DIGITAL ACCESS TO SCHOLARSHIP AT HARVARD
DASH.HARVARD.EDU

HARVARD LIBRARY
Office for Scholarly Communication

Reexamining Operating System Support for Database Management

Citation
Vasil, Tim. Reexamining Operating System Support for Database Management. Harvard Computer Science Technical Group TR-02-03.

Permanent link
http://nrs.harvard.edu/urn-3:HUL.InstRepos:23017277

Terms of Use
This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA

Share Your Story

The Harvard community has made this article openly available.
Please share how this access benefits you. Submit a story .

Accessibility

Reexamining Operating System Support for Database Management

Tim Vasil

TR-02-03

Computer Science Group
Harvard University
Cambridge, Massachusetts

FINAL PAPER

Reexamining Operating System Support for Database Management

Tim Vasil
vasil@fas.harvard.edu
CS 265: Database Systems
Harvard University

January 12, 2003

Abstract

In 1981, Michael Stonebraker [21] observed that database management systems written for commodity operating systems could not effectively take advantage of key operating system services, such as buffer pool management and process scheduling, due to expensive overhead and lack of customizability. The "not quite right" fit between these kernel services and the demands of database systems forced database designers to work around such limitations or re-implement some kernel functionality in user mode.

We reconsider Stonebraker's 21-year old observations in the context of a modern-day database system, Microsoft SQL Server 2000, and the commodity operating system for which it is explicitly designed, Microsoft Windows 2000. We show that operating system services have become more efficient and flexible so as to meet many of SQL Server's needs directly. We also identify areas where operating system services continue fall short of meeting the needs of a DBMS, and propose several enhancements to rectify these shortcomings.

1 Introduction

Twenty-one years ago Michael Stonebraker came down hard on the services provided by commodity operating systems, declaring them "either too slow or inappropriate." Indeed, key operating system services, such as buffer pool management, process scheduling, and locking primitives, were not designed to meet the efficiency and flexibility needs of a performance-driven database management system. Consequently, database system designers, including Stonebraker, grudgingly provided their own user-level implementations of such services in their software. For these designers, duplicating existing functionality was not only painful, it wasted valuable development, debugging, and maintenance time. The ideal solution, according to Stonebraker, was for an operating system to offer two levels of services: general-purpose services acceptable to most applications, and lower-level, customizable services that met the unique needs of performance-driven applications such as database systems.

Now, with over two decades having passed, we cite the need for a review of Stonebraker's criticisms. Not only have the demands placed on an operating system by a DBMS remained considerable, but entirely new classes of applications have arisen, such as web servers and directory servers, with similar demands. Consequently, determining how adequately an operating system supports the needs of these most demanding and increasingly popular applications has, if anything, has become an even more relevant question today. Surprisingly, little research has been dedicated to this area.

The conclusions we reach in this paper do not simply provide a "delta *x*" analysis of Stonebraker's work, they stand alone as relevant observations and criticisms of both application and operating system design. By revealing areas where applications do not take adequate advantage of improved operating system services, we help to identify and eliminate unnecessarily redundant code so as to reduce development and maintenance costs. Further, by revealing areas where operating systems continue to neglect the true needs of the most important and relevant applications of the day, we provide some direction for the further evolution of commodity operating systems. Finally, by conducting the research to investigate these two areas, we present performance metrics that may guide programming "best practices" as well as reflect on the overall effectiveness of commodity operating systems' one-size-fits-all design methodology.

Using our own custom benchmarks, we identify areas where modern operating systems now meet the needs of database applications, and also those areas where adequate support is still lacking. We hope that our research, and potential future research motivated by it, fosters a much better coupling between this class of high-performance applications and the operating systems upon which they rely.

1.1 Platform

Stonebraker's analysis focused on the needs of the INGRES DBMS and the services provided by the UNIX operating system, but his observations generalized to most of the database systems and commodity operating systems of the day. We likewise are motivated to choose a generalizable database/operating system combination so that our results have broad relevance. We aim to choose suitable parallels by considering a popular relational database system and a popular commodity operating system.

These criteria give us many choices, including the option of evaluating Oracle running on a Unix platform or that of evaluating Microsoft SQL Server 2000 running on the Microsoft Windows 2000 platform. For this paper, we choose the latter option for two important reasons. First, Yang and Li have already explored the suitability of operating system services for database systems on Sun Solaris platform [23], a flavor of Unix. Second, we discovered that SQL Server, unlike Oracle, is not designed to run on multiple platforms, enabling it to tightly integrate with the Windows operating system [6], just as INGRES tightly integrated with UNIX. Since the database is not limited to the least common denominator of services, which would likely be necessary if it were to support multiple platforms, we expect SQL Server to make better use of operating system services than Oracle, and thus provide us with the best case scenario of application-OS coupling from which to draw conclusions. At the same time, since Windows 2000 is a commodity operating system much like many of the flavors of Unix, we feel that the Windows 2000/SQL Server platform choice will have sufficient generalizability to other modern-day operating systems and the respective relational databases these operating systems support.

1.2 Windows 2000 Overview

Microsoft Windows 2000 is a symmetric multiprocessing (SMP), fully reentrant, preemptive multitasking operating system designed for both low-end client desktops and high-end datacenter servers. It maintains a flat 32-bit virtual memory address space, reserving the upper 2 GB for system memory and the lower 2 GB for private process memory.[1] Although available in four flavors—Professional, Server, Advanced Server, and Datacenter Server—the core codebase remains consistent. The more expensive flavors are distinguishable by their ability to utilize additional processors, physical memory, and concurrent network connections, as well as by the additional bundled software they ship with, including DHCP and DNS servers.

Windows 2000 supports exactly two modes of execution: kernel mode and user mode, even when the processor supports more. Only core operating system components and device drivers run in a privileged *kernel mode*, which permits access to all physical memory and system resources. All other code runs in unprivileged *user mode*, where its use of instructions and system resources are limited. The scheduling component distributes processing time at the thread level of granularity based on adjustable thread priority levels; it treats application and operating system threads identically.

The Windows 2000 kernel sits above a hardware abstraction layer (HAL) that provides a standard interface to hardware-dependent mechanisms such as acquiring locks, performing port I/O, and adjusting the interrupt request (IRQ) level. Processes interact with kernel services from user mode via the kernel's stub dynamic link library, typically via one of the three available environment subsystems: Win32, POSIX, or OS/2. Figure 1.2.1 summarizes the system's architecture.

The principal interface to Windows 2000 is the Win32 API, accessible from the Win32 environment subsystem. This API exposes all public kernel- and user-level services, and implements significant additional functionality, including the windowing and graphics system. The Win32 subsystem is the only environment subsystem required for Windows 2000 to run.

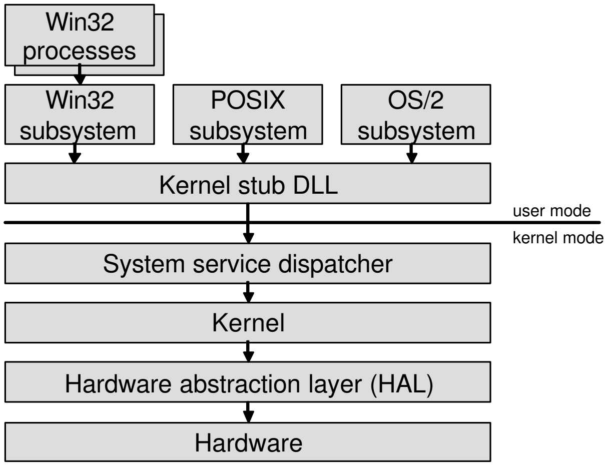

Figure 1.2.1

High-level view of the Windows 2000 architecture, (adapted from Inside Windows 2000, *figure 2-3* [18])

[1] To provide more memory to demanding server processes, Windows 2000 Advanced Server and Datacenter Server have a boot argument, /3GB, to limit operating system virtual memory to 1 GB so that processes may utilize 3 GB. For even more demanding server processes, Windows provides Address Windowing Extensions (AWE) that allows processes to map views of up to 64 GB of physical memory into their virtual address space.

1.3 SQL Server 2000 Overview

SQL Server 2000 is the most popular relational database running on the Windows platform, currently holding 38% of the market share [19]. Its success may be due, in part, to its tight integration with the Win32 API [6]. Unlike database management systems that typically re-implement preexisting kernel services to run on platforms lacking such functionality (the least-common-denominator approach to development), Windows integration is a specific mandate followed by SQL Server developers [6]. The DBMS ships in seven flavors—Standard, Enterprise, Enterprise Evaluation, Personal, Developer, Windows CE, and Desktop—distinguishable by the maximum database size, number of processors, and total physical memory supported.

The design goal of SQL Server 2000 parallels that of Windows 2000—to scale gracefully from low-performance desktop machines all the way up to high-performance datacenter servers [6]. Towards this end, SQL Server 2000 manages its memory buffers in a dynamic, intelligent way that we discuss further in section 4.

Though SQL Server runs as a single process, it achieves high-volume symmetric multiprocessing by scheduling queued tasks efficiently using its User Mode Scheduler (UMS). Windows 2000 provides the underlying thread scheduling and asynchronous I/O completion notifications to make this possible. We further discuss file I/O in section 5 and process architecture in section 6.

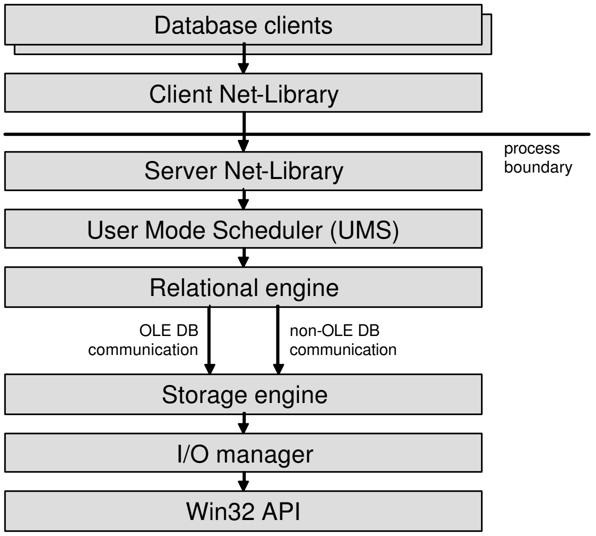

Figure 1.3.1

High level view of the SQL Server 2000 architecture (adapted from Inside SQL Server 2000, *figure 3-1* [6]*)*

SQL Server communicates with clients using a variety of interprocess communication (IPC) mechanisms, including popular networking protocols such as TCP/IP and IPX/SPX, named pipes, and shared memory. Both SQL Server and its clients use a *Net-Library* module to translate between protocol-dependent message formats and SQL Server's Tabular Data Stream (TDS) format. We discuss interprocess communication in depth in section 7.

Internally, SQL Server consists principally of a relational engine and a storage engine that communicate using an OLE DB interface and other mechanisms, as outlined in Figure 1.3.1. The relational engine performs higher-level functions such as command parsing and query optimizing, and the storage engine handles lower-level functions such as tuple access methods, buffering, locking, and transaction support. SQL Server ensures ACID transactions with its *serializable* transaction level, but it offers three less restrictive levels: *repeatable read*, *committed read*, and *uncommitted read*. Transaction support is based on write-ahead logging with fuzzy checkpointing.

To support transaction consistency, SQL Server protects data resources uses a two-phase locking algorithm at three levels of granularity—row, page, and table—dynamically escalates existing locks into coarser ones when it determines that doing so would increase overall efficiency. Objects are locked using ten locking modes, including the usual shared, exclusive, intention-shared, and intention-exclusive modes, plus some additional ones to efficiently support the bulk copy process (BCP) and other new functionality; deadlocks are resolved using a lock cycle detector thread. We discuss concurrency control in depth in section 8.

1.4 Paper Structure

Following a summary of related work (section 2) and a description of our suite of benchmarks (section 3), we analyze the same operating system services that Stonebraker discusses in his original paper: buffer pool management (section 4), file system management (section 5), scheduling and process management (section 6), interprocess communication (section 7), and consistency control (section 8). In each of these sections, we focus our analysis on the implementations of Windows 2000 and SQL Server, discussing the suitability of services provided by Windows 2000 for SQL Server and suggesting enhancements to both the Windows 2000 kernel and the Win32 API that would make these services more useful. We introduce some newly relevant services in section 9, and conclude in section 10.

2 Related Work

We reexamine Michael Stonebraker's 1981 determination that operating system services are inappropriate for database management systems [21]. Stonebraker's evaluation focuses on UNIX and INGRES, but many of his criticisms are generalizable to other platforms. We consider these criticisms to be as follows:

1. The buffer manager uses an inappropriate and unalterable replacement strategy—last recently used (LRU)—and does not have a selective force-out option to immediately commit pages to disk.
2. The file system does not efficiently provide for record-level access and wastes space by storing file control block trees, directory trees, and keyed access trees separately.
3. Task switching mechanisms are inefficient, and the scheduler may deschedule database processes while they are holding critical locks, instead of allowing database systems to run at a higher priority levels and granting them the power to deschedule themselves at the appropriate times.
4. Interprocess communication mechanisms are inefficient.
5. Native locking mechanisms and crash recovery support are insufficient.

2.1 Reassessments

Several teams of researches have attempted—or are currently attempting—to reevaluate Stonebraker's work in the context of modern-day commodity operating systems. Li Yang and Jin Li [23] examine Sun Solaris and Microsoft Windows NT in the areas of buffer pool management, file system support, process management and scheduling, and consistency control. They primarily discuss the details of the Solaris API, and only briefly acknowledge services provided by Windows NT. For both platforms, they fail to compare the investigated kernel services to comparable user-mode implementations or reveal whether or not these services meet the needs of modern-day database management systems.

Daniel Fellig and Olga Tikhonova [10] focus exclusively on the Windows NT 4.0 buffer management service and its use by SQL Server 7.0, the predecessor to SQL Server 2000. They claim to empirically investigate the Windows NT buffer management policy, but their testing procedure reveals that they are actually testing the replacement policy of the memory manager, not the replacement policy of the buffer manager. Additionally, their testing methodology is suspect for at least three reasons: 1) a background process runs simultaneously to dynamically change the amount of memory available, 2) they do not consider the effects of prefetching, and 3) key experimental variables—the number of pages to map and the number of pages to revisit—are fixed without explanation. This makes us question their conclusions that Stonebraker's complaints about buffer management remain relevant for Windows NT.

Josh Forman and Mark Luber [11] are concurrently completing work similar to that of our own, so their work is not yet available for review. Though they limit the scope of their study to buffer pool management, sequential file layout, and interprocess communication, they examine these services on both Windows 2000 and a Unix platform.

2.2 Operating System Criticisms & Designs

The insufficiency of commodity operating system services has been considered rather exhaustively, and overwhelmingly researchers concur with Stonebraker's assessment that operating system services are too opaque for some types of applications to utilize effectively. There are a wide variety of proposed solutions, which we loosely classify into the categories of "tweakable" kernels, extensible kernels, and exokernels. These solutions differ by the level of extensibility the operating system exposes to user applications, ranging from tweaks of existing services to the elimination of services beyond hardware abstraction.

2.2.1 Tweakable kernels

Margo Seltzer, Christopher Small, and Keith Smith [16] extend the BSD/OS 2.0 kernel so that applications may tweak the buffer read-ahead size, the lock granting policy, and the process scheduling order, among other things. Their results show that processes employing their interfaces can make better utilization of resources and run more efficiently. For example, the *gzip* compression utility runs 19% faster using the kernel's new interface to customize read-ahead size.

2.2.2 Extensible kernels

Extensible kernel systems are distinct from tweakable kernels in that they are designed from the ground up to support modularity and flexible configuration. The SPIN microkernel [3], for example, splits resource abstraction and resource management into separate domains, and allows processes to customize resource management decisions using *spindles*—system-specific privileged code that interfaces with the resource abstraction layer in kernel mode. The VINO microkernel [17] represents resources with objects that can be overridden dynamically by user processes in a technique called *grafting*. Grafted code runs in kernel mode and alters the behavior of a kernel service or adds pre- and/or post-processing to the service's default behavior. Other microkernels, such as Bridge and Scout, are also under development.

SPIN and VINO both require the machine code of their extensions to be produced using a special compiler that ensures safe behavior, such as memory accesses that only operate within acceptable ranges. This verification technique is essential for the stability of the operating systems, since poorly designed extensions, which execute in unprotected kernel mode, could potentially crash the entire system.

Though sometimes referred to as one [4], Windows is not a microkernel because its core services are not modular or extensible [18]. Drivers designed in accordance with the Windows Driver Model (WDM) can extend limited subsets of functionality, however; for example, file system drivers can interface with the buffer manager to slightly tweak the buffer replacement strategy. And, like extensible kernels, Windows uses a verification technique to ensure kernel-privileged code is stable. In contrast with the verification-by-compilation technique used by SPIN and VINO, however, Windows 2000 employs a verification-by-testing technique that requires "certified" drivers to be evaluated via rigorous testing conducted by humans. Those drivers which pass these tests and meet quality assurance requirements are marked with a special digital signature indicating compliance.

While SQL Server 2000 could have included a WDM driver that bypasses file system overhead and writes directly to raw partitions (like Oracle does), its designers chose not to do so. In fact, the server does not have any direct hooks into the kernel or extend the kernel in any way; it interfaces solely with the Win32 API.

2.2.3 Exokernels

Dawson Engler and M. Kaashoek argue that operating systems are fundamentally flawed because the resource abstractions they provide are unreliable, hard to change, inefficient, and inflexible—in essence, that abstractions cannot serve disparate application needs appropriately [7]. They advocate an abstraction-free kernel, called an *exokernel*. Aegis [9] is exokernel concept taken to the extreme, providing very limited functionality with a mere handful of system calls. It allows user-mode extensions for a layered approach to operating system design. The ExOS kernel [8], for example, is an exokernel written as an extension to Aegis. User applications can be designed to run directly on ExOS, or on a library operating system, such as one with a POSIX-compatible interface, that sits on top of ExOS. In this manner application are empowered to choose the appropriate level of granularity—Aegis, ExOS, or POSIX—for their efficient operation. The abstraction/efficiency tradeoff seen here is exactly the type of functionality for which Stonebraker was asking. We show that in certain areas Windows 2000 provides similar tradeoffs, but without the complexity of multiple OS interfaces.

3 The *Kub* Benchmark

Our own observations and conclusions derive from two principle sources: technical documentation and benchmark results. All of our benchmark applications belong to *Kub*, a kernel vs. user mode comparator benchmark suite. We wrote *Kub* from scratch specifically for this paper, even though other benchmarking suites such as *lmbench* and *kbench* are available. We consider these other suites unsuitable for our purposes since they are not designed to interface with the Win32 API optimally and do not reveal how much more efficient user-mode implementations of these services might be.

Kub solves these difficulties by interfacing directly with the Win32 API and providing alternative user-mode implementations of the kernel services it tests. These user-mode implementations are not designed to be full-featured solutions, but rather provide only the necessary, bare-bones support to reveal the minimum cost associated with implementing each service in user mode. Using *Kub* we can compare timing statistics to reveal the maximum relative gain obtainable by bypassing a kernel service and instead implementing it in user mode. This type of comparison is crucial to the analysis which follows.

All of the benchmarks in the *Kub* suite report kernel and user times in milliseconds and absolute run times in microseconds. The kernel and user times are determined by calls to the `GetProcessTimes` Win32 API function, and the microsecond measurements are tracked by the CPU's high-resolution timer and determined via calls the `QueryPerformanceCounter` API function. On the system we used for testing, the timer has a resolution of about 0.28 us.

The *Kub* benchmarking suite principally consists of six benchmarks:

1. ***KubBind.*** This benchmark measures the costs of performing all of the operations related to mapping views of pages into virtual memory, including binding and flushing (see section 4).
2. ***KubLocality***. This benchmark correlates the degree of spatial locality among file fragments with sequential read time (see section 5).
3. ***KubLock***. This benchmark compares the efficiency of the file system locking mechanism to a user-level locking mechanism we designed ourselves (see section 8).
4. ***KubMessage***. This benchmark measures the costs of using several IPC technologies to transport messages of varying size (see section 7).
5. ***KubReplaceStrategy***. This benchmark examines the effectiveness of the buffer manager's replacement strategy (see section 4).

6. ***KubSchedule***. This benchmark compares thread scheduling to user-mode fiber scheduling to suggest the most efficient process architecture (see section 6).

We compiled these programs using version 6.0 of the Microsoft C++ compiler and executed them on an 800 MHz Intel Pentium III machine with 256 MB of RAM running Microsoft Windows 2000 Server (build 2195). We discuss the details of these programs, and their output, in subsequent sections. Source code for the entire suite is available online.[2]

All of the benchmarks' threads run at a high-priority level, which means they are always be scheduled when they have work to do, preempting all other system and application threads in the *normal* process priority class or lower. Only a select few system processes run at a higher priority class, so the impact of other processes on the benchmark timings is assured to be minimal. The *null* benchmark, which simply starts and stops the timer, reports user and kernel times of 0 ms and total execution time as 3-4 us. Since these numbers are negligible, we do not adjust the times we report to account for the costs associated with starting and stopping the timer. Because the kernel and user times are supplied at such a high level of granularity, we avoid describing them in our results. Additionally, we avoid short-running benchmarks so that the total execution time is always at least several orders of magnitude greater than 5 us. In order to ensure that our benchmarks run for an adequate duration, we often execute a task many times; for example, in *KubLock* we measure the time it takes to acquire and release not a single exclusive lock, but thousands of such locks in sequence.

We now consider the five areas of operating system support for database management with the help of the *Kub* benchmark suite.

4 Buffer Pool Management

Stonebraker has wide-ranging concerns regarding operating system-provided buffer management services. He claims that the overhead of memory-mapped files is too high, that the latency of buffer accesses is too great, and that the buffer manager's replacement and prefetching strategies are inappropriate and not customizable by applications. We consider how far Windows 2000 has come from this sorry state of affairs.

4.1 Memory-Mapped Files

We use the *KubBind* benchmark to evaluate the efficiency of the Windows 2000 file mapping implementation and compare it to the traditional method of file I/O. The file mapping technique utilizes the functionality of the memory manager to map sections of a file to an application's virtual memory as it needs them, and write these pages back to disk when they are dirtied. The memory manager takes care of performing all file I/O to make this possible. Some of the intricacies of memory mapped files are discussed in detail in the context of concurrency control (see section 8.1).

The *KubBind* benchmark determines the latencies of memory-mapped operations and compares these latencies to the analogous operations performed by traditional (unmapped) file I/O. The latencies of memory-mapped operations are discovered by timing specific system calls involved in the memory mapping process. The results are shown in Figure 4.1.1.

The *open file* time corresponds to latency of the `CreateFile` system call, which obtains a handle to the file *KubBind* wishes to map. We vary the file size from 64 KB to 64 MB. The *create map* time corresponds to latency of the `CreateFileMapping` system call, which provides a handle to a read-write virtual memory map of the entire file. The function has arguments that allow for the creation of smaller maps, but we are interested in mapping the entire file at once to determine how the function's performance changes as file size increases. The *bind* time corre-

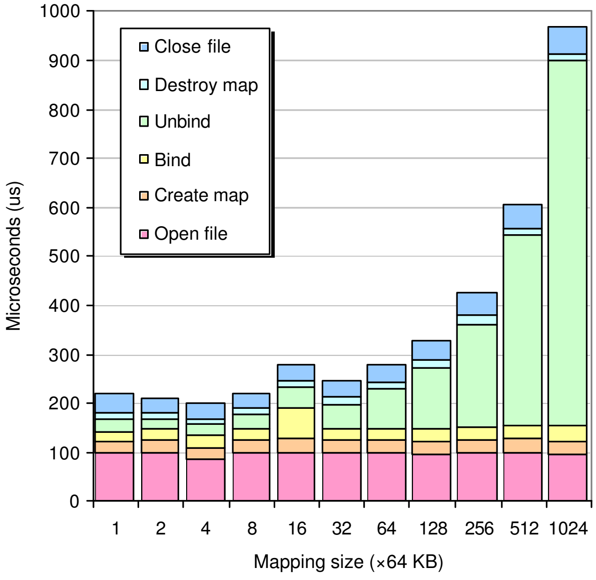

Figure 4.1.1

Costs associated with mapping views of a file of various sizes, ranging from 64 KB to 64 MB. All of the pages are dirtied and flushed before being unbound (source: KubBind *benchmark)*

[2] The *Kub* codebase and the benchmark results we cite in this paper are available in ZIP format on the web: *http://www.fas.harvard.edu/~vasil/kub*

sponds to the latency of the `MapViewOfFile` system call, which provides a pointer to this mapped view. Since the pointer indicates the starting address of a contiguous buffer in virtual memory that maps to the file's contents, the buffer size matches the file size. The *unbind* time corresponds to the latency of the `UnmapViewOfFile` system call, which unmaps all pages in the buffer's range. Between the mapping and unmapping calls, *KubBind* modifies all pages in the map by incrementing every byte in the buffer range, and then forces a flush of all modified pages to disk. (We evaluate read/write and flush times later in this section.) Note that *KubBind* opens all files with the `FILE_FLAG_WRITE_THROUGH` flag, which guarantees that flush functions do not return until all dirty data has made its way to disk. The *destroy map* time corresponds to the latency of the `CloseHandle` system call, which destroys the system's internal memory-mapped objects, and the *close file* time corresponds to another `CloseHandle` system call, which closes the mapped file.

As we see from Figure 4.1.1, all of the memory-map operations are performed in constant time, except for *unbind*. The latency of unbind is a function of the number of pages involved in the memory map, which makes sense since memory associated with each one of these pages must be deallocated and the system's page tables updated. The *bind* time does not show the same type of latency because the system does not adjust the page tables when the map is created; instead, it uses a lazy evaluation technique, waiting for page faults before it does any substantial work. The bind operation simply notes mapping request in a virtual address descriptor (VAD) tree by inserting a node into the tree indicating the desired mapping range. When a page fault occurs for a given address, the memory manager uses this tree to determine whether or not a given page should be mapped to a file, and, if so, it loads the appropriate data from the file into physical memory and then modifies the page tables accordingly. Consequently, the cost of the memory mapping is not reflected in Figure 4.1.1; the cost is accrued by the first read or write access to a bound page.

The much higher costs of reads, writes, and flushes are illustrated in Figure 4.1.2. Here we compare costs of file I/O via memory-mapped files to that of traditional (unmapped) files. For memory-mapped files, *initial read/write* and *subsequent read* measure the time it takes *KubBind* to increment every byte on every page of the mapped pages. The initial read/write always takes longer than the subsequent one, even though the same code is executed in both cases, because, as we just noted, the first read prompts the operating system to load the request page into memory and alter its page tables accordingly. The *flush* value measures the cost of the `FlushViewOfFile` system call, where all dirty pages (the entire file) are written to disk. *Total time* sums all of the costs involved in the file mapping (from opening the file and creating the map, to modifying all pages, to flushing these pages, to closing the map and file).

For unmapped files, *initial read/write* refers to reading the entire file into memory and writing this buffer back to disk in 64 KB chunks using the `ReadFile` and `WriteFile` system calls. *Subsequent read* refers to incrementing each byte of this memory, as in the mapped file case. *Flush* measures the cost of the `FlushFileBuffers` system call, where all modified pages of the file that remain in the system's buffer pool are written to disk. *Total time* sums all the costs involved in the process (from opening the file to

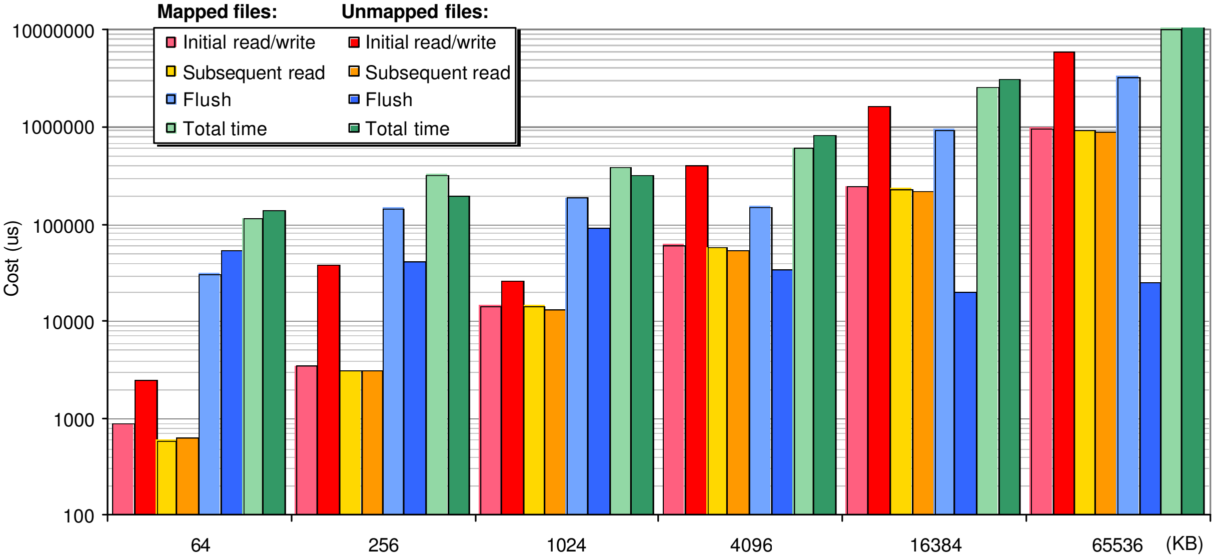

Figure 4.1.2

Comparison of costs performing file I/O functions with memory-mapped files and traditional (unmapped) files (source: KubBind *benchmark)* modifying the entire file to flushing modifications in the buffer pool to closing the file).

The two file I/O techniques appear to have some tradeoffs. While the *initial read/write* time for memory-mapped files is smaller, the *flush* time for unmapped files is smaller. We attribute these differences to the distinct caching policies used by the memory manager (in the mapped file cases) and the buffer pool manager (in the unmapped file cases). The buffer pool manager is more eager to flush dirty pages to disk than the memory manager, and begins doing so while the initial read/write is proceeding; thus, the *initial read/write* times of file I/O reflect this cost. The memory manager, on the other hand, has a *modified page writer* thread that writes dirty pages every five minutes or as the number of modified pages exceeds a dynamically calculated threshold value [18], so hardly any page flushing happens at the *initial read/write* stage.

Since the background activity of the memory manager and the buffer pool manager make policy decisions and perform flushes at various stages of *KubBind's* execution, the only truly reliable metric is *total time*, which reflects the total latency of the entire process. As the green bars illustrate in Figure 1.4.2, the difference in cost between mapped file I/O and unmapped file I/O is negligible. Stonebraker's worries over the expense of mapping files into virtual memory space seems to be unfounded. Modern hardware and Windows 2000 are able to efficiently manage virtual memory.

The decision concerning when to use a particular method should depend in part on which method is more convenient for a specific task, and how much control is desired over when the actual disk reads and writes occur. The unmapped technique seems to offer more control, since `ReadFile` fetches data immediately and `WriteFile` writes data in short order. The memory-mapped technique uses lazy reading—by waiting for the first access to a page—and writes dirty pages at a much less frequent interval than the buffer pool manager.

4.2 Buffer Management Strategies

The buffer manager exposes four types of buffer management directives that processes can choose from to indicate their preferred the prefetching and replacement strategies: [18]

- **Normal strategy**. This strategy tells the buffer manager to asynchronously read ahead up to 64 KB of data by extrapolating the last three reads. Once referenced, pages are unmapped and placed at the end of the memory manager's standby or modified list, depending on whether or not they are dirty.
- **Sequential strategy**. This works like the normal strategy, but tells the buffer manager to read ahead three times as much data. Once referenced, pages are unmapped and placed at the front of the standby or modified list.
- **Random strategy**. This strategy prevents read-aheads and minimizes page unmappings.
- **No caching**. This strategy tells the buffer manager not to prefetch or cache any pages.

We use the *KubReplaceStrategy* benchmark to determine the effectiveness of these different strategies by performing file I/O both sequentially and randomly on 64 KB chunks of data in a 1 MB file. The benchmark iterates over four variables: the prefetching/replacement strategy (sequential, random, none), the type of I/O (read or read/write), the access pattern of I/O (sequential or random), and the file I/O technique (unmapped or mapped). For each of the 24 tests, *KubReplaceStrategy* first creates a fresh 1 MB file, copies it, deletes it, and then performs the I/O operations on the copy of the file. This guarantees that the buffer pool does not contain any cached data of the file in memory, which would skew results.

When the type of I/O is simply *read*, 16 reads of 64 KB each are performed on the file. With a sequential access pattern, these 64 KB chunks are read in sequence from file offset zero upward. With a random access pattern, these 64 KB chunks are selected at random 64 KB-offsets. All of the random access pattern tests use identical offsets, which are randomly selected at the start of benchmark using the C runtime library's `rand` function.

When the type of I/O is read/write, each read occurs in the manner previously described. The subsequent write first increments every byte in the 64 KB buffer and then writes this buffer to disk. When the memory-mapped I/O technique is used, touching each byte ensures that all 16 pages of the 64 KB view are dirtied. At the end of the write process, `FlushFileBuffers` (for unmapped I/O) and `FlushViewOfFile` (for mapped I/O) are called to ensure that the entire 64 KB chunk of data is sent to disk immediately.

The results of the benchmark appear in Figure 4.2.1. Concerning unmapped files, employing either prefetching technique helps reduce read latency. As expected, the sequential prefetching technique does especially well as lowering the latencies of sequential reads. When writes are added to the mix, however, the benefits of prefetching are dimmed by the high latencies of buffer copies and disk I/O.

The mapped files do not benefit from the prefetching techniques chosen by *KubReplaceStrategy*, apparently because the memory manager does not ask the buffer pool manager for hints when loading pages. We do see that the clustering technique employed by the memory manager to take advantage of spatial locality (by faulting nearby pages) does provide some benefit to memory-mapped files when the reads are sequential. We also see that reads to mapped files are about an order of magnitude faster than reads to unmapped files; this difference results from the latency of the buffer copies needed to transfer data from the buffer pool to the process' own memory in the unmapped file case. For mapped files, no buffer transfer is necessary. Note that for write operations the expense of these extra copies is negligible due to the much larger expense of disk I/O.

In summary we see that prefetching by the buffer manager can reduce read latencies for unmapped files, although such benefits are minimal when there is a high volume of writes. Mapped files do not benefit from the services of the buffer manager, but reads to mapped files are faster because there is no need for a buffer copy.

4.3 Support for Database Management

The *KubBind* benchmark addresses Stonebraker's concern about the overhead of memory-mapped files; it shows that memory-mapped files offer no real disadvantage over traditional unmapped file I/O: the overall latencies are the same. The potentially substantial bookkeeping task of identifying pages currenty mapped in memory does, however, remain a burden to the DBMS. The *KubReplaceStrategy* benchmark shows how choosing among a limited number of prefetching strategies may increase performance, but there remains no way for a DBMS to offer its own prefetching advice, though certainly it would know more about the data to prefetch that the buffer manager could possibly surmise from access history alone. Overall, we have seen some improvement in efficiency, but very little improvement in flexibility. Successful database systems likely need to forego the file mapping technique so that they have more precise control over when pages are actually read and written, and they need to bypass the buffer manager in order to fine-tune page prefetching and replacement.

For database data, SQL Server does not take advantage of memory-mapped file functionality or the buffer pool. This may seem surprising, since the *KubReplaceStrategy* suggests that the alternative, unmapped file I/O, would be more expensive than using mapped I/O. This is true, but there is yet another alternative for file I/O: scatter/gather I/O. This technique allows the file system to access process buffers directly without the overhead of a single memory copy. Even better, a single read or write request can involve regions of discontinuous memory (hence the name "scatter/gather"). The only restrictions are that each buffer is aligned on a page boundary (easily accomplished by allocating memory with the `VirtualAlloc` function) and that the write sizes are multiples of the sector size. The technique actually performs better than simple mapped files because it bypasses the memory manager. It also gives processes precise control over when buffers are read from disk or written to disk, unlike memory-mapped pages.

Since disk latency remains on the critical path of efficient query processing, SQL Server implements its own prefetching and replacement strategy on top of scatter/gather I/O to obtain maximum disk performance. SQL Server performs different types of read-ahead for tables and indexes. It also includes various optimizations prevent buffer thrashing as a result of merry-go-round scans. Finally, it self-tunes the amount of physical memory it utilizes to keep its

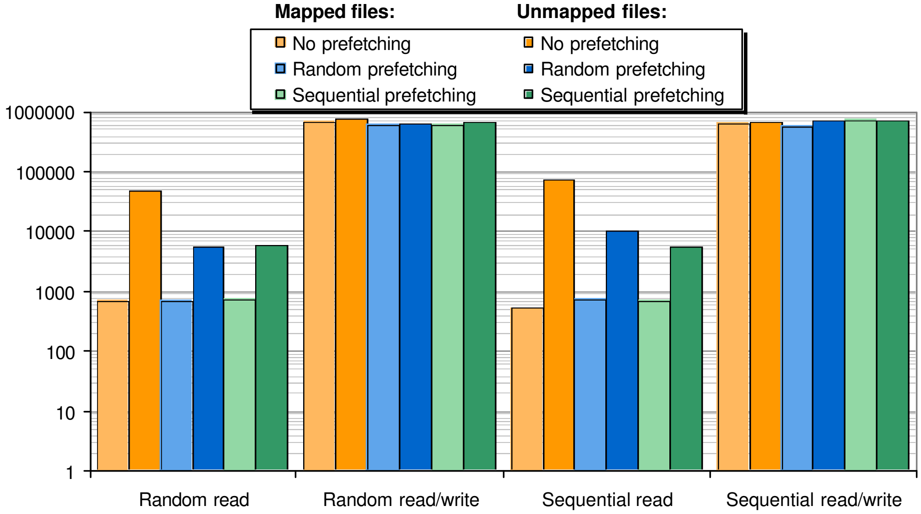

Figure 4.2.1

Cost of traditional and mapped file I/O in tandem with various buffer prefetching techniques. Chunks of 64 KB are read and written both sequentially and randomly from and to a 1 MB file.
(source: KubReplaceStrategy *benchmark)*

buffer as large as possible while not hindering system performance [6].

We find SQL Server's techniques to managing its own buffer pool remain painfully redundant with those of the memory manager and buffer pool manager. Unfortunately, in the current implementation of Windows 2000, scatter/gather I/O is incompatible with both managers. In fact, any file used for scatter/gather I/O must have caching explicitly disabled. Ideally, scatter/gather functions would be able to operate on mapped views of files so that SQL Server and similar applications need not duplicate the fine-tuned mechanisms of the memory manager. This would also require that the memory manager give applications greater flexibility concerning when dirty pages are written to disk, something we address in section 8.1.

5 File System Management

Stonebraker makes several complaints about the functionality of file systems. The UNIX file system he reviews did not provide extent-based allocations (for spatial locality), atomic record I/O, or keyed access trees, and it wasted space by maintaining separate trees for organizing directories and file fragments. Considering these drawbacks, we understand why databases sometimes bypass file systems entirely and perform I/O directly on raw disk partitions. System R did so back in 1976, and Oracle recommends doing so today.

After discussing the tradeoff between using the raw partitions versus the native file systems supported by Windows 2000, we evaluate how well these file systems alleviate Stonebraker's criticisms. For completeness, we add two important file system-related topics not considered by Stonebraker: zero-fill guarantees and advanced I/O techniques.

5.1 File Systems vs. Raw Partitions

Windows 2000 includes native support for several types of file systems, including FAT16, FAT32 and NTFS. These file systems vary by the maximum partition size they support, the robustness and efficiency of their recoverability scheme, and value-add services, such as data encryption.

FAT32 uses a lazy-write recoverability scheme. It performs I/O using an intelligently managed cache that it flushes modifications to disk in an optimized way, so as to minimize actual disk activity. The drawback to this approach is that there are times when the physical disk is in an inconsistent state; thus system crashes or power failures necessitate the need for a disk checker utility that repairs such inconsistencies. Sometimes, however, these inconsistencies are not recoverable, and user data is permanently lost.

NTFS uses a write-ahead logging scheme similar to that employed by databases. NTFS flushes to disk log entries of metadata operations before actually performing them; thus it can roll back to a consistent state interrupted metadata operations, such as file extends and directory structure modifications, by reading the log entries for unfinished metadata operations in reverse. As a value-add, NTFS also includes support for security descriptors, compression, and encryption.

While the NTFS file system makes better assurances about recovery than FAT, neither system makes any guarantee about the recoverability of user data. The designers of these file systems chose not to do so as an intentional trade-off between recoverability and performance [18]. Neither file system exposes any transactional support primitives to assist applications in making this data recoverable, either.

At a superficial level, these file systems do not seem to offer much more support to a database system than that offered by a raw partition. A database system writing directly to a raw partition, which is not managed by any file system, would not lose or gain any degree of recoverability, but would certainly gain in efficiency, since the overhead of maintaining the file system has been eliminated. There are drawbacks, however. Database administrators must manage database data, and they are accustomed to doing so using traditional file utilities and operations (move, copy, etc.), which would be unavailable if database data were to reside on a raw partition (since these things require a file system driver).

As it turns out, the gains in efficiency resulting from bypassing FAT or NTFS are negligible. Even when the I/O system is fully throttled, the performance difference between running SQL Server 2000 on a file system versus a raw partition is around 2% [6]. Therefore, on the Windows platform, the administrative costs of managing the raw partition outweigh their benefits. This explains why Microsoft recommends that its SQL Server customers store their database files on FAT or NTFS partitions, and not on raw partitions. Given this preference toward file systems, we examine the relevance of Stonebraker's criticisms concerning these systems.

5.2 Spatial Locality

Extent-based allocations are important to a DBMS since disk arm movement is a very expensive part of disk I/O. By clustering related data on physically close sectors, a database system can cut down on read time.

SQL Server 2000 supports both FAT and NTFS file systems, though neither offers extent-based file allocation to promote spatial locality. Windows 2000 does offer defragmentation I/O control codes for use with the `DeviceIoControl` function that allow processes to determine the actual disk sectors allocated to a file (`FSCTL_GET_RETRIEVAL_POINTERS`) and defragment files as necessary (`FSCTL_MOVE_FILE`), but these control codes are low-level and intended for defragmentation utilities that run occasionally, not high-performance servers. Further, they do not work with earlier versions of Windows or with NTFS when cluster sizes are greater than 4 KB, and they don't guarantee that files are placed on physically proximal clusters, since two logically numbered clusters may in fact be on opposite parts of a disk.

To put the locality problem in perspective, we use the *KubLocality* benchmark to estimate how impaired read time becomes as the spatial locality of file fragments deteriorates. *KubLocality* creates a 640 KB data file in 4 KB chunks. In between each 4 KB write, it appends some data into a separate "delocalizing" file on the same disk (separate runs of the benchmark vary the size of this data from 4 KB to 4 MB). Since FAT and NTFS do not support extents, they interleave the data of these two files and thus provide them both with poor spatial locality. We verified this behavior by using the `DeviceIoControl` function to determine exactly how many fragments into which our test file was divided, and the number of fragments did in fact reveal interleaving. The NTFS disk we tested had an allocation size of 4 KB, while the FAT32 disk had an allocation size of 32 KB. For NTFS, our 640 KB file was reported as occupying 160 distinct fragments (and 640 ÷ 160 = 4 KB). For FAT32, our 640 KB file was reported as occupying 20 distinct fragments (and 640 ÷ 20 = 32 KB). The space between each fragment matched the amount of interleaved data we wrote to the de-localizing file in between each write to the data file.

In turn, *KubLocality* creates fragmented 640 KB data files on both NTFS and FAT32. One such file has 4 KB interleaving chunks between each allocation unit, another has 8 KB, and so on, up to 4 MB. *KubLocality* reports sequential scan latencies for all of these files, and the results are shown in Figure 5.2.1 (in the "single delocalizing file" category). The data is presented using a bubble graph because a third dimension of data appeared during our testing: the number of fragments. Unexpectedly, the number of fragments our data file occupies decreases under FAT32 as the interleaved chunks exceed 64 KB. Thus, the third dimension of data in Figure 5.2.1, the size of the bubbles, represents the number of fragments into which the data file is divided. To be specific, the number of fragments drops to 16 with 128 KB interleaved chunks, to 9 with 256 KB chunks, 5 with 512 KB chunks, and 4 with 1024 KB chunks. NTFS always remains consistent with 160 fragments, irrespective of interleave size. We figure this behavior may be due to some intelligence within the FAT32 file system driver that the NTFS driver lacks: the ability to identify large files and reserve extra space for their growth—in other words, that it has some primitive ability to support extents. Recall that in *KubLocality*, we use a single "de-localizing" file to interleave with our data file. As this delocalizing file grows large (it becomes 640 MB for the 4 MB interleaving test), perhaps FAT32 dedicates some extra space for extents, allowing, as a side effect, for our data file to have better spatial locality. To test this, we ran *KubLocality* again, but had it create separate delocalizing files in between each 4 KB write to the data file. In this case, FAT32 in fact remained consistent in distributing the data file into 20 fragments, even at the higher interleave sizes. This suggests that FAT32 may indeed have some extent-like logic, but since we ran our tests on a well-used (though mostly unfragmented) file system, we do not have the control conditions necessary to be confident in such a theory. The behavior is reproducible on our test system but may, in fact, be coincidental.

Regardless of whether the FAT32 behavior is intentional or not, we certainly see from the results of the *KubLocality* benchmark that spatial locality is crucial to efficient file I/O. On the disks we tested, once the space between fragments extend beyond 100

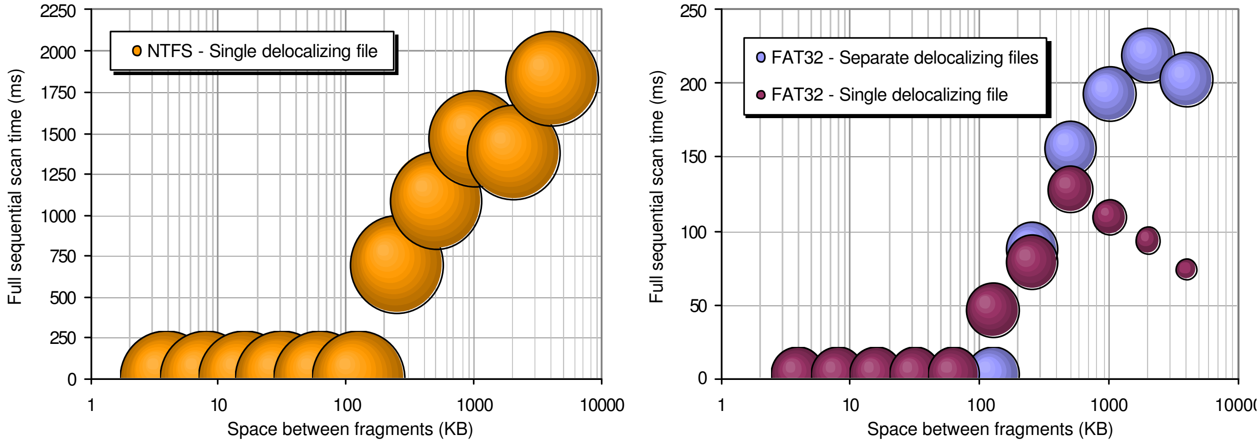

Figure 5.2.1

Cost of sequentially reading a 640 KB file with varying space between fragments; the size of the circles show the relative number of fragments into which the file is divided (source: KubLocality *benchmark)*

KB, we see that the seek time increases dramatically; sequential scan times become up to 800 times longer in our tests.

Windows 2000 file systems need to provide some mechanisms for extent-based allocations. Since databases typically grow files in large chunks, anywhere from 64 KB to several megabytes, file systems should make a best-effort attempt to give these large allocations good spatial locality. As we have seen from the *KubLocality* benchmark, any simultaneous file appends by other process may seriously disrupt spatial locality otherwise, and in the multithreaded Windows environment, this is a very likely scenario. The extent functionality Windows should provide need not be explicitly exposed through an API—the file system could perform such a task automatically, as FAT32 seemed to attempt in our benchmark runs.

5.3 Record-Level Primitives & Triple Indexing

The character array-based NTFS and FAT file systems may not adequately meet the needs of some important data-centric applications such as web servers, e-mail clients, personal information managers, and, of course, databases. These applications tend to store and traverse large amounts of homogenous data, suggesting natural proclivities to a record-base file system. A file system that provided record-level primitives guaranteeing read/write atomicity and natively supporting indexing structures would seem highly relevant and useful. The IBM VSAM file system and the Compaq RMS file system are two examples of record-based file systems that do offer primitive record-based file I/O, but neither guarantees atomicity for writes. There has been some interesting research into offering such guarantees [12], but to our knowledge no commercially viable system is yet available.

If Windows were to adopt a record-based file system, some of the aforementioned applications would benefit. We argue that SQL Server likely would not. There are a host of integration issues making the use of a record-base file system complex and perhaps inefficient. For example, record-based file systems often include mechanisms for keyed access. Such functionality could not match the robustness of SQL Server's own keyed access data structures, which includes a tightly integrated two-phase locking protocol and intricate prefetch and caching routines. In essence, for the record-level file system to be advantageous to the database system, it would have to incorporate much of the database system's functionality or provide sufficiently flexible interfaces to permit the database system this functionality. The benefit of all this complex integration seems minimal.

Stonebraker hoped that file, data, and keyed access trees could all be consolidated into a single tree structure. His hope was that the integration would reduce overhead. Yet to date the layered approach to technology (e.g. OS buffers on top of hardware caches, client-middleware-server architectures, business/presentation layers), has proved to be relatively successful. While disk is now cheap enough so that the overhead of maintaining separate trees is minimal, we also see that for SQL Server the overhead of the file system in its entirely is, in the worst case, a minimal 2%.

5.4 Zero-Filled Pages

One point neglected by Stonebraker but certainly important for database system designers is guarantees the operating system may or may not make about the state of freshly allocated data. Ideally, designers like to see newly allocated memory or disk sectors zero-filled, since this represents a valid unallocated state, as opposed to undefined garbage bits which a DBMS would be burdened with initializing.

Windows 2000 guarantees that all pages allocated in memory using the `VirtualAlloc` function are zero-filled, but it does not make any guarantee about the state of file extents made using the `SetEndOfFile` function. These extents may or may not be zero-filled, so the database designer must manually initialize them. If the database wishes to grow a data file by even a modest value, say 64 KB, it must zero-fill the entire extent itself; and this can delay other processing.

Though really nothing more than a minor nuisance, the lack of support by the operating system to assist in the zero-filling task is discouraging. A possible workaround might involve combining the zero-filling capability of the memory manager with the functionality of the file system driver by using memory mapped files. A call to `CreateFileMapping` that specifies as its boundary the current file size plus the extent size desired will, in fact, grow the file and zero-fill it, since the memory manager zero-fills new pages and these pages will get flushed to the new extent. Unfortunately, memory-mapped files are not a viable method of file I/O for database systems, as we discuss in section 8.1.

5.5 Advanced I/O Techniques

SQL Server 2000 takes advantage of two new file I/O features in Windows 2000: asynchronous I/O and scatter/gather I/O. Asynchronous I/O allows SQL Server to better utilize the CPU while the disk processes I/O requests. Scatter/gather I/O, as we discuss in section 4, allows reads and writes of contiguous logical storage to be based on discontinuous allocations of memory, which makes SQL Server's job of maintaining its memory easier and more efficient.

6 Scheduling & Process Management

Without operating system support for threads, database designers typically contemplate a process-per-user approach or a server approach to DBMS design. Ever-negative Stonebraker considers the drawback to both approaches [21]:

- **The process-per-user approach** creates a new DBMS process for each database client. Though these processes share data segments such as the lock table and the buffer pool, it requires an excessive amount of memory to maintain multiple process states, suffers from the potentially high expense of task switches, and requires mechanisms to synchronize inter-process memory sharing.
- **The single-server approach** uses a single DBMS process to handle all client transactions. The process must use IPC to communicate with database clients, and it must provide its own scheduling mechanisms to handle client requests. The approach is akin to re-implementing the kernel's process scheduler in user mode.

The process-per-user approach is unacceptable for any large-volume DBMS which may have thousands of concurrent users. The overhead needed to maintain each process is simply too great. Most database system designers take the server approach and implement their own scheduling package.

Windows 2000 natively supports threads, which makes the single-server approach much easier to implement. Additionally, the Win32 API supports fibers, which are like light-weight threads; each fiber has its own call stack but executes within the context of a specific thread. Fibers allow programs to utilize less memory, avoid the cost of some context switches, and take greater control over their own scheduling.

Database designers thus have four viable process management options on Windows 2000:

1. **The thread-per-user approach**. This is similar to the process-per-user approach in that the DBMS trades the costs of process switches and overhead for that of thread switches and overhead. Though thread contexts require less memory than that of processes, their overhead is still relevant. Since Windows 2000 schedules threads, not processes, the latency of a thread switch is comparable to that of a process switch. The benefit of thread-per-user over process-per-user is that all of the threads share the same virtual address space, making it easier to share common data structures.
2. **The fiber-per-user approach**. This is similar to the single-server approach in that fibers are a user-mode construct provided above the kernel, and that processes are responsible for the scheduling of their own fibers. The benefit of this approach over the single-server approach is that the Win32 library automatically maintains a distinct stack for each fiber, which simplifies any user-implemented scheduling code.
3. **The thread pooling approach**. This concept adapts the thread-per-user approach by placing a limit on the number of threads a DBMS server may create. Once the limit is reached, additional clients must wait in a queue for an available worker thread before being processed. In asynchronous thread pooling implementations, threads that would normally block on system calls such as disk I/O can instead use asynchronous I/O and service clients in the queue while these system calls complete.
4. **The fiber pool approach.** In this approach, a thread is created for each CPU and a fiber pool handles user requests. The DBMS schedules fibers to run on available threads and deschedules them as they wait for asynchronous operations to complete. Of the four approaches, this approach makes maximal use of system resources with minimal overhead.

6.1 Cost Analysis

We use the *KubSchedule* benchmark to analyze the overhead of threads and fibers to determine whether or not the performance gain from using fibers is enough to motivate database systems to perform their own scheduling. The benchmark evaluates four scheduling techniques:

1. **Thread-switching overhead**. Two threads are created with the `CreateThread` function; these threads share two event objects. An event object is created with the `CreateEvent` function, and allows a thread to block until another thread sets the event using the blocking `WaitForSingleObject` function. Each thread in *KubSchedule* sets the other thread's event and then blocks to wait for its own event to be signaled a given number of times. *KubSchedule* measures the costs associated with this many thread context switches. Each context switch is triggered by the blocking call almost as soon as a thread is scheduled—much sooner than when the quantum assigned to the thread would expire (typically between 10 and 15 ms on Intel's x86 systems [18]).
2. **Fiber-switching overhead**. Fibers must be scheduled on top of existing threads, and a fiber can be created only by another fiber. To get the process started, a single thread uses a `ConvertThreadToFiber` call to initialize Win32 memory structures to support fibers and convert itself to a fiber. Subsequently each fiber calls `SwitchToFiber` to switch fiber contexts a given number of times. *KubSchedule* measures the costs associated with this many fiber context switches. Note that these switches happen in the context of a

single thread, so that they do not involve the overhead of the Windows 2000 scheduler.

3. **DLL procedure call overhead**. This is not a scheduling technique per se, but is useful to get a relative sense for how expensive a context switch is compared to a DLL procedure call. All Win32 function calls, at minimum, involve the overhead of a DLL procedure call. *KubSchedule* repeatedly calls a DLL procedure that simply returns.
4. **Procedure call overhead**. To compare the overhead of DLL calls to intra-process procedure calls, *KubSchedule* also repeatedly calls an intra-process procedure that simply returns.

The *KubSchedule* benchmark measures the amount of time it takes to perform between 100 and 100,000 scheduling switches (be they thread switches, fiber switches, or procedure calls). The results of *KubSchedule*, shown in Figure 6.1.1, demonstrate that thread context switching is an order of magnitude less efficient than fiber context switching, and that fiber scheduling is an order of magnitude less efficient than procedure calls. Do note, however, that our fiber context switching code was devoid of any user-mode scheduling algorithm whereas the thread context switching made use of the Windows 2000 scheduler. In actual applications, the user mode scheduler necessary in a successful fiber-based implementation may put the two technologies more in line with each other with respect to latency, although fibers retain the benefit of a smaller memory footprint.

Note that Stonebraker's original complaint about the inefficiency of context switches was that they cost over 1,000 instructions on UNIX while efficient systems had reduced the expense to about 100 instructions [21]. Now compare this order-of-magnitude difference between the costs associated with thread and fiber scheduling. It appears that Windows has provided database systems with a more efficient, less abstract interface in which to perform scheduling with the order-of-magnitude reduction Stonebraker requests.

While the costs of the scheduling mechanisms generally grow linearly, the lower three lines seem to bend a bit between 1,000 and 10,000 switches, most likely because of interruptions by the Windows scheduler to see if other threads are available to run. Note that the Windows 2000 scheduler queues waiting-to-run threads in linked-list structures, with one such structure for each priority level, so that it can make scheduling decisions in constant time, irrespective of the number of threads in the system. When we modify *KubSchedule* to create up to 2,000 background threads at normal priority before beginning its usual tests (recall that benchmark tests always run at a high priority level), we find no differences in the results. Thus, performance of the system scheduler does not change as the number of threads increases.

As an interesting aside, we note that the cost associated with a DLL call matches that of a standard procedure call, indicating that calls to Win32 functions are no more expensive than calls to intra-process functions. We attribute this to Windows' native support for dynamic link libraries.

6.2 Support for Database Management

Databases want symmetric multiprocessing to 1) handle background operations such as checkpointing, lazywriting, log writing, and cleanup tasks, and 2) to process concurrent client requests.

The first need is easily satiable by giving each of these background operations its own thread, assigning relative priority levels to these threads, and letting Windows 2000 perform thread scheduling using its own highly-tuned algorithms. In fact, this is exactly what SQL Server does.

The second need cannot be solved using the same technique, since a thread-per-client approach requires too much overhead. Instead, SQL Server uses thread pooling to dynamically create between 16 and 100 worker threads (as usage demands dictate) [6]. SQL Server provides its own User Mode Scheduler (UMS) to handle dispatching client requests to threads.

As an optimization, SQL Server exposes a "lightweight pooling" option that permits database administrators toggle the server between thread and fiber

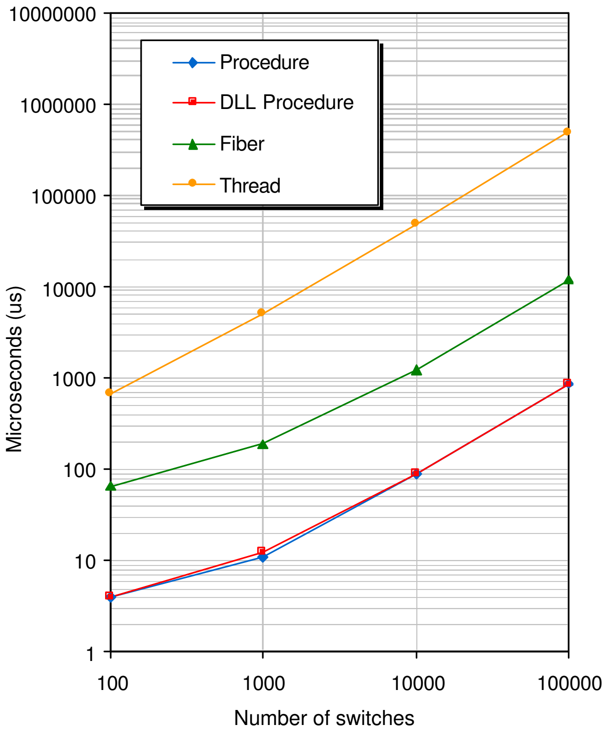

Figure 6.1.1

Scheduling technique cost vs. number of switches (source: KubSchedule *benchmark)* scheduling mode. The fiber mode creates no more worker threads than CPUs available, and has the UMS distribute work to fibers that run on top of these threads. Using fibers on a multiprocessor system with high CPU utilization improves performance by around 5% [20].

While SQL Server does make extensive use of the Windows 2000 threading model and I/O completion ports to handle a high volume of concurrent connections without the need for a transaction manager [6], it must still provide its own UMS to take advantage of thread pooling. This doesn't represent a mismatch between kernel services and user needs, but rather a gaping hole in the functionality of kernel services. A kernel-level thread pooling service would likely benefit a broad range of server applications, including web servers and Windows services, which must each implement its own UMS to efficiently handle a high volume of concurrent users.

7 Interprocess Communication

Stonebraker considers the expense of interprocess communication (IPC) to be a major roadblock in the efficient implementation of the single-server DBMS architecture SQL Server employs [21]. IPC costs have been reduced dramatically in recent years [2], but not enough for these costs to be disregarded, especially for client-server designs where performance depends heavily on IPC costs [13].

We evaluate various Windows 2000 IPC mechanisms to determine whether the associated costs should concern SQL Server 2000 developers, as it did Stonebraker. Windows 2000 offers many IPC techniques, including:

- **Component object model** (COM) interfaces, which provides object linking and embedding (OLE) support in compound documents,
- **Dynamic data exchange** (DDE), which is based on Windows clipboard technology and message passing via the Win32 message pump,
- **Mailslots**, which use file I/O functions for unreliable unidirectional data transfer,
- **Named and unnamed pipes**, which use file I/O functions for reliable bidirectional data transfer,
- **Remote procedure calls** (RPC), which rely on a Microsoft Interface Definition Language (MIDL) compiler to generate proxy functions that package and transmit function arguments when RPC functions are called,
- **Shared memory**, which is available only through virtual memory file mapping and requires processes to provide their own synchronization mechanisms, and
- **Windows sockets**, which provides TCP/IP and UDP/IP network transport services.

With the exception of the shared memory technique, all of the methods listed above support IPC over a network. Not all of them are designed with efficiency in mind, however. COM and DDE technologies in particular involve a high overhead and are not suitable for communication between a client and a database server. SQL Server supports the more efficient IPC mechanisms for client-server communication: named pipes, shared memory, and Windows sockets. Using SQL Server's *Net-Library*, database clients may choose protocols that work with any of these three mechanisms.

7.1 Cost Analysis

We use the *KubMessage* benchmark to determine the cost of IPC on Windows 2000. We are specifically interested in the overhead of the kernel-supplied named pipe service verses a roll-your-own service that a process might choose to implement using shared memory or Windows sockets.

Two instances of *KubMessage* run on the same machine. One instance of *KubMessage* acts as a client, sending messages to the server in page-sized units, varying from 8 KB to 256 KB. The other acts as a server, reading these messages over each IPC mechanism and sending them back to the client in their entirety. The client subsequently reads the response. For efficiency, the first four bytes of the message describe the full length of the message; this way, the size of a message can be easily determined in a protocol-independent way.

The following IPC mechanisms are tested by *KubMessage*:

- **Named pipes**. The server creates a named pipe using the `CreateNamedPipe` function, which permits an unlimited number of client connections. Using the `ConnectNamedPipe` function, each client connection opens a private bidirectional pipe between the client and the server.
- **Shared memory**. The only way to share memory between processes is to map a file to virtual memory and share the map's object handle with another process.[3] *KubMessage* uses the `CreateFileMapping` and `MapViewOfFile` API functions to accomplish this. Instead of maintaining its own file, it uses the system page file by passing a null pointer to the `CreateFileMapping` function. In addition to the shared mapping, the two instances of KubMessage also share two mutexes: one guards the shared memory against simultaneous access, while the other sig-

[3] Technically, the `ReadProcessMemory` function also provides inter-process memory sharing, although this function requires the possession of special security rights by the calling process and is supplied by the operating system primarily for interactive debuggers.

nals alerts each instance to the arrival of new, unprocessed messages in this memory. The mutexes are necessary due to the lack of kernel support for shared memory synchronization.

- **TCP/IP**. The *KubMessage* server uses standard Berkeley socket interface to bind to a port and accept connection requests.
- **Intra-process function calls**. Not an IPC technique by definition, we use intra-process function calls to compare the overhead of IPC to that of message passing within a process. A pointer to the message is passed from a "client" procedure to a "server" procedure via a standard procedure call. The server process makes a local copy of the message before returning.

KubMessage makes all latency measurements at the client end of communication. Specifically, it measures the time it takes to connect to the server, the time it takes to write the entire message to the IPC buffer, the time it takes to receive the first 4 bytes of the response from the server, the time it takes to receive the remainder of the message back from the server, and the time it takes to shutdown the communication channel.

Figure 7.1.1 illustrates the results of the overall *KubMessage* benchmark. The intra-process mechanism performs most efficiently for small message sizes, as we expect, and grows linearly with message size. The shared memory IPC technique is nearly as efficient as intra-process message passing, and is even more efficient as message size increases beyond 64 KB. We see this behavior because we specifically coded the "server" end of the intra-process technique to make a copy of the message, whereas the shared memory technique maintains a single copy of the message.

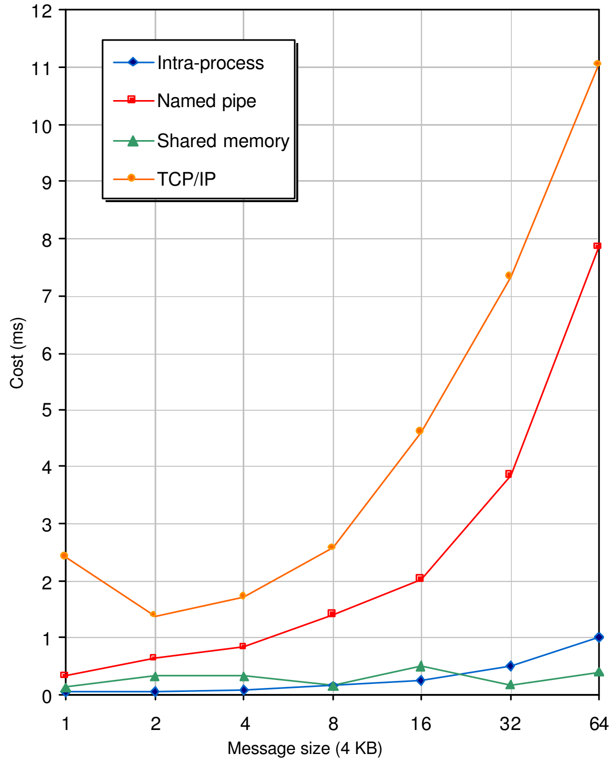

Figure 7.1.1

Costs of various IPC techniques by message size (source: KubMessage *benchmark)*

Since both named pipes and TCP/IP must copy the message from the client buffer to the operating system buffer, then from the operating system buffer to the server buffer, and subsequently perform the process in reverse, they are more expensive and become increasingly more so as message size increases. TCP/IP performs the worst because it suffers from the additional overhead of the protocol stack, but this overhead becomes less relevant as the message size increases. In our experiments we set the read and write named pipe buffer sizes to 64 KB explicitly, and use the system default read and write buffer sizes of 8 KB for TCP/IP.

When we examine the costs of specific IPC tasks, such as connecting, writing, and reading, as illustrated in Figure 7.1.2, we notice that the costs of IPC tasks vary with IPC technique, and that the costs generally do not remain proportional as message size increases. Connection times remain relatively constant, except in the case of TCP/IP. TCP/IP connections are typically more expensive the first time they are made, since the process must resolve the hostname and warm up the socket library. This may include an expense for mapping the socket DLL into the process' virtual address space and initializing internal socket data structures. Consequently, TCP/IP connect time diminishes as more messages are exchanged.

Write and reads become more expensive for named pipes and TCP/IP as message size increases. We find the write cost to a named pipe alarming for large message sizes. While a 256 KB memory copy takes about 1 ms using the intra-process mechanism, a 256 KB synchronous write onto a named pipe using the `WriteFile` function takes over 4 ms, almost 2 ms longer than TCP/IP's corresponding `send` call. We attribute this extra cost to the size of the read and write buffers the system maintains for the named pipe. As our message size extends beyond buffer capacity, the system must either transmit the message in chunks—continuing to block the client as it waits for the server to read from the full buffer—or allocate more memory for the buffer. Either method is expensive. We discovered that by increasing the buffer size from 64 KB to 256 KB + 50, we reduce the cost of writing a 256 KB message by 50%. We posit that the system must store state information in addition to the actual message in the buffer because a buffer size of 256 KB is not sufficient to realize this gain.

We use the `setsockopt` function to adjust the TCP/IP send and receive buffers from their 8 KB defaults, but actually notice performance degradation as the buffer size increases beyond 16 KB. The socket library must be allocating buffer space as it needs, and not at socket create time; the expense of this allocation ends up outweighing any possible gain. For longer IPC conversations, where the socket is kept open for an extended period of time, we do find gains.

To summarize, we see that shared memory IPC performs best, especially for larger message sizes, because it avoids buffer copies. For IPC on a single system, the named pipe technique fairs second best. For inter-system IPC, we find that TCP/IP outperforms named pipes since the named pipe implementation requires the services of IPX, TCP/IP, or NetBEUI (depending on what protocols the client and server share) in addition to its own overhead.

7.2 Support for Database Management

As we observe in section 6, SQL Server is architected as a single process, so IPC is not necessary for the tasks Stonebraker has in mind, such as requesting disk I/O from other processes [21]. This doesn't make IPC irrelevant, however, since its database clients use some form of IPC to communicate with the database server.

We see that shared memory is the most efficient IPC mechanism, but this mechanism does not provide any synchronization or notification support that is a prerequisite of message passing. For a rather modest amount of overhead, named pipes comfortably supply this functionality with a straightforward and versatile API. In addition to providing security, multiple instance support, and bidirectional communication, the named pipe implementation offers some interesting functionality via the `CallNamedPipe` and `TransactNamedPipe` functions, which combine writes and reads into a single system call.

SQL Server uses named pipe IPC by default mostly for historical reasons [6]; Microsoft research agrees with our previous analysis that inter-system IPC performs best using TCP/IP. In our own testing, we found that query execution speeds vary negligibly as we change IPC mechanisms, since the query execution time dominates.

7.3 Suitability of IPC

Despite the robustness of the Win32 IPC API, Stonebraker's question still looms over us: does the cost of IPC outweigh the benefits of the flexible client-server architecture? The alternative, of course, would be to employ some form of the process-per-user architecture, perhaps where the database server and database client are linked together into a single executable image. This approach would eliminate any IPC cost, for sure, but would not allow for multiple client connections. If the client statically linking with

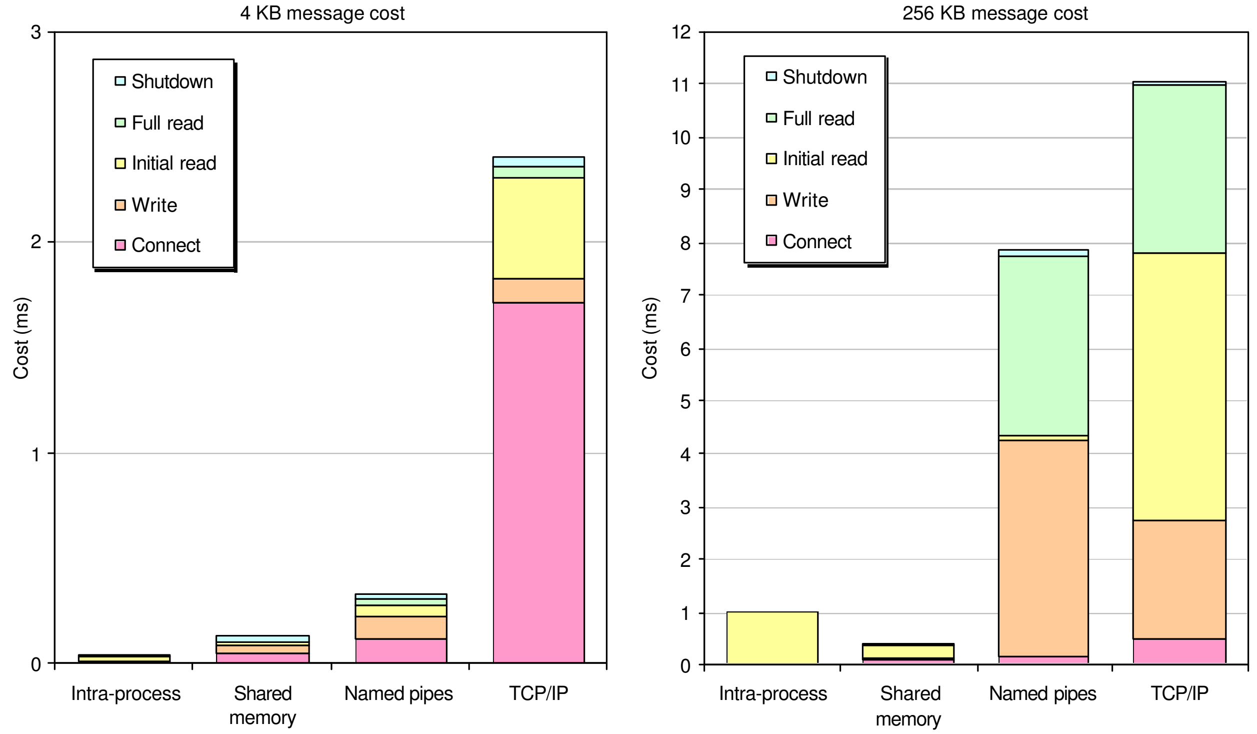

Figure 7.1.2
Costs of performing IPC tasks with various IPC techniques using 4 KB and 256 KB message sizes (source: KubMessage *benchmark)*

the database server happens to support clients of its own, however, we see something interesting. Consider the example of a web server that statically links with a database server. In this case, the web server receives client requests over HTTP, and these client requests, which presumably must be served by operating on data in the database, prompt the web server to open multiple database connections on behalf of these multiple web clients. In essence, the database server is now serving web clients through a web server proxy, and the only difference between this model and the single-server model is that the IPC costs between the web server and the database server have been eliminated. The cost of IPC has been traded for the much lower cost of a statically linked procedure call. Interestingly, most business with web servers move in the opposite direction, adding more layers of IPC on top of the single-server database architecture by means of middleware. Some researchers have balked at this trend, instead touting the advantage of statically linked databases to a world that remains quite fond of the client-server model [15].

The analysis of scheduling and process management in the previous section demonstrates how the cost of a context switch is at least an order of magnitude more expensive than a simple procedure call (see Figure 6.2.1). Since the client-server architecture requires, at minimum, shared memory and a context switch, the costs associated with IPC when compared with the static linking approach should not be overlooked. Amdahl's Law tells us to make the common case fast and not fret about the rest, but it remains unclear where IPC lies in this continuum. For long-running queries that don't return much data, IPC costs may be negligible. For fast-running queries that return much data, these costs may be substantial.

Ultimately, then, the relevance of IPC cost is dependent upon the average latency of database requests and the amount of data transmitted between the client and the server, and these metrics are highly specific to individual use cases. Also important in this consideration is the flexibility the client-server model provides, allowing multiple, disparate applications to access a common data source simultaneously. SQL Server has chosen to support the more common use case with the client-server model, and the plethora of IPC mechanisms provided by Windows 2000 have certainly been supportive. Ultimately, however, the decision to accept the cost of IPC with the client-server model or turn to an embedded database system must be evaluated on a case-by-case basis. Until IPC becomes, in a relative sense, no more expensive than a procedure call there will be no "right" answer.

8 Consistency Control

There are two types of operating system services that help database systems maintain consistency: crash recovery and locking. Stonebraker considers both areas problematic. In the case of crash recovery, operating systems may not provide a selective force-out option from their file buffers to guarantee that transaction data, or even the log entries for these transactions, are successfully written to disk before a transaction commits. In the case of locking, operating systems typically provide file-level read and write locks, but not locks with the granularity or escalation properties that a DBMS desires. We will consider each of these criticisms in turn.

8.1 Selective Force-Out Functionality

Most applications perform file I/O using the general `ReadFile` and `WriteFile` functions; for this approach the only force-out function Windows provides is `FlushFileBuffers`. This function always writes all buffered data to disk, not allowing a program to specify a data range to flush. While the interface is simple, it goes no further in meeting the needs of a DBMS than the UNIX buffer pool did back in 1981 [21].

Fortunately, Windows does provide a selective force-out option when file I/O is carried out using a different approach, by mapping files in an application's virtual memory space. After a program maps views of a file (in units of the page size) using the `MapViewOfFile` function, it may flush any page or range of pages within this view using the `FlushViewOfFile` function. By using memory-mapped files instead of typical file I/O, programs gain granular control over the flushing of individual byte ranges of a file, all the way down to the sector size of the disk.

To combine the benefits of prefetching with the finer control over individual pages, Windows allows memory-mapped files to be simultaneously managed by the buffer pool. In this case, the buffer pool operates in the mode prescribed by a program (i.e. *sequential* or *random*) and the application's view of the file and the buffer manager's view of the data are both mapped to the same segments of physical memory. In other words, the virtual memory manager provides consistency between these two views, which is made possible because the buffer manager uses the same file-mapping interface of the memory manager to buffer and prefetch data. By combining traditional file I/O with a memory-mapped view of the file, an application now has selective force-out ability over buffer manager pages, because, with the consistency of the two views, a flush of a mapped view by a program necessarily corresponds to a flush of that data from the buffer pool. This appears to provide the selective force-out option desired by database systems.

Note that when physical memory is low, or dirty pages are in memory for more than 5 minutes, the virtual memory manager proactively begins writing dirty pages to the mapped files backing them. Since the *modified page writer* thread runs at a higher priority level than normal application threads [18], the operating system may actually hinder database performance by writing dirty pages at exactly the wrong time. This also has ramifications on transaction processing, since the virtual memory manager may write out uncommitted transaction's data before the log entries describing the transaction. If a system failure were to occur between these two steps, the database likely would not be able to recover.

A database system that does not want its pages to be flushed to disk by the virtual memory manager may use the `VirtualLock` function to lock pages into physical memory. To protect the system from the instability that may arise when too much physical memory is locked, Windows limits an application to 30 locked pages by default and never allows an application to adjust its maximum allotment much beyond this [18]. Since a database system cannot guarantee the order in which pages are written to disk using the memory-mapped technique, it cannot use it. Once again an operating system service has come tantalizingly close to exactly what a database system needs, but ultimately the miniscule "gotcha" is enough to disqualify the entire service from consideration.

SQL Server is forced to manage its own buffers, and dynamically tweak the size of these buffers in response to available system memory by using the data from the `GlobalMemoryStatus` system call. Ultimately, SQL Server winds up duplicating various memory manager services mainly so it has control over the exact time pages are flushed to disk.

8.2 Locking

Windows 2000 is notoriously deficient in the area of locking services. While it does provide the ability to lock byte ranges of a file using shared locks and exclusive locks via the `LockFile` function, it does not offer the ability to escalate shared locks to exclusive locks. If the holder of a shared lock attempts to obtain an exclusive lock on the same byte range, the system call blocks indefinitely.

If a database system desires additional lock types, lock escalation, or multiple levels of locking granularity, database system designers must write this functionality themselves. We use the *KubLock* benchmark to evaluate whether or not designers should base their locking implementation on the primitive locking mechanism already provided by Windows 2000, or disregard this mechanism and develop their own locking implementation from scratch.

KubLock tests the efficiency of file system locks and a custom locking package we adapt from the work of Ruediger Asche [1]. Our locking package implements functionality identical to that provided by the file system; it includes methods to acquire and release read locks and write locks, but does not provide any mechanism for deadlock detection, lock escalation, or starvation prevention. As part of the implementation, we use two operating system primitives: event objects, which are synchronization objects that can be explicitly signaled and reset, and interlocked variable access functions, which perform atomic arithmetic operations on integer variables.

In order for the first read or write lock to be acquired, the thread desiring access must wait for the *drop-to-zero* event object to be signaled; Windows ensures that only one waiting thread will get the signal. Once the thread gets the signal, it sets a flag indicating its desire for read or write access. If the flag is set for read access, additional read locks are obtainable by other threads simply by incrementing an associated reader count variable using the `InterlockedIncrement` atomic increment function. When a thread finishes with a read lock, it decrements the reader count variable using the `InterlockedDecrement` function. A thread holding a write lock doesn't bother incrementing or decrementing any variable, since the lock compatibility matrix guarantees that no other thread will simultaneously be holding the lock. Once the last thread holding the lock finishes, it signals the *drop-to-zero* event so that the lock may be picked up by another thread interested in acquiring the lock for either read or write access, and the process repeats. We have omitted some subtleties of our implementation; the interested reader should examine the *KubLock* source code for more details.

Since Windows makes no guarantees about which thread in a set of waiting threads is awoken by an event object's signal, this locking mechanism makes no guarantees about fairness. Also, the implementation waits indefinitely for the *drop-to-zero* event, but could be adapted to timeout after a given amount of time and check for deadlocks. Compare this to the file-based locking technique: the `LockFile` function has no wait timeout, so there is no possibility of incorporating deadlock detection support.

The *KubLock* benchmark exercises the native file locking mechanism and our custom locking package by acquiring and releasing 5,000 read locks and then acquiring and releasing 5,000 write locks. These lock acquisitions are all performed by a single thread, so as to hone in on the cost of locking overhead without having to consider the related costs of thread scheduling and contention. We chose to perform each operation 5,000 times so that the benchmark runs long enough to produce quantifiable differences. The benchmark shows our custom read locks to be 9% more efficient than file system read locks and our custom write locks to be 68% more efficient than file system write locks. The results are illustrated in Figure 8.2.1.

The *KubLock* benchmark, in its second phase, creates an additional thread. While the first thread attempts to acquire 5,000 write locks, the second thread acquires and releases read locks in an infinite loop. Both threads run at the same high priority level, above other processes and most system threads. Our goal here is to get a rough idea of the relative throughput of the two locking techniques. The results show that our custom locking package allows the second thread to acquire and release 4,567 read locks while the first thread acquires and releases the 5,000 write locks; the total processing time here is 35 ms. The file system locking implementation allows for only 13 read locks to be acquired and release for the same 5,000 write locks; the total processing time is 22 ms. Our custom technique appears to allow for substantially more data throughput.

8.3 Support for Database Management

Based on the results of *KubLock*, we conclude that the primitive Windows 2000 locking mechanisms are not sufficient for use as the foundation of a full-feature database locking mechanism. Despite the ease of use of the `LockFile` call, database designers lose both efficiency and the ability to prevent deadlocks, which are unacceptable limitations.

The Win32 API designers should consider supporting abstract locking functions independent of files. Any multithreaded application that maintains buffers of shared data, including SQL Server, will likely benefit by a more abstract and feature-rich locking interface.

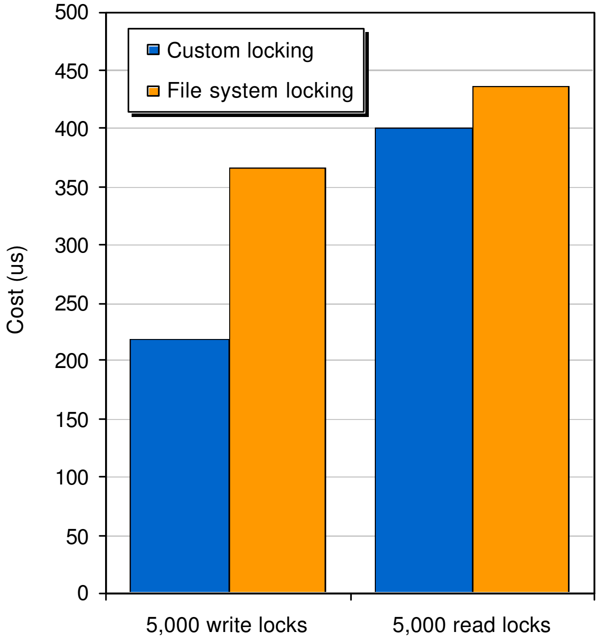

Figure 8.2.1

Comparison of the cost of acquiring 5,000 locks in sequence using our custom user-mode locking package and the Windows 2000 file locking function `LockFile` *(source:* KubLock *benchmark)*

9 New Domains

Stonebraker addresses five principal services required by database management systems, which we have addressed in detail in the preceding five sections. Now we briefly consider additional services provided by the operating system that are useful to database management systems and evaluate how well they meet the needs of a modern DBMS and its administrators.

9.1 Security Management

Especially in recent years, data security has become a critical concern for system administrators. Windows 2000 is certified as EAL4-compliant by the international Common Criteria body, meaning it includes support for secure logons, access controls, auditing, and trusted facility management, among many other features [18].

Windows 2000 manages user accounts and privileges either locally or via a centralized domain controller or Active Directory (AD) server. Client-server processes such as SQL Server can use these privileges and even define new privileges. Win32 functions such as `ImpersonateNamedPipeClient` and `ImpersonateLoggedOnUser` are extraordinarily useful in that they allow a server process to temporarily assume the security context of an arbitrary user. In this manner processes can access resources using client credentials and never have to explicitly make security checks; the system handles this automatically by returning "access denied" error codes when a process attempts to manipulate resources the impersonated user is prohibited from using. This alleviates error-prone development efforts by greatly simplifying a server process' code base. It also allows system administrators to configure user accounts once and have these accounts work with a wide variety of server applications.

SQL Server 2000 exploits Windows authentication techniques, but also provides the ability to create SQL Server-specific users for compatibility with clients connecting from non-Windows platforms.

The NTFS file system supports data encryption that is opaque to user processes such as SQL Server. When this feature is enabled, all user data is stored in encrypted for on the file system and only the owner of the data has the security key needed to decrypt the stored data.

Additionally, Windows natively supports SSL communication mechanisms that provide secure network communications. The SQL Server Net-Library takes advantage of this functionality.

9.2 Performance Monitoring

Windows 2000 provides three distinct services that allow system administrators to track the performance metrics of processes:

1. **Performance Monitoring**. The performance monitoring interface, part of the Win32 API, allows processes to register performance statistics on the system and supply a stream of performance information. The Performance Monitor utility that ships with Windows displays these statistics graphically, and updates its display as fast as once per second.
2. **Simple Network Management Protocol** (SNMP). Windows allows processes to supply a Management Information Base (MIB) of performance statistics that SNMP management tools can navigate, and send SNMP traps alerting these managers to critical events.
3. **Windows Management Instrumentation** (WMI) uses the industry-standard Information Model (CIM) to display performance statistics in a web interface.

SQL Server supports all three of these performance monitoring technologies to provide up-to-the-second statistics such as memory usage, concurrent connections, CPU utilization, and transactions per second. Such statistics are invaluable to administrators who are sensitive to resource utilization and want to tune the DBMS for maximum efficiency.

The drawback to these technologies is their redundancy. Microsoft should find a way to provide all of this functionality through a single interface to reduce development time and avoid confusion.

9.3 Programmatic Interfaces

Windows 2000 ships with support for the Component Object Model (COM), Remote Procedure Calls (RPC), and several scripting languages including JavaScript and VBScript. SQL Server leverages these technologies to provide administrators and developers methods for programmatically tuning and administering the server using Rapid Application Development (RAD) tools. For example, using the SQL-DMO COM object model of SQL Server, a Visual Basic developer can log on to a SQL Server and enumerate the space available on all of its managed databases using no more than 10 lines of code [6].

10 Conclusion

We reevaluate operating system support for database management and find that in some areas the operating system has definitely improved, giving database systems a more efficient and flexible API to work with. There are areas, however, particularly buffer management and consistency control, where we see virtually no improvement at all.

We discover several new operating system services—such as security and performance monitoring—that are now being exploited by database systems. We intentionally neglected putting these new systems under the same scrutiny as those originally cited by Stonebraker because these systems are not on the critical path to efficient query processing; rather, they provide support to database administrators to ease the burden of management. We are encouraged by shift in focus from efficient computer resource utilization to that of efficient human resource utilization, since the latter category has not received nearly as much attention despite the fact that both are crucial to the ultimate success of a DBMS.

Table 10.1 summarizes the operating system services we explore, a brief analysis of their suitability for database systems, and our recommendations for further improvements. We attempt to offer reasonable advice, and believe that these improvements will greatly assist many Win32 developers, including, but not limited to, database developers.

10.1 Shortcomings

We attempt to cover a broad range of operating system services in a short space. Certainly we do not do justice to the intricate design of both Windows 2000 and SQL Server 2000. The interested reader should explore David Solomon's *Inside Microsoft Windows 2000* [18] and Kalen Delaney's *Inside Microsoft SQL Server 2000* [6] for the details—these are two very insightful books. Additionally, the Microsoft Developer Network Technology Group publishes some interesting technical articles concerning the resourceful uses of the Win32 API [1][22].

10.2 New Directions

We understand the motivations behind new operating system designs, but our analysis shows that Windows 2000 is actually doing a very good job of balancing between easy-to-use least-common-denominator interfaces and robust, efficient, highly scalable interfaces. SQL Server has proven that designing a process to these interfaces can provide excellent performance. Part of the problem with other database systems, such as Oracle, is its need to support multiple platforms. This platform-independent nature forces Oracle to take a least-common-denominator approach to which services it utilizes from operating systems—in other words, don't expect Oracle to be taking advantage of Win32 fiber or scatter/gather I/O functions anytime soon.

Though perhaps wishful thinking, a common OS interface like POSIX but encompassing Win32 functionality or, at minimum, an agreed upon list of OS service prerequisites would alleviate this problem. It would define interfaces operating system services, such as fibers and asynchronous I/O routines, that disparate OS vendors could standardize and implement. This would allow developers to support multiple platforms without suffering from the inefficiencies and code duplication inherent in the least-common-denominator approach.

More practically, we hope operating system designers consider the suggestions we outline in Table 10.1. Doing so will allow Windows to meet virtually all of the basic needs of a DBMS and many of today's other prevalent applications.

Operating system service:	Database suitability:	Areas of improvement:
Buffer pool management	**Poor**. Efficient scatter/gather I/O techniques are incompatible with the buffer manager and memory manager, leading to user-level duplication of these services.	The kernel should support scatter/gather I/O to/from mapped views of a file.
File system management	**Good**. Asynchronous I/O and completion ports improve efficiency and throughput while recoverability mechanisms protect the integrity of file system metadata upon which database data is stored.	The file system driver should make an effort to cluster large file allocations.
Scheduling and process management	**Very good**. Threads are scheduled by the kernel to support efficient symmetric multiprocessing of concurrent client connections and background tasks.	The kernel should supply a thread pool service for processes that deal with a high volume of concurrent connections.
Interprocess communication	**Excellent**. The Win32 API offers a variety of efficient, robust IPC mechanisms for client-server communication. (Note, however, that some database workloads may benefit from an embedded architecture that eliminates IPC.)	No areas cited for improvement.
Consistency control	**Poor**. The selective force-out mechanism still allows pages to be written to disk without DBMS approval, and the primitive locking mechanisms do not even provide a practical or efficient foundation upon which to build a robust custom locking package	The virtual memory manager should consult an application before writing dirty memory-mapped pages to disk, and the Win32 API should provide generic locking services, with lock escalation support, to complement its existing set of synchronization primitives.
New domains (security, performance monitoring, programmatic interfaces)	**Very good**. The Win32 API offers robust support for security, an overly sufficient supply of performance monitoring interfaces, and technologies to support RAD development of programmatic application administration.	The plethora of performance monitoring functions is confusing; WMI in particular is too complex.

Table 10.1
Summary of operating system support for database management systems

References

[1] Asche, Ruediger. “Compound Win32 Synchronization Objects.” Microsoft Developer Network Technology Group (July 1994).

[2] Berchad, Brian. “The Increasing Irrelevance of IPC Performance for Microkernel-Based Operating Systems.” School of Computer Science, Carnegie Mellon University (March 1992).

[3] Bershad, Brian, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, E. Sirer. “SPIN – An Extensible Microkernel for Application-specific Operating System Services.” Dept. of Computer Science and Engineering, University of Washington, Seattle, Technical Report 94-03-03 (March 1994).

[4] Chen, J. Bradley, Yasuhiro Endo, Kee Chan, David Mazières, Antonio Dias, Margo Seltzer, Michael D. Smith. “The Measured Performance of Personal Computer Operating Systems.” Division of Applied Sciences, Harvard University (August 1995).

[5] Chew, Khien-Mien, Avi Silberschatz. “Toward Operating System Support for Recoverable-Persistent Main Memory Database Systems” (1992).

[6] Delaney, Kalen, et al. “Inside Microsoft SQL Server 2000,” Microsoft Press (2000).

[7] Engler, Dawson R., Frans Kaashoek. “Exterminate All Operating System Abstractions” (1995).

[8] Engler, Dawson, Frans Kaashoek, James O’Toole Jr. “Exokernel: an operating system architecture for application-level resource management” (1995).

[9] Engler, Dawson, Frans Kaashoek, James O’Toole Jr. “The Operating System Kernel as a Secure Programmable Machine.” *Proceedings of the Sixth SIGOPS European Workshop* (September 1994).

[10] Fellig, Daniel, Olga Tikhonova. “Operating System Support for Database Management Systems” (2000).

[11] Forman, Joshua J. Mark A. Luber. “Support for Database Management Systems in Modern Operating Systems.” CS265: Database Systems, Harvard University (October 2002).

[12] Grimm, Robert, Michael M. Swift, Henry M. Levy, “Revisiting Structured Storage: A Transactional Record Store,” University of Washington (2000).

[13] Hsieh, Wilson, Frans Kaashoek, William Weihl. “The Persistent Relevance of IPC Performance: New Techniques for Reducing the IPC Penalty.” MIT Laboratory for Computer Science (1993).

[14] Hulse, David. “Operating System Support for Persistent Systems: Past, Present and Future” (2000).

[15] Seltzer, Margo, Sleepycat Software. “High Performance != Client/Server: The Case for Embedded Databases.” High Performance Transaction Systems Workshop Paper Submission. *James Hamilton* (personal home page). http://research.microsoft.com/~jamesrh/hpts2001/submissions/MargoSeltzer.htm#ref1 (October 2001).

[16] Seltzer, Margo, Christopher Small, Keith Smith, “The Case for Extensible Operating Systems” (1995).

[17] Small, Christopher, Margo Seltzer. “VINO: An Integrated Platform for Operating System and Database Research,” *Harvard Computer Science Laboratory Technical Report TR-30-94* (1994).

[18] Solomon, D. A. “Inside Windows 2000,” Microsoft Press (2000).

[19] “SQL Server Fast Facts.” http://www.microsoft.com/sql/evaluation/overview/2000/fastfacts.asp (August 2000).

[20] “SQL Server Performance Tuning Questions & Answers: Q&A #8,” *Sql-Server-Performance.Com*. http://www.sql-server-performance.com/q&a8.asp (December 2001).

[21] Stonebraker, Michael. “Operating System Support for Database Management” (1981).

[22] Vert, John. “Writing Scalable Applications for Windows NT.” Microsoft Developer Network Technology Group (June 1995).

[23] Yang, Li, Jin Li. “Operating System Support for Databases Revisited” (2000).