Operating System Support for Databases Revisited

Li Yang and **Jin Li**
Department of Computing Sciences
University of Wisconsin, Madison
{yangli, jinli}@cs.wisc.edu

Abstract

In 1981, Stonebraker wrote a scathing criticism of OS support for databases (and how lacking it was). This paper is pointed to as why operating systems fail to adequately support databases. Since it is 20 years old, it motivated us to revisit OS support for databases. In this project, based on Stonebraker's framework, we reexamined two modern operating systems, Solaris and Windows NT, to see how well they would support databases in the modern world. We discuss buffer pool management, file system support, scheduling and process, and concurrency control in Solaris and Windows NT. For each topic, we first summarize the criticisms from Stonebraker's paper, then discuss the current situation in Solaris and NT separately. As a result, we can see that both NT and Solaris offer better services in some aspects mentioned by Stonebraker's paper, providing a suitable environment for running a reliable, multiuser database system. We also discuss some features that are not mentioned by Stonebraker's paper, but are of special interests to and are used in DBMSs.

1 Introduction

During the early 1980s, Unix was considered a hostile environment for running a multiuser database system. As pointed in Stonebraker's paper in 1981 [?], operating systems often provide wrong services or severe performance problems exist. It is true that early systems did indeed present large obstacles for developing database systems, since they did not provide any form of IPC, did not provide any form of record locking, etc.

In a general sense, operating systems are designed to provide least-common-denominator service to a variety of applications. They export few internal kernel facilities. As a result, resource intensive applications such as database management systems, are often poorly served by the operating system. DBMSs often go to great lengths to bypass normal kernel mechanisms to achieve acceptable performance. For example, most mature database systems implement their buffer management, concurrency control, and crash recovery in user space.

Stonebraker's paper is pointed to as why operating systems fail to adequately support databases. However, twenty years have passed. We feel that it is high time to revisit operating system support for databases. In this project, based on Stonebraker's framework, we reexamined two modern operating systems, Solaris and Windows NT, to see how well they would support databases in the modern world. We selected the two operating systems because of their popularity and commercial successfulness. However, we should notice that there are some research prototypes which proposed initial ideas that can provide better support for database systems. Such systems include Scheduler Activations [?], which proposed a method for sharing scheduling policy between kernel and user; Cao's work on application-controlled file caching [?], which addresses buffer cache management; VINO's kernel architecture [?], which addresses the limitations of conventional operating systems, etc. In this paper, we only address Solaris and NT.

Based on Stonebraker's frame work, we discuss buffer pool management, file system support, scheduling and process, and concurrency control. For each topic, we first summarize the criticisms from the original paper, and their hope proposed in the paper. Then we discuss the current situation in Solaris and NT separately. As a result, we can see that both Solaris and NT offer better services in some aspects mentioned by Stonebraker's paper, providing a suitable environment for running a reliable, multiuser database system. We also discuss some features that are not mentioned by Stonebraker's paper, but are of special interests to and are used in DBMSs.

2 Buffer Pool Management

2.1 Criticism

Traditionally, the OS buffer pool management is not desirable for DBMS for the following reasons: (1) poor performance because of the memory to memory copy, (2) no prefetching advice, (3) no block management advice, and (4) no selected force out support for concurrency control.

The strategy used by most DBMSs is to maintain a separate cache in user space, which is managed by a DBMS specific algorithm to circumvent the problems mentioned above. Therefor, the paper argues that a better OS buffer management facility should offer some of the following properties: (1) Improve the performance, reduce access overhead, (2) Smart way in more accurate prefetching, (3) Accept advice from a DBMS concerning the replacement strategy, and (4) Selected force out to support crash recovery (write ahead log).

2.2 Solaris support

As a whole, Solaris buffer management is general. Though Solaris doesn't offer any smart way in more accurate prefetching and can't accept from a DBMS concerning the replacement strategy, some improvement has been seen in other aspects. We discuss synchronization flags and direct I/O below. The former enforces file integrity and data integrity, which can be used in implementing selected force out to support crash recovery (write ahead log)

2.2.1 Synchronization flags

Solaris provides file flags that set different levels of data synchronization and file integrity. Thus, DBMSs have some flexibility when read and write files, albeit at increasing cost as the level of integrity is increased. For example, pure synchronous flag can be used in implementing selected force out. We discuss the flags below.

Without any flags, the default situation is essentially asynchronous in nature. Three applicable flags can be set in the open system call: O_SYNC, O_RSYNC, and O_DSYNC. The O_SYNC flag tells the operating system that file data and inode integrity must be maintained. That is, when the O_SYNC flag is set on a file descriptor and a write is done to the file, the write system call does not return to the calling process or thread until the data has been written to the disk and the file inode data has been updated. The O_DSYNC flag also provides synchronous write functionality, meaning that the write system call does not return to the calling process until the write data has been committed to the disk storage. However, O_DSYNC does not require that the file inode data be committed to the disk. The O_RSYNC flag provides for read synchronization and is used in conjunction with the O_SYNC or O_DSYNC flag. With the O_SYNC, file integrity is enforced; with O_DSYNC, data integrity is enforced, meaning that any pending writes to the file are completed before the read is done.

2.2.2 Direct I/O

To provide near-device performance, many file systems offer the option to bypass the file system cache through a mechanism known as direct I/O. This option reduces the overhead of managing cache allocation and completely removes all interaction between the file system and the memory system. In many cases, the resulting performance can be many times worse, since there is no cache to buffer reads and writes, but when caching is done in the application, direct I/O can be a benefit. Another important use of direct I/O is backups, where we don't want to read a file into the cache during a backup.

Applications such as databases do their own caching, and direct I/O offers a mechanism to avoid the double caching that would occur if applications were to use a regular file system. Without direct I/O, an application reads a file block in to the Solaris file system cache, and them reads it into the database shared buffer (e.g., Oracle's block cache), so the block exists in two places. However, with direct I/O, the block is read directly into the database cache without passing through the regular file system cache.

Because direct I/O bypasses the file system cache, it also disables file system read-ahead. This means that small reads and writes result in many I/O requests to the storage device that would have otherwise been clustered in to larger requests by the file system; hence, direct I/O should only be used for random I/O or large-block sequential

I/O.

Another side effect of direct I/O is that it does not put a load on the Solaris memory system and removes the typical paging that can be seen when a regular file system is used. This effect is a frequent motivator for the use of direct I/O, but note that the new Priority Paging feature (discussed in Section 2.1.3) of Solaris provides similar separation between the file systems and applications.

2.3 NT Support

NT does not specifically provide support for any of the features mentioned in Stonebraker's paper. NT does however, provide support in buffer management and I/O for DBMS in the following aspects, which are used in SQL Server.

1. **Reserving virtual address space.** NT supports the notion of reserving virtual address space and then optionally committing and decommiting physical store behind it, where physical is either real memory or page file space. SQL Server uses this to allow its cache size to dynamically resize itself in response to other memory demands on the system.

2. **Scatter/gather I/O.** Another NT feature used in buffer and I/O management is scatter/gather I/O. This allows DBMS to read and write from a contiguous range on disk and place the resultant data in discontinuous addresses (pages) in the buffer pool without performing an intermediate copy.

3 File System Support

There are two concerns in Stonebaker's paper regarding file system support. (1) DBMS manages structured records on top of character arrays of dynamically varying size, which often results in physical incontiguity. (2) Unix uses three tree structures to store information about file systems: inode structure, a hierarchical directory structure, and B-trees index.

To solve these limitations, the author proposed the following desired services: instead of providing records management on top of character arrays, it is possible to do the converse. That is, OS designers should contemplate providing DBMS facilities as lower level objects and character arrays as higher level ones. Specifically, they believe that a DBMS prefers a so-called extent based file system to one which scatters blocks. As we can see, both Solaris and NT provide much better support in file system as discussed below.

3.1 Solaris Support

Solaris not only supports extent based file system, but also provides other facilities that are especially suitable for database systems. We will discuss asynchronous I/O, large files, priority paging, extent based file system, sparse files and integrated volume management respectively.

3.1.1 Asynchronous I/O

Asynchronous I/O interfaces have been available in Solaris for some time. It provides the ability to do real asynchronous I/O in an application: applications can issue I/O requests and not have to block or cease working until the I/O has completed. The asynchronous I/O interfaces in Solaris are aioread(3) and aiowrite(3). Two kinds of asynchronous I/O exist in Solaris: file system asynchronous I/O and kernel asynchronous I/O. We will discuss them below.

- **File system asynchronous I/O**

  Asynchronous I/O is implemented using the light-weight process (lwp) system calls, which are the lower level implementation of the user-level threads library. The original implementation of the aioread(3) and aiowrite(3) routines creates a queue of I/O requests and processes them through user-level threads. When the aioread(3) or aiowrite(3) is entered, the system will simply put the I/O in a queue and create an lwp to do the I/O. The lwp returns when the I/O is complete (or when an error occurs), and the calling process is notified via a special signal, SIGIO. As an alternative to the signal-based SIGIO notification, users have the option of calling aiowait(3) after issuing an aioread(3) or aiowrite(3). This will cause the calling thread to block until the pending asynchronous I/O has completed. There's a time-value that can be set and passed as an argument to aiowait(3) such that the system only waits for a specified amount of time.

While the threads library implementation of asynchronous I/O works well enough for many applications, it didn't necessarily provide optimal performance for applications that made heavy use of the asynchronous I/O facilities, such as commercial relational database systems. The reason lies in the overhead associated with the creation, management, and scheduling of user threads. This motivated the implementation of kernel asynchronous I/O, or kaio, which requires less overhead and provided better performance and scalability.

- **Kernel asynchronous I/O**

  Kaio implements asynchronous I/O inside the kernel rather than in user-land via user threads. The I/O queue is created and managed in the operating system. The basic sequence of events is as follows. First, when an application calls aioread(3) or aiowrite(3), the corresponding library routine is entered. Then the library tries to process the request via kaio. A kaio initialization routine is executed to create a "cleanup" thread, which is intended to ensure that there are no remaining memory segments that have been allocated but not freed during the async I/O process. Once that's complete, kaio is called, at which point a test is made to determine if kaio is supported for the requested I/O.

  Kaio first appeared in Solaris 2.4 and has been available, with some restrictions, in every Solaris release since. The restrictions have to do with which devices and software include kaio support and which ones don't. The good news, from an application standpoint, is that the question of whether or not there is kaio support for a given combination of storage devices, volume managers, and file systems is transparent: If kaio support exists, it will be used. If it doesn't, the original library-based async I/O will be used. Applications don't change in order to take advantage of kaio. The system figures out what is available and allocates accordingly.

- **Testing**

  We ran two tests to determine whether or not we have kaio support available for a device file and a regular file. It involves running a small program that calls aioread(3) and using truss(1) on the program to watch the system-call activity.

  Test 1

  ```
  # truss -t kaio,lwp_create aio /dev/rdsk/c0t3d0s0
  kaio(5, 0x080476AC, 0xDFBB72CC, 0xDFBB7000, 0xFFFFFFFF, 0x00001FFF, 0x00000000) = 0
  lwp_create(0x08046ECC, 0, 0xDFADFF6C) = 2
  lwp_create(0x00000000, 0, 0x00000000) = 0
  kaio(AIOREAD, 3, 0x080471CC, 1024, 0, 0x080475CC) = 0
  kaio(AIOWAIT, 0x00000000) = −268438128
  aio succeeded
  ```

  We run this test as root because we are reading directly from a raw device-special file. The truss(1) output shows a kaio system call, followed by two lwp_create calls, and finally one more entries into the kaio routine to do the actual read, followed by the aiowait. The last line is the output from the aio program (aio succeeded). In this test, kaio is in fact a supported facility for this I/O operation, as the kaio call with the AIOREAD flag did not return an error.

  Test 2

  In the second test below, we use a normal file as the argument:

  ```
  # truss -t kaio,lwp_create aio txtfile
  kaio(5, 0x080476A8, 0xDFBB72CC, 0xEF68FB50, 0xFFFFFFFF, 0x00001FFF, 0x00000000) = 0
  lwp_create(0x08046EC8, 0, 0xDFADFFCC) = 2
  lwp_create(returningasnewlwp...) = 0
  kaio(AIOREAD, 3, 0x080471C8, 1024, 0, 0x080475C8)Err#48ENOTSUP
  lwp_create(0x08046E98, 0, 0xDFAD6F6C) = 3
  ```

$lwp\_create(returningasnewlwp...) = 0$
$lwp\_create(0x08046E98, 0, 0xDFACDF6C) = 4$
$lwp\_create(returningasnewlwp...) = 0$
$lwp\_create(0x08046E98, 0, 0xDFAC4F6C) = 5$
$lwp\_create(returningasnewlwp...) = 0$
$lwp\_create(0x08046E98, 0, 0xDFABBF6C) = 6$
$lwp\_create(returningasnewlwp...) = 0$
$lwp\_create(0x08046E98, 0, 0xDFAB2F6C) = 7$
$lwp\_create(returningasnewlwp...) = 0$
$lwp\_create(0x08046E98, 0, 0xDFAA9F6C) = 8$
$lwp\_create(returningasnewlwp...) = 0$
$lwp\_create(0x08046E98, 0, 0xDFAA0F6C) = 9$
$lwp\_create(returningasnewlwp...) = 0$
$lwp\_create(0x08046E98, 0, 0xDFA97F6C) = 10$
$lwp\_create(returningasnewlwp...) = 0$
$lwp\_create(0x08046E98, 0, 0xDFA8EF6C) = 11$
$lwp\_create(returningasnewlwp...) = 0$
$kaio(AIONOTIFY, -542282888) = 0$
$kaio(AIOWAIT, 0x00000000) = 1$
aio succeeded

In this example, we are reading a file in the file system. Since kaio isn't supported for file system I/O, it failed as expected. As is shown, the program entered the kernel with a kaio system call, created a couple of lwps, and attempted to do the async I/O through the kaio facility. The second kaio call failed with the ENOTSUP (error: not supported) error, and the system dropped back to the library implementation, resulting in the creation of 9 lwps up to the completion of the I/O. Note that the last line, aio succeeded from the program output indicates that the aioread(3) was in fact successful – we just didn't use the kaio facility to do it.

It's clear that the user-threads implementation results in a lot more lwp creation and management, which adds overhead and reduces efficiency. So why does Solaris create 11 lwps for a single async I/O request? The answer has to do with the initialization of the async I/O library. The first time an aioread(3) or aiowrite(3) is called, an async I/O library initialization routine is called, which creates several worker threads to do async I/Os. The system is simply getting ready to process several async I/Os, and thus creates four reader lwps and four writer lwps, along with one thread to do file syncs. That accounts for the nine successful lwp_create calls you see above. If you were to add a second aioread(3) or aiowrite(3) to the test program and rerun it with the truss(1), you wouldn't see all the lwp_creates for the second async I/O. The library handles subsequent async I/O requests with the worker threads created during initialization (though it will, of course, create more worker threads to keep up with the level of incoming async I/O requests). While the creation of a pool of worker threads up front helps provide better scalability for the user-threads async I/O facility, it still involves more overhead and a longer code path than kernel async I/O.

3.1.2 Large files support

The original file system and file size limit in Solaris is 2 gigabytes. This size was a constraint imposed by the number of bits in the offset data types in the kernel. The size limitation occurred in three main places: the disk address type, the file system interface and the uio structure used for device drivers and some file system interface.

With databases growing to over 1 terabyte in size and single disk devices growing past 2 gigabytes, application vendors required a mechanism to access data sizes beyond 2 gigabytes. Now Solaris not only provides 64-bit file application interfaces, but it is itself a large file operating system. We discuss them below.

- **64-bit file application programming interface**

  The first 64-bit file I/O interfaces found their way into Solaris in the 2.5.1 release, with the introduction of just two new read and write APIs: aioread64(3) and aiowrite64(3), which are extended versions of the aioread(3) and aiowrite(3) asynchronous I/O (aio) routines that have been available in Solaris for some time. The goal was to provide relational database vendors a facility for asynchronous (async) I/O on raw devices that weren't limited to 2 gigabytes in size. Since a vast majority of Sun servers run some form of database application, it made sense to get 64-bit file support out the door early for these applications.

- **Large file OS**

  Solaris 2.6 was the first Solaris release to provide support for files greater than 2 gigabytes. The file systems were enhanced to deal with files crossing the 2-gigabytes boundary, and a 64-bit file API was introduced to allow applications to access large files. The following changes in Solaris 2.6 provide large-file support.

  1. *Appropriate error handling and checking in the 32-bit interfaces for large files.*

     The open system call will fail to open a large file in a non-large-file application, and all 32-bit file APIs return E_OVERFLOW when they encounter a regular file whose size is greater than or equal to 2 gigabytes. Therefore, within the contest of large-file support, a program is considered to be either large-file sage or large-file aware.
  2. *A new set of large-file summit APIs to access large files.*

     In order to maintain consistency across the various implementations of Unix that are commercially available today, a number of summits were held, including the Large File Summit held in January 1995. The large-file summit specified a new set of interfaces, similar to the Unix 95/POSIX interfaces, but with an extension of 64 added to the name of the interface, including lsee64, open64, read64, write64, etc. Along with the interface name, a data type following the same convention exists for each of those interfaces for data types related to file offsets, such as off64_t instead of the 32 bit off_t. These transitional APIs can be called explicitly from the program, or compilation flags passed at build time can force their inclusion.
  3. *Support for large file complication environments.*

     For compiling programs that are large file aware, meaning that the program can process large files properly, the _LARGEFILE64_SOURCE flag can be set in the source code, which will make all 64-bit functions and data types available to the programmer. It is up to the programmer to make explicit use of the large file interfaces if desired. Alternatively, a programmer can set _FILE_OFFSET_BITS to 64, which will force the inclusion of all the 64-bit interfaces (i.e., all open(2)s will be mapped to open64(2), etc., by the compiler). No source code changes are needed to take advantage of large files. In addition, the set of xxxx64() interfaces will work just fine with any size file.
  4. Other improvements include file systems capable of large files, limited initially to UFS; solaris commands enhanced to be large-file aware and large-file safe; and an NFS implementation that is large-file compatible and safe.

- **Testing**

  We ran a program that calls open(2), and trussed the output for open(2) calls.

  # gcc -o lf lf.c
  # truss -t open lf
  $open("txtfile", O\_RDWR|O\_CREAT, 0777) = 3$
  open succeeded

  # gcc -o lf lf.c -D_FILE_OFFSET_BITS=64
  # truss -t open lf
  $open64("txtfile", O\_RDWR|O\_CREAT, 0777) = 3$
  open succeeded

  To reduce clutter in the column, we deleted the opens of the system libraries that take place when program execution begins. The program name is lf.c, and we first compiled it without any flags. As is shown from the truss, we executed the 32-bit open(2) call. The next compilation included setting _FILE_OFFSET_BITS to 64. This time, the compiler linked to the open64(2) routine, which is what was executed when I ran the program the second time.

Since the file I/O subsystem is tightly coupled to the virtual memory system, large file implementation required 64-bit offset support in several areas of the kernel in order to facilitate the Solaris dynamic page cache for caching of file system objects (files) in physical memory. The virtual memory system added support for 64-bit offsets in the page structure, vnode layers, and segment drivers that manage address space mappings for VM objects. Support was added in the anonymous memory and swap layers, so swap files can be larger than 2 GB. An example is shmget(key_t key,size_t size,int shmflg)system call, which supports large files. This feature has been used in Oracle.

3.1.3 Priority paging

It may be normal to have high paging and scanning rates for file system. However, it is likely that the page scanner will be putting too much pressure on application's private process memory. If we scan at a rate of several hundred pages or more per second, then the amount of time that the page scanner takes to check whether a page has been accessed falls to a few seconds. This means that any pages that have not been used in the last few seconds will be taken by the page scanner when the user is using the file system. This behavior can negatively affect application performance.

Take an OLTP application as an example, which makes heavy use of the file system. The database is generating file system I/O, making the page scanner actively steal pages from the system. The user of the OLTP application has paused for 10 seconds, during which time those pages associated with the application may have been made available for stealing. When the user hits the next keystroke, the user is forced to wait for an application to page in from the swap device, even though the application is running on a system with sufficient memory to keep all of the application in physical memory. In general, if a system seems slow while file system I/O is going on, it may be because certain applications are being paged in and out as a direct result of the file system activity.

To solve this problem, priority paging was introduced. The priority paging algorithm effectively places a boundary around the file cache so that file system I/O does not cause unnecessary paging of applications. The algorithm prioritizes the different types of pages in the page cache: pages associated with executables, shared libraries, including application process memory have highest priority, while regular file cache pages are considered to have lowest priority. By default, priority paging is disabled. To enable priority paging, we set the following in /etc/system: *set priority_paging=1.*

Under Solaris 8, an extended set of paging counters allows us to see what type of paging is occurring. We can see the difference between paging caused by an application memory shortage and paging through the file system. The memstat utility we downloaded shows the three types of paging: executable, application(anonymous), and file(filesys). The memstat fields are shown in the following table, we selected the relavant columns below:

```
         ---- paging ----- - executabl------anonymous------filesys----
         pi   po   fr   sr  epi  epo  epf  api  apo  apf  fpi  fpo  fpf
disabled 375  242  391  119 0    0    0    0    242  132  375  0    259
enabled  379  0    382  154 0    0    0    0    0    0    379  0    382
```

We randomly read our test files, and the scanner is scanning several hundred pages per second through memory. With priority paging disabled, the scanner causes executable pages to be freed and application pages to be paged out to the swap device as they are stolen. On a system with plenty of memory, this is not the desired mode of operation. With priority paging enabled, the scanner is freeing only file pages. The zeros in the executable and anonymous memory columns clearly show that the scanner is choosing file pages first. The activity shown in the fpi and fpf columns means that file pages are being read in and an equal number are freed by the page scanner to make room for more reads.

3.1.4 Other support

- **Extent-based allocation**

  There are two common types of file system space allocation strategies: block allocation and extent allocation. Block-based allocation creates incremental disk space for a file each time it is extended, which is used in traditional Unix file system. Extent-based allocation creates a large series of contiguous blocks each time the file exhausts the space available in its last extent, which is favored in Stonebraker's paper. We mainly discuss extent-based allocation here.

Extent-based file systems allocate disk blocks in large groups at a time, thus forcing sequential allocation. As a file is written, a large number of blocks are allocated when the file is first created; then writes can occur in large groups or clusters of sequential blocks. File system metadata is written when the file is first created; subsequent writes within the first allocation extent of blocks do not require additional metadata writes until the next extent is allocated.

This approach optimizes the disk seek pattern, and the grouping of block writes into clusters allows the file system to issue larger physical disk writes to the storage device, saving the overhead of many small SCSI transfers. Extent-based file systems provide good performance for sequential file access due to the sequential allocation policy and block clustering into larger writes. However, many of the benefits of extent-based file systems may not be realized when the file system is being used for random I/O. For example, if we want to read sequentially through an extent-based file, we only need to read the start block number and the length; then, we can continue to read all of the data blocks in that extent, which means there is very little meta-data read overhead in reading sequentially. In contrast, if we were to read a file in a random manner, we would need to look up the block address for the block we want for every data block read, which is similar to what we would have to do with a block-based file system.

- **Sparse files**

  Some file systems allow the creation of files without allocation of disk blocks. For example, the user can create a 1-gigabyte file by opening a file, seeking to 1 Gbyte, and then writing a few bytes of data. The file is essentially created with a hole in the middle, and although the file size is reported as 1 Gbyte, only one disk block would be used to hold such a file.

  Files with allocation holes are known as sparse files. Solaris support sparse files. Access to locations with a sparse file that has no blocks allocated simply return a series of zeros, and blocks are not allocated until that location within the file is written to. Sparse files are particularly useful when memory mapped files or files for databases are used, since they remove the need for complex file allocation algorithms within an application. For example, a simple database application can store records in a file by seeking to the required offset and storing the record; then the file will only use as much space as there are records in the file and will leave holes where there are empty records.

- **Integrated volume management**

  Volume management allows multiple physical disks to be used as a single volume to provide larger aggregate volume sizes, better performance, and simpler management. Volume managers are typically implemented as a separate layer between the physical disks and present themselves as a virtual disk device. Databases and file systems can be mounted on these larger virtual disk devices, and the same management techniques can be used to manage data within each virtual volume.

3.2 NT Support

1. **Asynchronous I/O**

   Databases, and in particular, OLTP systems, are I/O intensive. Bypassing operating system buffering and accessing data on the raw physical device directly has always been popular for OLTP, especially on Unix. Windows NT supports this raw access to devices, but such raw access is always at a cost of ease of administration. Windows NT boosts I/O performance with asynchronous I/O, where the originator of an I/O request can continue executing, rather than waiting for its I/O request to be completed.

   In addition, Windows NT supports fast disk arrays in RAID 0, 1 and 5 configurations. As a low-cost alternative to hardware RAID, NT also provides software RAID.

2. **Large extent size**

   Using NT File System (NTFS) is very good for database performance due to the fact that it allows larger (64KB) extent sizes. Most NT database products, including SQL Server, use actual files rather than raw devices.

3. **Write through capability**

   The UNIX database management vendors had to historically implement a simple file system on top of raw devices to avoid the block cache which would prevent them writing through to implement WAL. NT's file system has always provided the appropriate support to allow clients to write directly through to the disk on a file by file basis.

4 Process management and scheduling

The main problem discussed in the paper is what model should be used for a multi-user database system. Traditionally, applications were split into multiple processes, and some form of IPC was used to communicate among the processes. Process structure was hierarchical, with a parent process creating a number of child processes. This involves some related issues such as performance and flexibility described below:

- None of the two models, server model or the individual process model, seems attractive as far as performance is concerned, because of the overhead of context switch and messages.
- Since OS has no knowledge about the DBMS, it may deschedule a database process while it is holding a lock.

The possible solutions proposed in the paper are as follows:

- Make these facilities cheaper or provide special fast path functions for DBMS consumers.
- Create a special scheduling class that allows DB to voluntarily relinquish the CPU without being forcibly descheduled.

4.1 NT Support

4.1.1 Multithreading

NT breaks away from this hierarchical process structure in favor of multithreaded process structure. NT's multithreading improves performance a lot. Unlike Unix, Windows NT was designed from the ground up to support multithreading.

4.1.2 MTS(Microsoft Transaction Server) support

The introduction of Microsoft Transaction Server dramatically reduces the complexity of using threads. MTS provides a framework for apartment threading where individual instances of the same class can run on different threads. Developers can take advantage of this framework without explicitly creating threads in their code.

4.1.3 User mode threads package

There is currently no special support for DBMS, like fast path functions or special scheduling class. NT does provides a user mode threads package (Fibers) which allows a process to implement its own scheduler.That is, the process can control which of the DBMS threads are schedulable. The OS can preempt the DBMS thread to run a thread from some other process however.

4.2 Solaris support

4.2.1 Multithreading

In [1], two approaches to organizing a multiuser database System are presented: process-per-user structure and server DBMS structure. But none of them seemed attractive. The criticism is that performance is *low*, concurrency is *poor*

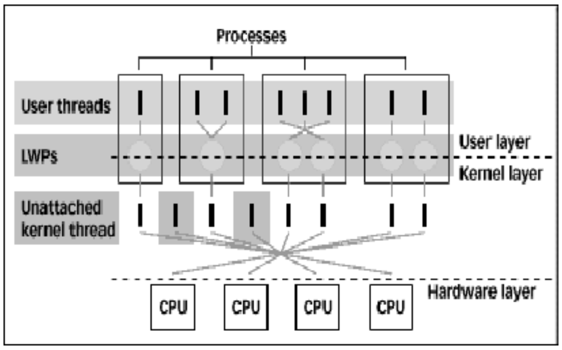

Figure 1: Two-Level Thread Model

and task switching is *expensive*. In current Solaris, the mechanism of multithreading has solved this problem greatly.

The Solaris kernel is implemented with multiple threads of execution to allow concurrency across multiple processors. Within the kernel, multiple threads of execution share the kernel's environment. This architecture is a major departure from the traditional UNIX scheduling model: In traditional Unix implementations, processes maintain a priority and are put on run queues and scheduled ; In Solaris, kernel threads are the fundamental unit that is scheduled and dispatched onto processors.

Processes in user layer also contain one or more threads, which share the virtual memory environment of the process. The threads within each process share the virtual memory environment, open file list, and other components of the process environment. Each user thread requires its own stack. Processes sometimes referred to as "heavyweight", while threads are typically referred to as "lightweight". "Weight" refers to the amount of kernel resources involved in maintaining, scheduling, and executing processes. Fig.1 describes Solaris two-level thread mode.

The diagram is intended to drive home an architectural feature of the Solaris threads implementation. The goal of the implementation is to maximize the efficiency of user threads by not imposing any significant amount of burden on the kernel for the management and scheduling of user threads, while at the same time providing processes the ability to have multiple threads of execution running concurrently on multiple processors. In order to achieve those goals, there needs to be a layer between the user threads in a process and the entity that the Solaris kernel schedules to run on a processor. That is why the multi-layer model exists.

The kernel is blissfully unaware of user threads. The threads library deals with user thread priorities and scheduling user threads without kernel involvement. In order for a user thread to execute on a processor, it must get mapped to an underlying LWP or kernel thread. The LWP and kernel thread are visible to the kernel. Once a user thread has been bound to a LWP and has one-to-one corresponding to LWP/kernel thread pair, it's up to the Solaris kernel to schedule the kernel thread. User-level threads have their own priority scheme and are scheduled with a scheduling thread that is created by the library when a multithreaded program is compiled and executed. This way, multithreaded applications can have literally thousands of threads and not impose any significant load on the kernel.

Till now the scheme of process model in modern Solaris has been shown out. In Stonebraker's times, process structure was hierarchical, with a parent process creating a number of children. He criticized that context switching were so expensive and could cost a thousand instructions or more. Modern Solaris has made these facilities cheaper through multithreading mechanism with two-level thread model. Threading offers several advantages. The most salient advantage is that it provides applications a means of getting high degrees of parallelism on multiprocessor systems without having to use the traditional *fork/exec* multiprocess execution model. Simply put, the system can create, manage, and schedule threads within a process with much less overhead and fewer kernel resources consumed than an equivalent number of processes. To give out a direct image how process compares to thread, I have two test programs: *p.c* and *t.c* for processes and threads, respectively. (source code is appended). In *p.c*, *main()* creates n

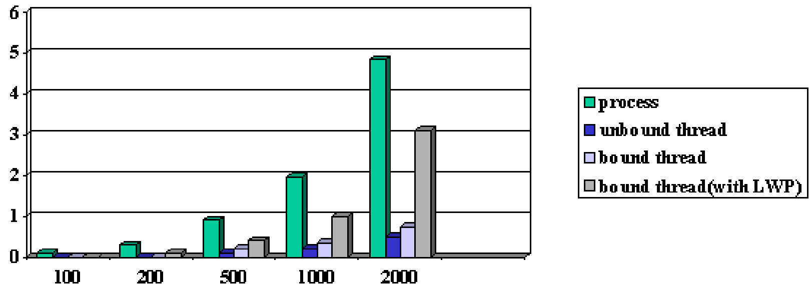

Figure 2: Light-weighted thread and Heavy-weighted process

processes in a loop and each child process execute a null procedure. In *t.c*, *main()* does the same loop, only it creates the same number of threads instead of processes. Each test program was execution on a single processor, and the results were averaged across multiple repetitions, using */usr/bin/time* to calculate the time consumed. The results are shown in Fig.2.

In Fig.2, X-axis is the number of processes or threads created and Y-axis is the time(sec) consumed. Four columns are *process*, *unbound thread*, *bound thread*(without creating new LWP) and *bound thread*(with creating new LWP). For a user thread, it can be created by *thr_thread()* system call with different options. It can be created as an unbound thread with the least overhead, it also can be created as a bound thread, permanently bound to a LWP and to contend among system-wide resources. Or it can be created as a bound thread, at the same time creating a new LWP to be added to the pool of system LWPs. We conclude that with the same number of *fork()/thr_thread()*, time for process creation is greater than time for thread creation. Though bound thread creation with new LWP creation will have more overhead, it is still less than process creation and most important, such implementation can achieve the goal of having literally thousands of threads and not imposing any significant load on the kernel. For a database application, such light-weighted threads can improve performance dramatically in case of a buffer pool miss where context switching will not be avoided.

4.2.2 Scheduling classes

In the early 1980s, when Stonebraker wrote the paper, processes were scheduled by just round-robin policy and no priority mechanism at all. Thus existed the critical section problem mentioned in his paper: "a user-level thread could be executing in a critical section at the instant when it is blocked or preempted." There are two possible ill effects: poor performance (e.g., because other threads continue to test an application-level spin-lock held by the pre-empted thread), and deadlock (e.g., the preempted thread could be holding the ready list lock; if so, deadlock would occur if the upcall attempted to place the preempted thread onto the ready list). Prevention and recovery are two approaches to dealing with this problem. In his paper, Stonebraker suggested that one ultimate solution to task-switch overhead might be for an operating system to create a special scheduling class for the DBMS and other 'favored' users. Processes in this class would never be forcibly descheduled but might voluntarily relinquish the CPU at appropriate intervals. In essence, this is a kind of prevention solution and Solaris has implemented such kind of scheduling class.

The process priority scheme and scheduling have been completely redesigned from prior implementations in Solaris. The original UNIX model provided a timesharing scheduling policy that essentially applied a round-robin approach to scheduling processes on processors so that there was a fairly even distribution of CPU time for all processes. The SVR4 scheduler introduced the notion of the scheduling class, defining the policies and algorithms applied to a process or thread as belonging to a particular scheduling class, based on values established in a dispatch table. For each scheduling class, there exists a table of values and parameters the dispatcher code uses for selecting a thread to run on a processor, in addition to setting priorities based on wait time and how recently the thread had execution time on a processor. Solaris 2.X originally shipped with three scheduling classes by default: *Systems (SYS)*, *Timesharing(TS)*, and *Real-time (RT)*. Somewhere around the Solaris 2.4 timeframe, the *Interactive (IA)* scheduling class was added to provide snappier interactive desktop performance. Each thread or process is given a priority within a scheduling class. Scheduling decisions are made based on global priority. Maintenance and administration command *dispadmin()* displays or changes process scheduler parameters while the system is running. Thread can dynamically change its scheduling class and priority through *priocntl(1)* or *priocntl(2)*.

With scheduling classes, the Solaris scheduler adds several features that enhance the overall usability and flexibility of the operating system, among them are:

- **The intuitive priority scheme:** Higher priorities are better priorities (the traditional Unix implementation was designed such that lower priorities were better priorities). The priorities are assigned as follows:
  - 000 to 059 TS & IA
  - 060 to 099 SYS
  - 100 to 159 RT
  - 160 to 169 interrupt thread priorities (100 to 109 if a realtime scheduler isn't loaded).
- **Real-ime support:** Solaris now provides support for real-time applications, which require a predictable, bounded dispatch latency (the amount of time between when a thread needs to run and when it actually begins executing on a processor).
- **Table-driven scheduler parameters:** This provides the ability to "tune" the dispatcher for specific application requirements by altering the values in the dispatch table for a particular scheduling class. Note that altering the dispatch table values can dramatically affect the behavior of an application, sometimes for the worse if the user and/or administrator doesn't understand well what the values mean.
- **Object-like implementation with loadable module support:** Additional scheduling classes can be created and added to the system without the need for a complete kernel build (e.g., a batch scheduling class was created by Sun Customer Engineering for specific use).
- **Priority inversion:** The issue of a higher priority thread being blocked from execution because a lower priority thread is holding a resource (e.g., lock) needed to run is described as priority inversion. The Solaris dispatcher addresses this problem through priority inheritance. The class init routine is called at boot time for all the preloaded scheduling classes. By default, the timeshare, system, and interactive classes are loaded. Only the real-time class is not. I did a small test to confirm this new characteristic. After system booting up, use dispadmin to show:

  ```
  > dispadmin -l
  CONFIGURED CLASSES
  ==================

  SYS     (System Class)
  TS      (Time Sharing)
  IA      (Interactive)
  >
  Now I dynamically change a process(process PID 833, my test program, sleeps for a minute
  then exits) from TS class to RT class.

  > priocntl -s -c RT -i pid 833

  Now take another look at which scheduling classes are loaded on the system:
  > priocntl -Lc 833
  ```

```
F   S   UID   PID  PPID   CLS PRI ADDR    SZ  WCHAN  TTY         TIME CMD
000 S  1221   833  8847   RT  159   -    5419 do_sel ?           0:01 /u/y/a/yangli/a.out

> dispadmin -l
CONFIGURED CLASSES
==================

SYS     (System Class)
TS      (Time Sharing)
IA      (Interactive)
RT      (Real Time)
>
```

4.2.3 Scheduler Activation

Scheduling class in Solaris provides a solution to the critical section problem, which is a kind of prevention solution to this problem; while scheduler activation provide the mechanism of recovery solution. Scheduler Activation is a fast communication facility between user-level and kernel. It is based on upcall mechanism which is built on Solaris Doors and shared information through shared memory. I will discuss Solaris Doors and shared memory in later paper. When an upcall informs the user-level thread system that a thread has been preempted or unblocked, the thread system checks if the thread was executing in a critical section. (Of course, this check must be made before acquiring any locks). If so, the thread is continued temporarily via a user-level context switch. When the continued thread exits the critical section, it relinquishes control back to the original upcall, again via a user-level context switch. At this point, it is safe to place the user-level thread back on the ready list. The related APIs are schedctl_int, schedctl_lookup, schedctl_exit, schedctl_start and schedctl_stop, which provide limited control on process preemption and thus can support adaptive locks for better priority management.

Scheduler Activation combines the advantages of both LWPs and User Threads without inheriting their drawbacks. To achieve this, the kernel and the user level thread library exchange a lot of information such that each of them can perform those operations it is best able to. The kernel creates a "virtual multiprocessor" for each application or process. That is, for each processor it hands an execution context (called "Scheduler Activation") to the user level scheduling library. In a multiprocessor system there can be many Scheduler Activations assigned to one process. But there might also be more processes than processors in a system, so that the execution context(s) have to be shared among all processes. The kernel is responsible to allocate the Scheduler Activations to the processes, while the processes (i.e. the user level thread libraries) assign their threads to the available execution contexts. The scheduler might ask the kernel for more or less processors. Vice versa, the kernel might notify the scheduler of different events like the blocking and unblocking of a thread, preempting of Scheduler Activations or allocation of a new processor to an application. These notifications use the execution context of a Scheduler Activation to communicate their messages. For example, if a thread unblocks, the kernel preempts another thread of this application and makes an

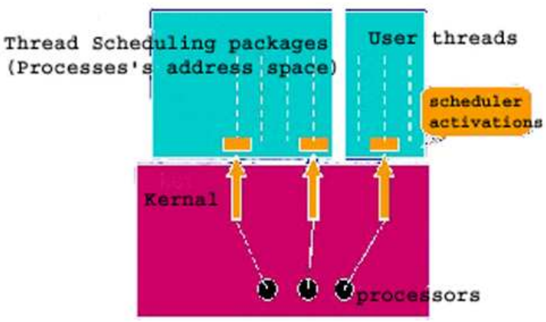

Figure 3: Scheduler Activation

Time(us)	UNIX processes	LWPs	User threads with LWP	Scheduler Activation
Null Fork	11300	948	34	37
Signal-wait	1840	441	37	42

Figure 4: Performance comparison

upcall (calling a predefined method) to the scheduler, handing it a Scheduler Activation with the enclosed message, that two if its threads are able to run (the one that has unblocked and the preempted thread). The thread scheduling facility is informed about low level events and can use its (user lever !) knowledge about priorities to schedule the threads. It does not schedule on a LWP that itself is scheduled independently by the kernel but has complete control over its assigned part of the hardware. This also results in a correct synchronization behaviour. Since none of the thread operations involves the kernel, there is no mode switch required and thread manipulation becomes very fast. Threads and the scheduling library can be much more specific to the application, resulting in less overhead for general purpose functions or data structures. Fig.3 shows the system architecture with Scheduler Activation.

Some test work has been done and here I cite in Fig. 4 the test results directly [2]. The two benchmarks are Null Fork, the time to create, schedule, execute and complete a process/thread that invokes the null procedure (in other words, the overhead of forking a thread), and *Signal-Wait*, the time for a process/thread to signal a waiting process/thread, and then wait on a condition (the overhead of synchronizing two threads together). Each benchmark was execution on a single processor, and the results were averaged across multiple repetitions. For comparison, a procedure call takes about 7 usec.

The results show that using Scheduler Activations is faster than using either LWPs or user threads, no matter which of the latter ones performs better.

4.2.4 Processor Sets and Solaris Domain

Processor sets, internally, are referred to as CPU partitions. Some number of processors can be configured into a set for resource management and application binding. Processes and kernel threads can be bound to a processor set. For a critical-mission DBMS application, we can create a processor set and bind it to the set, thus other activities on the server do not impact the performance of the DBMS application. Solaris 2.6 added processor sets and thus comes a method of binding processes to CPUs. Command *psrset* can be used to control the management of processor sets. Once a processor set is created by command: *psrset -c cpu_id*, the processors in the set are no longer available for running all kernel threads systemwide. Rather they will only run kernel threads that have been bound to the processor set with *psrset(1M)* or *pset_bind(2)*. Once a process is bound, the binding is exclusive, which means only threads bound to the processor set are scheduled and executed on CPUs in the set.

Domains on high-end systems like the Sun Enterprise 10000(Starfire) server allow groups of processors, memory, and I/O devices to be configured as physically separate servers, delivering the superior reliability, availability, and serviceability of the Starfire server while allowing resources to be configured as needed to meet application demands. An additional benefit of Solaris domains is the ability to reconfigure resources as needed in response to an un-predicted faiure, or in response to scheduled changes in demand. For example, processors and memory can be easily re-allocated at night from lightly-loaded on-line transaction processing applications and moved to support long-running batch queries in a data warehousing domain.

4.3 Conclusion

Multithreading is popular in modern OSs. Using threads eliminates the need for any IPC and reduces context switching overhead. Threads can be scheduled concurrently on multiple processors.

Taking advice from upper level applications are implemented in some OS such as solaris and some research prototypes, such as scheduler activation.

5 Consistency Control

The problem addressed in the paper is that although most operating systems provide locking for files, there are fewer which support finer granularity locks, such as those on pages or records. Therefore, a desirable service provided by OS is to provide finer granularity locks. Also, the services of buffer management, concurrency control, and crash recovery should be either all provided by OS, or by DBMS in user space.

5.1 NT support

5.1.1 Write consistency

NT allows one to create a file whose pages will never be cached when opened. A flag can be passed into one of the file create or open routines that tells the OS not to do any buffering. This means that a read or write will always result in a disk read or write and leaves caching up to the processes that use the file. When combined with synchronous I/O, I/Os complete before giving control back to the calling process, ensuring write consistency.

5.1.2 Synchronization primitives

NT does provide fine grain synchronization primitives. However, if the database product is doing its own thread management, it typically implements its own synchronization mechanisms. In SQL Server, all of the functionality in such as locking, buffering, and recovery are provided by the DBMS in user space.

5.2 Solaris support

5.2.1 Memory locking

Solaris supports finer granularity locks on pages.In a virtual memory system, any memory reference can potentially cause a page fault.The time required to bring the referenced data into physical memory from disk can cause a real-time application to lose its determinism.The Solaris Operating Enviroment addressesses this problem by allowing the locking of process pages into memory using *mlock()* or *mlockall()*. Locked translations can be unlocked with *munlock()*. Liking placing an application in the real-time scheduling class, root access is required to lock pages into memory.

5.2.2 Shared Memory

Shared memory is the fastest form of IPC available. Once the memory is mapped into the address space of the processes that are sharing the memory region, no kernel involvement occurs in passing data between the processes. What is normally required, however, is some form of synchronization between the processes that are storing and fetching information to and from the shared memory region. Consider the normal steps involved in the client-server file copying program that we used as an example for the various types of message passing, shown in Fig.5.

total of four copies of the data are normally required. Additionally, these four copies are done between the kernel and a process, often an expensive copy(more expensive than copying data within the kernel, or copying data within a single process). The problem with these forms of IPC is that two processes to exchange information, the information

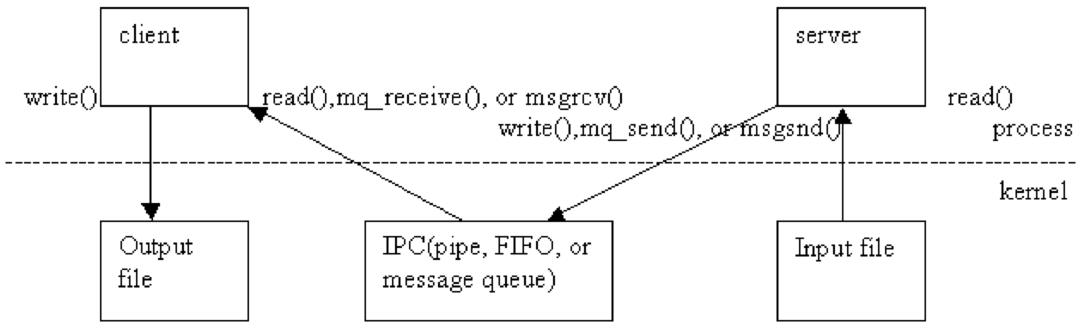

Figure 5: Flow of file data from server to client

has to go through the kernel. Shared memory provides a way around this by letting two or more processes share a region of memory. The scenario is depicted in Fig.6.

In Fig.6 the data is copied only twice: from the input file into shared memory and from shared memory to the output file. Related APIs are *shmget()*, *shmat()*, *shmdt()* and *shmctl()*. I wrote a program *sharedmem.c* to use *shmget()* creating a shared memory segment. And then use ipcs command to confirm that share memory is activated:

```
>ipcs
 IPC status from <running system> as of Mon Dec 18 09:41:39 CST 2000
 Message Queue facility inactive.
 Shared Memory facility inactive.
 Semaphore facility inactive.
>gcc sharedmem sharedmem.c
>sharedmem
>ipcs
 IPC status from <running system> as of Mon Dec 18 09:58:14 CST 2000
 Message Queue facility inactive.
 T         ID      KEY            MODE        OWNER    GROUP
 Shared Memory:
 m    2  0xaf --rw-rw-rw-      yangli    1221
 Semaphore facility inactive.
>
```

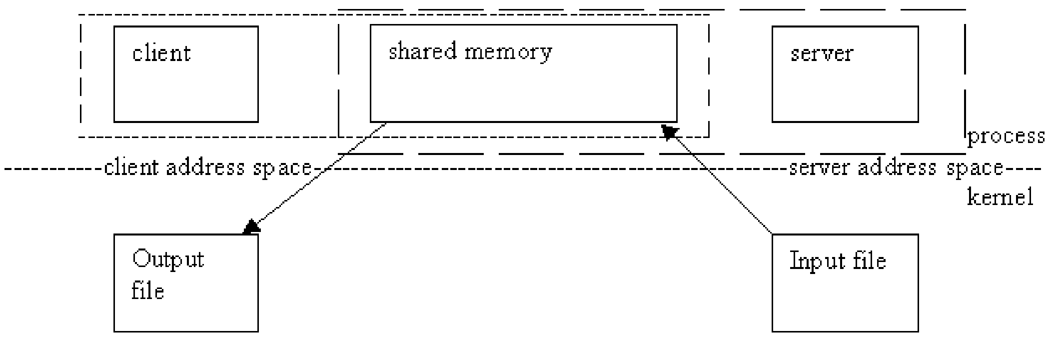

Figure 6: Copying file data from server to client using shared memroy

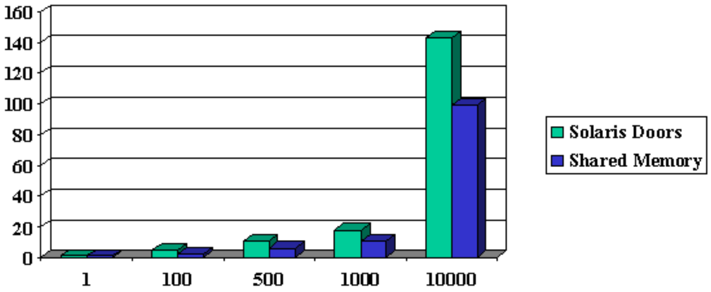

Figure 7: Solaris Doors vs Shared Memory

5.2.3 Solaris Doors

Doors are a very recent addition to Solaris. Simply put, doors are a new IPC (interprocess communication) mechanism that evolved out of an operating system known as Spring, or SpringOS. SpringOS was a research project out of Sun Labs. Doors provide a means of doing RPC-like calls across processes running on the same system. A thread running in a process can issue a door call that results in a thread in another process executing a routine on behalf of the calling process. A return is done to the caller with appropriate status information. Processes export routines by using the *door_create*(3X) routine. Other processes may then issue *door_call*(3X) calls to make a call to the exported routine. For interprocess communication that does not involve passing data, doors have proven to be much faster than traditional IPC mechanisms (e.g. semaphores, pipes, etc.). Available APIs are *door_call()*, *door_create()*, *door_info()*, *door_return()*, *door_revoke()*. To show quantitatively how Solaris door works and how it compares to other IPC mechanisms, I conducted two tests(source codes are appended.) One test conducts an IPC involving no data passing, using Solaris Doors and Shared Memory separately. The results turn out that the time consumed by both are very small, just a few clock ticks calculated by *times*(3B) utility and show 'Real time: 0.0' as the result of using */usr/bin/time*. Another test conducts an IPC involving a large number of data passing. I wrote two programs, one is *d.c* and the other is *s.c*. Program *d.c* uses Solaris Doors and let child process call the door in a loop. Program *s.c* uses shared memory and let child process do the same thing as what *d.c* achieves (pass a large number of data to the parent process). The results were averaged across multiple repetitions and showed in the following chart.

In Fig.7 x-axis is the number of child process communication with parent process and y-axis is the time consumed(unit: clock ticks). Two columns are separately Solaris Doors and shared memory. We can see that both Solaris Doors and shared memory are the fast and efficient forms of IPC. With the same workload of data passing, shared memory is faster than Solaris Doors.

5.3 Conclusion

Modern OSs provide finer grain synchronization primitives and synchronize I/O support. However, usually DBMSs usually implement all the services of buffering, scheduling, concurrency control and crash recovery in user space. There are two possible reasons for this:

- DBMS has to do so. A review of the issues involved can be found in the computing surveys article on the "Aries" transaction recovery system (to be found). The argument is that you cannot move all that stuff into the OS without the OS becoming the DBMS
- DBMS has already done so. DBMSs are very mature applications. The DB people have implemented their own versions of all of these, which took many efforts. In addition, in a product development, only ten percent of all the effort goes to development, while almost eighty percent goes to testing and maintenance. Taking this into consideration, there is no significant reason for them to switch.

Acknowledgments

We would like to thank the people from Microsoft Database Research Group, and ???? from Sun Microsystem, for answering our emails in time.

References

[1] M. Stonebraker, "Operating System Support for Database management," *Communications of the ACM*, Vol. 24, No. 7, July 1981, pp. 412-418.

[2] T. Anderson, B. Bershad, E. Lazowska, H. Levy, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism," *Proceddings of teh Thirteenth ACM Symposium on Operating System Principles*, October 1991, 95-109

[3] P. Cao, E. Felten, K. Li, "Application-Controlled File Caching Policies," *Proceedings of the 1994 Winter Usenix Conference*, June 1994, pp. 171-182.

[4] M. Seltzer, Y. Endo, C. Small, K. A. Smith, "An Introduction to the Architecture of the VINO Kernel," *TR-34-94*, Harvard University, Decemember 1994

[5] J. Mauro, R. McDougall, "Solaris Internals – Core Kernel Architecture," *Sun Microsystems Press*, October 2000

[6] "http://www.solarisinternals.com," "http://docs.sun.com," Sun Microsystem white paper

[7] D. A. Solomon, "Inside Windows NT," *Microsoft Press*, May 1998

[8] "http://research.microsoft.com/db/," Microsoft Database Research Group