A Kernel Mode Database System for High Performance Applications

By Andrei Gorine and Alexander Krivolapov

Introduction

The evolution of database management systems has been driven by widely varying innovations, targeting different points in system architecture, to eliminate latency and increase the prioritization of data management tasks. New physical database structures including ISAM, VSAM, clustered indexes and column-based databases all responded to a need for faster data retrieval. Innovations in storage modality have also played a role: long ago, caching was introduced as a means to keep selected data in memory for faster access; in the past decade, in-memory database systems (IMDSs) offered the ability to store the entire database in main memory, eliminating the overhead and unpredictability of disk I/O and caching logic.

To a degree greater than any other software component, operating system kernels embody the kind of high-priority, zero-latency responsiveness sought by database system developers. Typically viewed as the lowest-level software abstraction layer, the kernel is responsible for resource allocation, scheduling, low-level hardware interfaces, network, security and other integral tasks. Certain software categories such as security applications (access control systems, firewalls, etc.) and operating system monitors commonly place their functions in the operating system kernel *and* have a need for local, high performance data sorting, storage and retrieval. For example, in the access control application scenario mentioned above, the data structures and queries are inherently complex, yet the lookups and updates must be nearly instantaneous.

Yet the kernel has generally been viewed as off limits for database management systems, out of recognition that DBMS overhead, such as file and disk I/O, locking, cache management, and related logic could overwhelm kernel resources and disrupt OS-critical tasks. Kernel module developers have been left with the choices of “re-inventing the wheel” of fundamental database capabilities, albeit in a very limited and lightweight fashion, for use in the kernel; or deploying a complete DBMS in user-mode space and relying on expensive (in performance terms) context switches whenever kernel-mode processes require data lookup.

The advent of the ultra-small footprint and resource-conserving *e*X*treme*DB embedded IMDS presents a more efficient alternative, by making it possible to integrate an off-the-shelf database engine with the operating system kernel. McObject has developed *e*X*treme*DB Kernel Mode (KM) Edition, the first commercial, off-the-shelf (COTS) database management system designed for such a deployment. *e*X*treme*DB-KM consists of the standard *e*X*treme*DB in-memory embedded database runtime adjusted for kernel usage. The adjustments are relatively minor: database locking is implemented via kernel mode spinlocks rather than through synchronization primitives available in the user space, such as semaphores and spinlocks; and the database runtime does not use the C runtime (which is an option for the “normal” *e*X*treme*DB as well). To facilitate the implementation of user-mode applications accessing the kernel-mode database, *e*X*treme*DB-KM provides a simple interface compiler utility that is conceptually similar to the well-known Remote Procedure Call Interface Definition Language compilers.

To illustrate its use, the *e*X*treme*DB-KM distribution includes a reference application that creates a kernel mode database to enforce access rules as part of an access control system. The reference design includes another 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). The filter module exports two types of interfaces: the “direct” API available to other kernel modules and drivers; and the “indirect” API that implements the *ioctl* interface to the database module.

The challenge

Data management code that operates inside the kernel faces considerable challenges, 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. 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.

The kernel mode database is made available to user-mode applications through a set of public interfaces implemented via system calls (Figure 1)

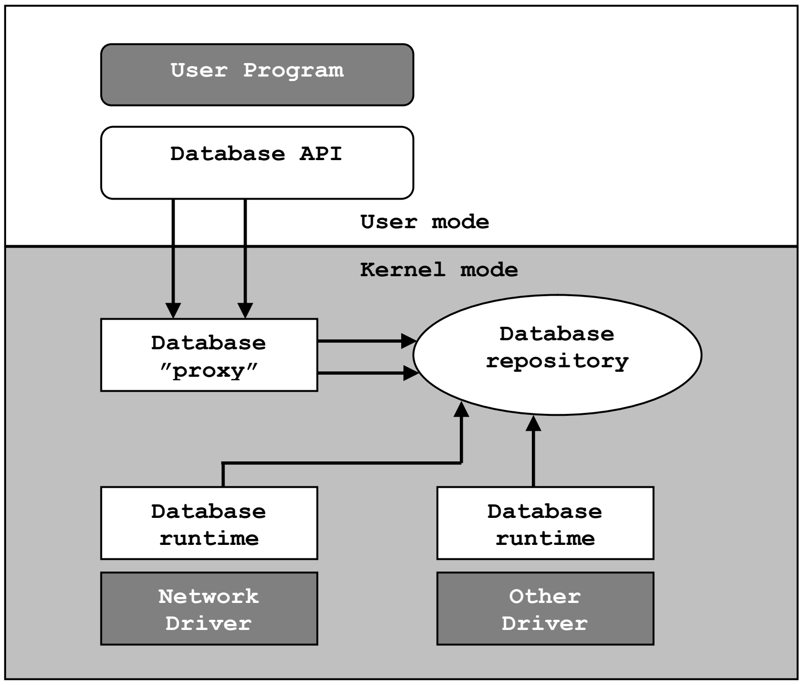

Figure 1

For the kernel-mode database integration depicted in Figure 1 to work, the kernel–based database runtime 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 on the atomic exchange instructions provided by many 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 *e*X*treme*DB-KM 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 basic access control system, utilizing *e*X*treme*DB-KM to create and maintain the access control database in the kernel space. The database maintains file access rules and the runtime provides drivers and user-level applications with high-performance access to the storage. The example code uses UNIX-like notations.

The application contains three major components (Figure 2):

- The *eXtreme*DB-KM database kernel module, responsible for storage, maintains database access logic
- a user-mode application that utilizes a user-mode database API
- the “filter” or kernel module that intercepts file system calls and provides a file access authorization mechanism to the system

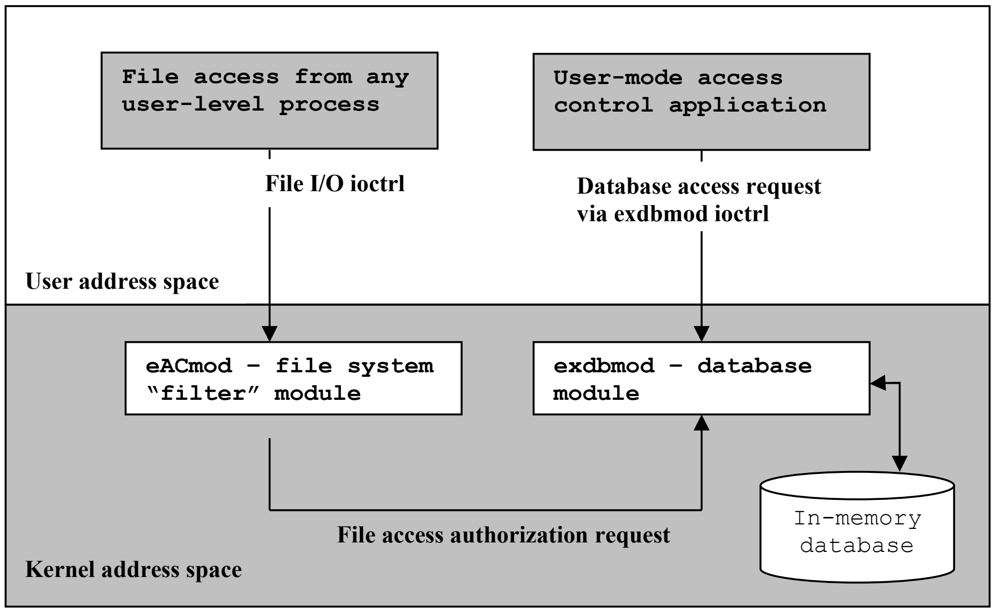
File access from any user-level process
User-mode access control application
File I/O ioctrl
Database access request via exdbmod ioctrl
User address space
eACmod – file system "filter" module
exdbmod – database module
In-memory database
File access authorization request
Kernel address space

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, lookup 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 on which it is located. 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[mem_size/sizeof(int)]));
             va+=PAGE_SIZE) {
       SetPageReserved(vmalloc_to_page((void *)(((unsigned long)va))));
    }
```

Figure 5 (locking virtual memory pages)

The module exports two types of interface: the "direct" API available to other kernel modules and drivers; and the "indirect" API that implements *eXtreme*DB-*KM*'s *ioctl* interface to the database module. The direct API is not available for user-mode processes, but is extremely efficient because it maintains only kernel-space references and eliminates expensive (in performance terms) translations from kernel address space to user address space. The *ioctl* interface provides user-mode applications with access to the kernel mode database.

The user-level application creates a user-level database access API exposed by the database module via a *ioctl* interface. This API allows user-mode processes (such as administrative applications) to interact with *eXtreme*DB-KM.

In order to facilitate the implementation of the "indirect" system call API, *eXtreme*DB-KM provides a simple *"interface compiler"* utility. The utility fills a need similar to that of a standard remote procedure call compiler, except that it generates the user/kernel mode interface files instead of remote access stubs. Developers define the API for the C language functions that the user-mode applications will use to access the kernel-mode database. The *eXtremeDB*-KM interface compiler generates interface files that implement the user-to-kernel-mode interface through the generic *ioctl* function. In particular, the interface compiler generates stub files that should be linked with the user-mode applications and the kernel-mode stubs that are included into the kernel module (Figure 6). The generated files encapsulate the user-to-kernel-mode transition and hide *ioctl* -based implementation details from kernel modules and user-mode applications.

In contrast with other interface definition language implementations, the *eXtremeDB*-KM interface compiler accepts standard C-language header files as a way to declare user-mode database access interfaces. The compiler recognizes a number of keywords in the form of comments that are used to declare string, union and array data types used as a part of the interface declaration. The interface definition file in Figure 7 illustrates the concept.

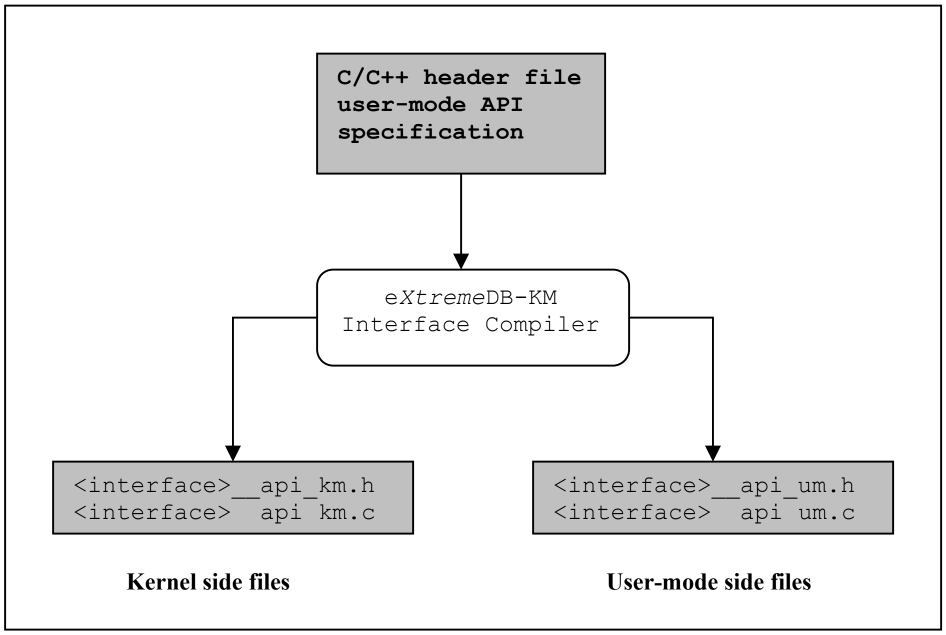

Figure 6 (e*XtremeDB*-KM interface compiler)

```
#ifndef __EX_DB_API_H
#define __EX_DB_API_H

typedef struct tagFile {
        unsigned char result;
        char     name[100];
        unsigned int  inode;
        unsigned int  device;
        unsigned int  owner;
        unsigned int  defaccess;
}File_t, *File_h;

typedef char* zstring    /*sero-terminated string*/;
typedef unsigned int* pint;

int exdb_init_database ( unsigned long mem_size );
int exdb_shutdown_database( );

int exdb_add_file
        (zstring file_name, unsigned int inode,
         unsigned int device, unsigned int owner,
         unsigned int defaccess );
int exdb_remove_file( zstring file_name );
int exdb_find_file( zstring file_name,
                   /*out*/ pint pinode,
                   /*out*/ pint pdevice,
                   /*out*/ pint powner,
                   /*out*/ pint pdefaccess );
int exdb_authorize_file( zstring file_name, unsigned int uid, unsigned int
access );

#endif /* __EX_DB_API_H */
```

Figure 7 (interface definition header file)

The interface compiler approach greatly simplifies access to databases created in the context of a kernel module. The user-mode application code that implements database access is almost undistinguishable from that used by the kernel-mode application, with the exception of a simple initialization step (Figure 8). There is no need for the user-mode application to serialize/de-serialize function parameters, and similar technicalities. The application only needs to define and implement its database access interface regardless of where the interface is used, inside or outside the kernel.

```
User mode

if ( 0 != (i = exdb_find_file( name, &inode, &device, &owner, &access ))) {
         printf("exdb_find_file(\"%s\")==%d\n", name, i );
         return (0);
 };

Kernel mode

if ( 0 != (i = exdb_find_file( name, &inode, &device, &owner, &access ))) {
         pr_info(DEV_NAME" exdb_find_file(\"%s\")==%d\n", name, i );
         return (0);
 };
```

Figure 8

The third component of the reference application presented here is the filter module. The filter module intercepts calls to the file system and replaces standard file access functions with its own, providing the user application with authorization to obtain the sought-after system resource. The implementation involves registering the custom module's file access functions upon module initialization (Figures 9 and 10). In turn, these custom functions provide authentication. This is a rather standard technique that is used in numerous applications. However, the filter presented here benefits from using the database access API exposed by the *eXtreme*DB-KM based database module to authenticate file access (Figure 11).

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

  eACm_api_Init();

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

Figure 9 (intercept_syscalls)

```
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 10 (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 11 (my_open() example)

Conclusion

A kernel mode database system meets the data management needs of any type of application that must run at least partially in the OS kernel to accelerate overall system performance, yet needs sophisticated data management functions. Security applications must run their policy engines as kernel components, to achieve needed performance. System monitors commonly supplied with operating systems 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. 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).

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 *eXtreme*DB-KM database runtime remains non-intrusive, refrains from monopolizing system resources, and does not increase interrupt latencies or noticeably affect overall kernel responsiveness. The query execution path for the 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.

Andrei Gorine is the Chief Technology Officer at McObject and can be reached at gor@mcobject.com.

Alexander Krivolapov is a senior software engineer with McObject and can be reached at kid@mcobject.com.