Kernel mode database integration for high performance applications

By Andrei Gorine and Alexander Krivolapov
McObject (www.mcobject.com)
33309 1st Way South, Suite A-208
Federal Way, WA 98003

Introduction

At the heart of many operating systems is the kernel, responsible for resource allocation, scheduling, low-level hardware interfaces, network, security and other integral tasks. Increasingly, system software vendors place application functionality there, to accelerate overall system performance. For example, firewalls and other security applications (access control systems, etc.) must run their policy engines as kernel components, to achieve needed performance. Another common example is a system monitor commonly supplied with operating systems. The OS monitors usually are hooked up to various system resources inside the operating system's kernel and provide user-mode applications with the ability to fine-tune specific aspects of the operating system environment.

These and similar high-performance application architectures increasingly require sophisticated data management functions. In the access control application scenario mentioned above, the data structures and data query algorithms are inherently complex, yet the lookups and updates must be nearly instantaneous. The system monitor software needs to provide a way for storing data collected from the various user-level processes, device drivers and kernel modules. Such data is usually transient. Once analysis has been performed, the data is no longer needed (except, possibly, for some archival purposes).

Despite the growing need for sophisticated kernel-based data management, the possibility of integrating a complete "off-the-shelf" database management system (DBMS) into the kernel has long been rejected, out of recognition that DBMS overhead, such as file and disk I/O, cache management, buffer management and related logic could overwhelm kernel resources and disrupt OS-critical tasks.

The challenge

Consider the requirements for any data management code that operates inside the kernel, whether this module is custom-made or a commercial off-the-shelf (COTS) database. Any kernel-mode driver or application component has to be non-intrusive to the system. Therefore, the integrated data management cannot write data to a hard disk, even on systems where a large file system cache is present, because the disk I/O upon transaction commit would cause too much impact on the system. Data management can't monopolize or extensively use any of the system's resources, increase interrupt latencies, or noticeably affect overall system responsiveness to outside events. Furthermore, it has to provide simultaneous access to data for all parts of the system, including from multiple user-mode processes and kernel threads. Further, this data access must include both write and read access and must be efficient enough to avoid stalling the kernel.

Kernel module developers have been left with few choices. One is "re-inventing the wheel" of fundamental database capabilities, albeit in a very limited and lightweight fashion, for use in the kernel. The other approach is to deploy a complete DBMS in user-mode space and rely on expensive (in performance terms) context switches whenever kernel-mode processes require data lookup.

The relatively recent advent of embedded in-memory database systems (IMDSs) presents a more attractive alternative, by making it possible to integrate a COTS database engine with the operating system kernel. By mapping IMDS run-time code directly into the kernel module address space, a low-overhead, yet full-featured, database engine can be integrated with kernel-mode software components. Moreover, the database runtime code is directly linked with the module, and any remote procedure calls are eliminated from the execution path. As a consequence, the execution path generally requires fewer CPU instructions. Such database runtime integration with the kernel provides the benefit of keeping the database in the kernel space, while enabling kernel threads to access the data via a lightweight programming interface. The databases are made available to user-mode applications through a set of public interfaces implemented via system calls (Figure 1)

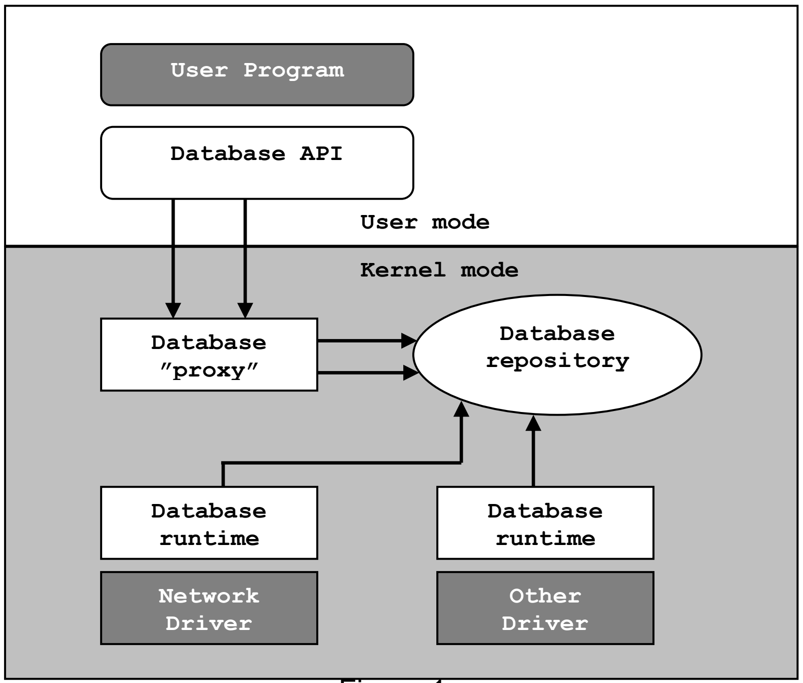

Figure 1

For such kernel-mode database integration to work, the kernel–based IMDSs must address many of the same challenges as kernel-based programs and embedded systems generally. The following issues and solutions involved in deploying a database in kernel space are also highly applicable to device drivers, file systems and other kernel modules. Challenges in kernel-mode database integration include:

- Synchronization primitive usage. In order to provide multi-threaded, simultaneous data access, the database runtime must use synchronization mechanisms. Nearly every driver requires synchronization mechanisms as well, because this function is needed for any shared data to be accessed by multiple threads, unless this access is read-only. Synchronization is also required when a set of operations must be performed atomically, inside a transaction. Operating system synchronization mechanisms, by their very nature, can cause performance bottlenecks. To improve locking performance, whenever possible, the kernel-mode database uses locks based the atomic exchange instructions provided by many modern CPU architectures. When the nature of the resource requires mutual exclusion between threads, the database runtime claims resources only for a short time. Thus the database locks use a low-overhead spinlock-based mechanism that protects the resource.

- Memory usage. Efficient memory usage is often a key to application performance, especially in the kernel, where the non-paged memory pool is limited and frequent paging could increase kernel latencies. To address this, in-memory databases often use a number of custom allocation algorithms to take advantage of problem-specific allocation patterns, such as infrastructure for the data layout, internal database runtime heap management, etc.
- Stack usage. The kernel-mode stack is a limited storage area that is often used for information that is passed between functions, as well as for local variable storage. Running out of stack space will cause the operating system to crash. Therefore, the database runtime integrated with the kernel module and with other drivers must carefully watch the stack usage. It must never allocate large aggregate structures on the stack, and avoid deeply nested or recursive calls; if recursion must be used the number of recursive calls must be strictly limited.
- C runtime. Not all of the standard libraries (C and especially C++) are present in kernel mode. Moreover, versions of standard libraries for use in kernel mode are not necessarily the same as those in user mode, as they are written to conform to kernel mode requirements. Kernel-mode implementations of standard libraries usually have limited functionality and are constrained by other properties of kernel mode. The kernel mode database runtime benefits tremendously by not using C the runtime at all. For instance, instead of relying on the C runtime for memory management, the database replaces those functions with custom allocators.

Reference application

The reference application presented here implements a rudimentary access control system. It utilizes McObject's *eXtreme*DB embedded in-memory database system to create and maintain the access control database in the kernel space. The database keeps file access rules and the runtime provides drivers and user-level applications with high-performance access to the storage. The example code shown in this paper uses UNIX-like notations; the application's source code (excluding the database runtime) is available for free download to anyone who is interested on McObject's website at www.mcobject.com/EmbeddedWorld_2007/kernelmode.php

The application contains three major components (Figure 2):

- a "database" kernel module, based on *eXtreme*DB and responsible for storage, maintains database access logic
- a kernel module that intercepts file system calls and provides a file access authorization mechanism to the system. This module is referred to as a "filter" module
- a user-mode application that implements a user-mode database API

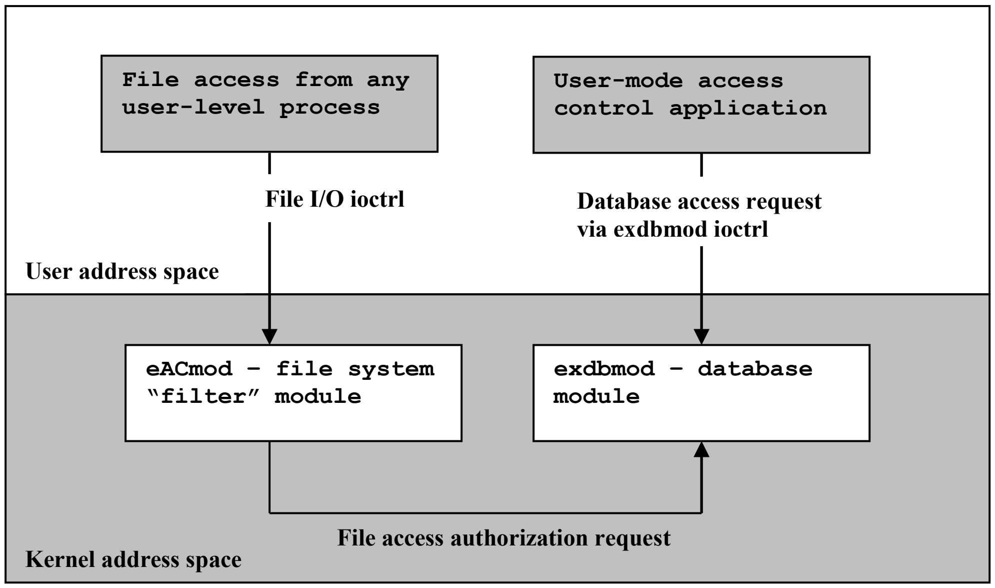
File access from any user-level process
User-mode access control application
File I/O ioctrl
Database access request via exdbmod ioctrl
User address space
eACmod – file system "filter" module
exdbmod – database module
File access authorization request
Kernel address space

Figure 2

The database kernel module implements kernel-mode data storage and provides the API to manipulate the data. The module is integrated with the *eXtreme*DB database runtime, which is responsible for providing "standard" database functionality such as transaction control, data access coordination and locking, search algorithms, etc. Figure 3 shows the data layout using the *eXtreme*DB Data Definition Language syntax.

```
struct ACL
{
  uint4   uid;       // user id
  uint4   access;    // access allowed for some user id
};

class File
{

  char<100> name;   // file name
  uint4  inode;     // file inode
  uint4  device;    // device
  uint4  owner;     // owner of the File
  uint4  defaccess;
  vector< ACL > acls; // access control lists

  hash < name >          hname[4096];
  hash < inode ,device > hfile[4096];
};
```

Figure 3 (database schema)

The class File describes a *file object* that is identified by the file's name, and the inode and device the file is located on. The rest of the fields (owner, defaccess and acl vector) are used to define file access rules. The database maintains two hash-based indices that facilitate fast data access.

The database itself could grow large. Therefore the database pool is allocated in virtual memory. In order to use the allocated memory pool, it is mapped to the physical page (Figure 4 and 5). Once the memory is allocated, the in-memory database is created and supports connections using standard database runtime functions.

```
/* allocate the database memory pool */
mem_ua_ptr = (char*)vmalloc( arg+PAGE_SIZE*2 );
if ( mem_ua_ptr == 0 ) {
     return -ENOMEM; /* error allocating memory */
}
```

Figure 4 (allocating virtual memory for the database pool)

```
/* calculate page aligned address */
mem_ptr = (char*) ( ((unsigned long)mem_ua_ptr+PAGE_SIZE-1) & PAGE_MASK );

/* lock pages */
for ( va=(unsigned long)mem_ptr;
       va<(unsigned long)(&(mem_ptr[arg/sizeof(int)]));
       va+=PAGE_SIZE) {
     mem_map_reserve(virt_to_page(virt_to_kseg((void *)va)));
}
```

Figure 5 (locking virtual memory pages)

The module exports two types of interfaces: the “direct” API available to other kernel modules and drivers; and the indirect API that implements the system call interface to the database. The direct API is not available for user-mode processes, but is extremely fast because it maintains only kernel-space references and eliminates translations from kernel address space to user address space. In order to implement the “indirect” system call API, during its initialization the module registers a number of I/O controls (Figure 6). Regardless of the interface type, these APIs completely hide all database access details from kernel modules and user-mode applications.

```
/* I/O controls registered by the database access module */
#define EXDB_IO_INITIALIZE              10     /* arg - memory amount */
#define EXDB_IO_CLEAR_UP                11

#define EXDB_IO_ADD_FILE                12     /* arg points to File_t */
#define EXDB_IO_REMOVE_FILE_BY_NAME     13     /* arg points to File_t */
#define EXDB_IO_FIND_FILE_BY_NAME       14     /* arg points to File_t */
#define EXDB_IO_AUTHORIZE_FILE          15     /* arg points to SetACL_t */
```

Figure 6 (ioctls)

The filter module intercepts calls to the file system and replaces standard file access functions with its own, providing authorization. The implementation involves registering the custom module’s file access functions upon module initialization (Figures 7 and 8). In turn, these custom functions use the database access API exposed by the database module to authenticate file access (Figure 9).

```
static int __init eACmod_init( void )
{
  if (!sys_call_table)
  {
    return -1;
  }
  if ( (major = register_chrdev( 0, DEV_NAME, &eACmod_fops )) < 0 )
  {
    return -EIO;
  };

  intercept_syscalls();
  return 0;
};
```

Figure 7 (module initialization)

```
extern void *sys_call_table[];
typedef int (*syscall_t)();

extern int my_open();
extern int my_creat();
extern int my_chmod();
extern int my_chown();
extern int my_unlink();
extern int my_execve();

struct replace_syscall replace_syscall[]={
       {INDEX_NR_open,     __NR_open,    (int(*)())0,   my_open},
       {INDEX_NR_creat,    __NR_creat,   (int(*)())0,   my_creat},
       {INDEX_NR_chmod,    __NR_chmod,   (int(*)())0,   my_chmod},
       {INDEX_NR_chown,    __NR_chown,   (int(*)())0,   my_chown},
       {INDEX_NR_unlink,   __NR_unlink,  (int(*)())0,   my_unlink},
       {INDEX_NR_execve,   __NR_execve,  (int(*)())0,   my_execve},
       {-1,                -1,           (int(*)())0,   (int(*)())0}
};

int nreplace_syscall = sizeof(replace_syscall)/sizeof(*replace_syscall)-1;

void intercept_syscalls()
{
  int i, f;

  for(i = 0; i < nreplace_syscall; i++)
  {
    f = replace_syscall[i].index;
    replace_syscall[i].original = sys_call_table[f];
    sys_call_table[f] = replace_syscall[i].seos_syscall;
  }
}
```

Figure 8 (intercept_syscalls)

```
asmlinkage int my_open(const char* fname, int fmode, int mode)
{
  int access, rv;
  /* some processing */


  rv = replace_syscall[INDEX_NR_open].original(fname, fmode, mode);
  return rv;
}
```

Figure 9 (my_open() example)

Finally, the user-level application's component creates a user-level database access API exposed by the database driver via a system call interface. This API allows user-mode processes (such as administrative applications) to interact with the kernel database. The API contains functions that correspond to the I/O controls exposed by the database module (Figures 10 and 11)

```
#ifndef __EX_DB_API_H
#define __EX_DB_API_H

int exdb_init_api (  );
int exdb_shutdown_api(  );

int exdb_init_database ( unsigned long mem_size );
int exdb_shutdown_database(  );
int exdb_add_file( char* file_name, unsigned int inode, unsigned int device,
                   unsigned int owner, unsigned int defaccess );
int exdb_remove_file( char* file_name );
int exdb_find_file( char* file_name, unsigned int *pinode,
                    unsigned int *pdevice, unsigned int *powner,
                    unsigned int *pdefaccess );
int exdb_authorize_file( char* file_name, unsigned int uid,
                         unsigned int access );
#endif /* __EX_DB_API_H */
```

Figure 10 (user-mode database access API implemented via ioctl)

```
int exdb_find_file( char* file_name,
                    unsigned int *pinode, unsigned int *pdevice,
                    unsigned int *powner, unsigned int *pdefaccess )
{

  File_t f;
  int    r;

  memcpy( f.name, file_name, strlen(file_name) + 1);
  r = ioctl( fd, EXDB_IO_FIND_FILE_BY_NAME, &f );

  if ( r == 0 )
 {
    *pinode     = f.inode;
    *pdevice    = f.device;
    *powner     = f.owner;
    *pdefaccess = f.defaccess;
    return f.result;
  };

  return r;
};
```

Figure 11 (an example of a user-mode function accessing kernel database)

Conclusion

With the approach presented in this paper, applications are able to take advantage of a full set of database features—including transaction processing, multi-threaded data access, ability to perform complicated queries using built-in indexing, convenient data access API, and a high-level data definition language—while still providing the near-zero latency of a kernel-based software component. The memory-only nature of the database eliminates unpredictable disk I/O, while direct pointers to data elements prevent expensive buffer management and remote procedure calls that can introduce latency.

As a result, the kernel-mode database runtime remains non-intrusive and refrains from monopolizing system resources, does not increase interrupt latencies, or noticeably affect the overall kernel responsiveness. The query execution path for such a kernel mode database generally requires just a few CPU instructions. Concurrent access to the kernel data structures and complex search patterns are coordinated by the database run-time, and the kernel mode database is made available to user-mode applications by a set of public interfaces implemented via system calls.

About the Authors

Andrei Gorine is a Chief Technology Officer and Alexander Krivolapov is a Software Engineer at McObject. They can be reached at gor@mcobject.com and kid@mcobject.com respectively.