H-0166 November 26, 2002
Other (Network Attached Storage)

IBM Research Report

Accessing Application Identification Information in the Storage Tier

Tsipora Barzilai, Gala Golan
IBM Research Division
Haifa Research Laboratory
Haifa 31905, Israel

Accessing Application Identification Information in the Storage Tier

Disclosure IL8-2002-0055

Version 1.0 – November 26, 2002

Tsipora Barzilai 853101, Gala Golan 413990

IBM Haifa Labs

Abstract

Work Load Management (WLM) is a methodology for dynamic relocation and tuning of computing resources, so that distributed work can meet end-to-end performance goals. WLM and QoS are policy based management methodologies. Work (requests, transactions, messages, etc.) is classified into service classes. Each service class is associated with a set of policies, which determine the allocation of resources to the work in the class. Classification is based on diverse attributes, some of which are governed by the application originating the work. In order to classify a unit of work, one has to determine the values of relevant attributes. However, in Linux 2.4 (and may be other systems) those values may not be available at any point while processing the work. In particular we refer to the asynchronous communication between the buffer cache associated with the file system and the I/O device. Most of the requests on this interface don't carry any information on who application originated the work.

Accessing application identification information from within the I/O device entails new mechanisms and modifications to the operating system kernel. This document describes a mechanism for accessing application information from the storage tier in the Linux operating system. The mechanism has been experimentally implemented in the Linux 2.4
kernel.

1. Problem

Work Load Management (WLM) is a methodology for dynamic relocation and tuning of computing resources, so that distributed work can meet end-to-end performance goals. Among the resources that can be monitored are: CPU, memory, network, and i/o. An example of this approach is the IBM eWLM software[1], which has been implemented on several platforms, including Linux, AIX, and zOS. Quality of Service (QoS) is a general approach for preferential treatment of traffic flows, which has been applied to network traffic, using, among others, Differentiated Services (DS)[2].

WLM and QoS are policy based management methodologies. Work (requests, transactions, messages, etc.) is classified into service classes. A service class is a collection of units of work having similar performance goals. Each service class is associated with a set of policies, which determine the allocation of resources to the work in the class.

Classification is based on diverse attributes, some of which are governed by the application originating the work. Specifically, classification often requires application identification information, such as: application name, process id, user id, etc. The eWLM management methods are predominantly based on application identification.

In order to classify a unit of work, one has to determine the values of relevant attributes in the context of the environment in which the classification is applied. This may not always be a simple task. One example is classifying i/o work within the storage tier (cache, SCSI, iSCSI, etc). In this environment, information about the user application that generated the I/O is frequently not available. A decision is made by the file system, whether to generate a synchronous or asynchronous I/O request. In the latter case the user application is disconnected from the lower layers of the I/O stack. For example, the SCSI initiator layer in the kernel usually runs in a dedicated system process, which has no notion of the applications originating the requests.

Accessing application identification information from within the storage tier entails new mechanisms and modifications to the operating system kernel. This document describes a mechanism for accessing application information from the storage tier in the Linux operating system. The mechanism has been experimentally implemented in the Linux 2.4 kernel.

2. Solution

When a process requests data from a file, the file system is responsible for locating that data on the device. READ Data that is in the local buffer cache and up-to-date will be returned immediately. Otherwise, I/O is needed, and an I/O request is passed through the storage tier to the device queue for the actual I/O to take place.

In case of accessing a SCSI disk, the SCSI command is generated by the device driver only after the file system has placed the request. At this stage the file system data structures (which hold the information about the process, user and application) are not always available. A few changes enable us to ensure access to this information on every I/O request generated by the file system.

The changes described in the sequel have been implemented as a prototype in the Linux 2.4. kernel.

2.1. Accessing the Buffer Header

In some cases (mainly synchronous write operations), the application requesting the I/O would like to wait for the operation to complete before continuing. For this to happen, the buffer header is added to the request wait-queue, and the process state is changed from TASK_RUNNING to TASK_UNINTERRUPTIBLE. When the I/O completes and the device driver has been interrupted, a special completion function will wake up the process, and the application will continue running.

The SCSI command structure holds a pointer to the I/O request posted by the file system. The request structure (**`struct request`**), contains the queue **(`request_queue_t`**) in which the buffer header is placed, and through this header one can reach the process information. Yet, this queue is non-null only if the process is indeed waiting for the request to complete.

Using this sequence of pointers, it is possible to access the process id of the application process that generated the I/O request.

2.2. Making the Buffer Header Available at All Times

The goal is to make the process information available for all I/O requests. Since access is via the request queue, the obvious solution is to add all buffers to this queue, enabling access to them. However, this would force all I/O requests in the system to be synchronous, and no process could ever run while allowing I/O to complete in the background.

A better solution is to add the buffers to the queue without changing the process state, thus making the information accessible without forcing a delay in the application's normal run. In addition, it is necessary to change the request completion function, so that no running processes will be awakened.

2.3. Implementation Issues

2.3.1. Scope of the Changes

The changes are done in generic functions, which are used by various file systems, and during boot as well. The purpose is to make the changes flexible, so they can be used for selective mount points. It was decided to activate the changes only on explicit demand. This is easily done using a mount option. A new option, called **`pid`**, was added to the mount command. The option results in adding a flag to the super block of the file system. The setting of this flag will trigger the activation of the new mechanisms.

2.3.2. Structure of the Queue

Theoretically, more that one buffer can wait in the same queue, so it must be ensured that the correct buffer is deleted from the queue without awakening its process. Since there is no reserved field in which to save a pointer to the header, one has to rely on two Linux implementation facts for correctness.

1. Before posting a request for a buffer, the file system makes sure there are no requests pending for that buffer. This means that the buffer will not be waiting on any other queue when it is added to our queue.

2. The queue is implemented as a linked list. The head of the list is in the request structure, and new buffers are always added to the beginning of the list. Since the buffer is the first one to be added to the list, it will be the last one on it when the completion function is called. One can simply remove the last buffer from the queue before letting the original function work on the queue. In practice, no other buffer will be added to the queue because of item 1, so the buffer is the only one on the list.

References

[1] V. Chase, *eWLM: Enterprise Workload Management*, IBM Think Research, March 2002. http://www.research.ibm.com/thinkresearch/pages/2002/20020529_ewlm.shtml

[2] Blake, S., et al., *An Architecture for Differentiated Services*, Network Working Group RFC 2475, December 1988.

Appendix A. References to the Code

1. Accessing the process ID through the SCSI command structure.

```
        scsi_command* sc;
        struct list_head* lh ;
        wait_queue_t* wqp ;
        int pid ;

            ...

        if ((sc->request).bh }(
          lh = ((((sc->request).bh)->b_wait).task_list).next ;
          wqp = list_entry(lh, wait_queue_t, task_list ;(
          if (wqp(
            if (wqp->task)
               pid = wqp->task->pid  ;
      {
```

2. Changes in fs/buffer.c:

There are three functions in which the change had to be made.

```
      __block_write_full_page()
      block_read_full_page()
      brw_page()
```

- Assigning the new completion function:

```
      if (test_opt(inode->i_sb,PID))
        bh->b_end_io = end_buffer_io_async_pid;
      else
         bh->b_end_io = end_buffer_io_async;
```

- Adding the buffer to the queue:

```
      if (test_opt(inode->i_sb,PID((
      {
            wait = kmalloc(sizeof(wait_queue_t), GFP_KERNEL ;(
            wait->task=current;
            (wait->task_list).prev = NULL ;
            (wait->task_list).next = NULL ;
            add_wait_queue(&bh->b_wait, wait);
      {
```

- Creating a new (modified) completion function:

```
      lh = (bh->b_wait).task_list.prev ;
            wqp = list_entry(lh, wait_queue_t, task_list) ;
      remove_wait_queue(&bh->b_wait, wqp);
      kfree(wqp);
```

Appendix B. Relevant Linux Data Structures and Macros

Functions and macros for locking of the buffer header:

fs/buffer.c:

```
void __wait_on_buffer(struct buffer_head * bh)
}
      struct task_struct *tsk = current;
      DECLARE_WAITQUEUE(wait, tsk);


      atomic_inc(&bh->b_count);
      add_wait_queue(&bh->b_wait, &wait );
      do {
            run_task_queue(&tq_disk );
            set_task_state(tsk, TASK_UNINTERRUPTIBLE);
            if (!buffer_locked(b))
                  break;
            Schedule();

      }while (buffer_locked(bh));
      tsk->state = TASK_RUNNING;
      remove_wait_queue(&bh->b_wait, &wait );
      atomic_dec(&bh->b_count);
}
```

lock.h:

```
extern inline void wait_on_buffer(struct buffer_head * bh)
{
             if (test_bit(BH_Lock, &bh->b_state))
                         __wait_on_buffer(bh);
}

extern inline void lock_buffer(struct buffer_head * bh)
{
             while (test_and_set_bit(BH_Lock, &bh->b_state))
                         __wait_on_buffer(bh);
}

extern inline void unlock_buffer(struct buffer_head *bh)
{
             clear_bit(BH_Lock, &bh->b_state);
             smp_mb__after_clear_bit();
             if (waitqueue_active(&bh->b_wait))
                         wake_up(&bh->b_wait);
}
```

Data structures:

include/linux/fs.h:
```
struct buffer_head }
...
      char * b_data;                /* pointer to data block */
      struct page *b_page;          /* the page this bh is mapped to */
      void (*b_end_io)(struct buffer_head *bh, int uptodate); /* I/O
completion */
...
      wait_queue_head_t b_wait;
      struct inode *         b_inode;
      struct list_head     b_inode_buffers;     /* doubly linked list of
inode dirty buffers */
;{
```

drivers/scsi/scsi.h:
```
struct scsi_cmnd {
             struct Scsi_Host *host;
             unsigned short state;
             unsigned short owner;
             Scsi_Device *device;
...
             unsigned int target;
             unsigned int lun;
...
             /* These elements define the operation we are about to perform*/
             unsigned char cmnd[MAX_COMMAND_SIZE];
...
             struct request request;       /* A copy of the command we are
                                                     working on */
...
};
```

include/linux/blkdev.h:
```
struct request}
...
        struct buffer_head * bh;
        struct buffer_head * bhtail;
        request_queue_t *q;
};
```

include/linux/sched.h:
```
struct task_struct {
             volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
...
             pid_t pid;
...
/* process credentials */
             uid_t uid,euid,suid,fsuid;
             gid_t gid,egid,sgid,fsgid;
             int ngroups;
             gid_t            groups[NGROUPS];
             kernel_cap_t   cap_effective, cap_inheritable, cap_permitted;
             int keep_capabilities:1;
             struct user_struct *user;
/* limits */
             struct rlimit rlim[RLIM_NLIMITS];
             unsigned short used_math;
             char comm[16];
...
};
```

<u>include/linux/ext2_fs_sb.h:</u>

```
struct ext2_sb_info {
             ...
             unsigned long  s_mount_opt;
             ...
};
```

List and queue handling:

<u>include/linux/list.h:</u>

```
struct list_head {
             struct list_head *next, *prev;
};

#define LIST_HEAD_INIT(name) { &(name), &(name) }
...
/**
 * list_entry - get the struct for this entry
 * @ptr:           the &struct list_head pointer.
 * @type:          the type of the struct this is embedded in.
 * @member:        the name of the list_struct within the struct.
 */
#define list_entry(ptr, type, member) \
             ((type *)((char *)(ptr)-(unsigned long)(&((type *)0)->member)))
```

<u>include/linux/wait.h:</u>

```
struct __wait_queue {
             unsigned int flags;
#define WQ_FLAG_EXCLUSIVE            0x01
             struct task_struct * task;
             struct list_head task_list;
};
typedef struct __wait_queue wait_queue_t;

struct __wait_queue_head {
             wq_lock_t lock;
             struct list_head task_list;
};
typedef struct __wait_queue_head wait_queue_head_t;
static inline void __add_wait_queue_tail(wait_queue_head_t *head,

wait_queue_t *new)
{
             list_add_tail(&new->task_list, &head->task_list);
}

static inline void __remove_wait_queue(wait_queue_head_t *head,

       wait_queue_t *old)
{
             list_del(&old->task_list);
}
```