeps track of the system wide semaphores. Each entry in the array is 8 bytes, so each increment to **semmns** will increase the boot time kernel size by 8 bytes.

The **semmsl** tunable defines the maximum number of semaphores per semaphore identifier. It does not size a specific table.

The Semaphore Undo table is slightly more complex. When a process terminates via a process abort or signal, the effects of a semaphore operation may need to be undone to prevent potential applications hangs. The system needs to maintain a list of pending undo operations on a per-process basis.



The **semume** tunable specified the number of semaphore undo operations pending for each process. Each semaphore undo structure is 8 bytes in size, and there are 24 bytes of overhead for each process. Thus increasing semume increases the semaphore undo table by 8 bytes per process.

The **semmnu** tunable specifies the number of processes that can have semaphore undo operations pending. Increasing semmnu increases the size of the Semaphore Undo table based on the value of semume. The following formula can be used to calculate the size of the Semaphore Undo table:

$$((24+(8*\text{semume})) * \text{semmnu})$$

It is important to keep this formula in mind when changing these tunables since the effects on the boot time kernel size are multiplicative. How many undo structures does one need? Well, there is no single answer to that, but it is important to note that undo structures are only allocated if the application specifies the SEM\_UNDO flag. Thus, it is not necessary to bump these values unless the application installation guides instruct you to do so. How can one go astray? Here is a real life example – a customer tuned **semmni** to 10000 and decided to set **semume** and **semmnu** to 10000 also. The first change was not significant, consuming only 80,000 bytes. However, the second set of changes resulted in the allocation of  $((24+(8*10000))*10000)$  or 800,240,000 bytes! This was clearly not what the customer really wanted.

The following table is a summary of memory usage based on the semaphore tunables:

| Kernel Table         | Tunable | Element size             | Default setting            |
|----------------------|---------|--------------------------|----------------------------|
| Semaphore Map        | semmap  | 8 bytes                  | semmni+2                   |
| Semaphore Ids        | semmni  | 88 bytes                 | 64                         |
| Semaphore Table      | semmns  | 8 bytes                  | 128                        |
| -                    | semmsl  | -                        | 2048                       |
| Semaphore Undo Table | semmnu  | 24 bytes +<br>(8*semume) | 30 (semmnu)<br>10 (semume) |

## Dynamic Semaphore Table

The static Semaphore Table has the disadvantage of consuming memory when the semaphores are not used since the entire table is allocated when the system boots up. Also, as semaphore sets with varying number of semaphores are created and removed, the Semaphore Table can become fragmented since each set of semaphores created by semget() must be contiguous.

Beginning with PHKL\_26136 on 11.0 and PHKL\_26183 on 11.11, the Semaphore entries that make up the Semaphore Table are now allocated dynamically. By allocating semaphores dynamically, the system can reduce memory usage previously caused by allocated but unused

semaphores as well as resolve the fragmentation issues that can occur in the Semaphore Table. Other tables such as the Semaphore Undo Table are still static.

With dynamic semaphore tables, the semaphore map is no longer used, thus the **semmap** tunable is effectively obsolete. However, space is still allocated for the table at boot time. Since it is unused, though, there is no need to change this tunable once these patches are installed, and in fact, it could be tuned down to take less space (although this table consumes very little space so the savings are miniscule).

The other significant change is related to the semaphore table itself. **semmns** represents a limit to the number of semaphores that can be allocated in total. They are allocated as needed and freed when not in use. There is a caveat here just like above. The boot time table is still allocated even though it is not used. As a result, you end up allocating space for **semmns** semaphores plus allocating space at run time for the semaphores that are currently in use. One other change of note was put into PHKL\_28703 (11.11), which lifts the restriction of 32767 on **semmns**. With this patch installed, the new limit is 1,000,000.

## **Guidelines for configuring semaphores**

Many third party applications make heavy use of the SysV IPC semaphore mechanism available within HP-UX. Understanding all of the tunables, what they represent, and how changing them can affect the memory consumed by your kernel is important so that you do not mistakenly tune the semaphores too high, leaving too little memory to accomplish other important tasks.

One of the most common mistakes is the setting of the Semaphore Undo table. Remember that there is a multiplied affect of setting **semmnu** and **semume**.

Also if you manage multiple systems, be sure the semaphore tables are scaled appropriately given the memory size of each. A given set of tunables may work great on a system with 8 Gb of memory, but have severe memory implications on a system with only 1 Gb of memory.