Operating System Supports for Database System Revisit
-- CS736 project report

Junfeng Zhang Demai Ni

Abstract

Operating systems and Database systems are two major research and commercial areas. Due to the different targets of services, database systems once complained about poor operating system supports for database system [Ston81]. What is the situation today? We choose Sun Solaris as an example of modern operating system, examining its various components. We also go over PostgreSQL as the example of database system. Our conclusion is, modern operating systems have provided lots facilities to help database systems, and database systems are taking advantage of them.

1. Introduction:

Operating Systems and Database Management Systems are two areas that absorb much attention in computer science and software industry. An operating system is identified as a generic resource manager, while a database management system uses the services provided by operating system to manager useful and important data. Due to the generic nature of Operating system, database systems once complained about operating systems' "Almost, but not exact" services. Stonebraker criticized almost every aspect of operating system [Stone81]. Almost two decade has passed since Stonebraker's criticism. Both operating systems and database systems evolve. What is the situation today?

In the first part of the paper, we briefly present Stonebraker's criticism. Next Sun Solaris is chosen as an example of modern operating system. Various components are examined and evaluated from database systems' view. Some tests are taken for detailed examination. After that we also choose PostgreSQL as an example of database system, watching how it is using operating system's services. Finally we present our summary and conclusion.

2. Background:

In 1981 Michael Stonebraker wrote a paper complaining operating system's poor support for database management system, based on Unix system and INGRES relational database system [Ston81]. He examined several operating system services, including buffer pool management, file system, scheduling, process management and interprocess communication, and consistency control. His criticism includes:

- Expensive overhead for buffer pool access. LRU page replacement algorithm does not work well for all cases. Prefetching does not applied to database systems' pseudo-random access pattern. And no selective force out services.
- Data blocks are not physically contiguous. And tree structured file system adds overhead to data access.

- Interprocess communication is poor. Task switch is expensive. Semaphore may cause convoy problem. Message passing has high cost.
- Poor support for concurrent access on file data. And crash recovery is not satisfactory.

With all the problems he addressed, Stonebraker concluded that operating systems failed to provide appropriate support for database systems, although they could have. He proposed a small efficient operating system with only desired services.

This paper gives a good overview of operating systems support to database systems. Since then almost twenty years has passed. Both operating systems and database systems evolve. What modern operating systems look like? How well do they support database systems? In the next section we will use Sun Solaris as an example, examining in detail with the services listed by Stonebraker, see how its services support database systems.

3. SunOS (Solaris)

SunOS is a UNIX system produced by Sun Microsystems. It is by far the most popular UNIX system. Its first version used BSD-UNIX interface, and changed to System V interface in later version [Vaha96]. Its current version is called Solaris. Solaris 7 conforms to POSIX.2, SUSv2, and lots of other standards and specifications. In the following section we will look at its file system, buffer cache management, process scheduling and interprocess management.

3.1 File Systems

Solaris's file system adopts a variation of BSD fast file system [Mcku84], which is called UFS (Unix file system)[Bert98]. UFS is described by the *superblock*, which contains the file system layout information. UFS divides a disk partition into one or more cylinder groups, each containing a set of consecutive cylinders. Each cylinder group keeps a copy of the superblock and summary information about that group, including an *inode bitmap*, some *inodes*, and *free block lists*. Each disk block can be further divided into more sub-blocks called *fragments*. A unique *inode* is allocated for each file and directory. The inode is used to describe the file's layout on disk. It consists of some control data for that file, followed by twelve pointers for the first twelve blocks of the file, one pointer for *indirect block*, one pointer for *double indirect block*, one pointer for *triple indirect block*. The UFS has sophisticate file block allocation policy. It uses the knowledge of the disk layout to place the data from the same file close to each other. To avoid filling an entire cylinder group with one large file, it changes the cylinder group when the file size reaches 48kilobytes and again at every megabytes. This is to ensure that small files can be located close to their corresponding parent directory, which limits the seek time required to reach them. UFS doesn't perform pre-allocation. Blocks are added to the file when a write occurs, and not before. The UFS is inherently a 32-bit package, but now it supports a 64-bit file size limit. Disk addresses are limited to 32 bits, each addressing 512 bytes of data (typical disk sector size), for a total of one terabyte for a file system. Users are limited to 2 gigabytes per file before Solaris 2.6, and one terabytes after. Find the file information from a large directory can be slow. UFS uses directory name lookup cache to fast this operation. UFS implements a file system clustering to improve the read and write performance [Vaha96]. A parameter *maxcontig* describe the size of a cluster. When a read request requires a disk access, the system will load the contiguous blocks of the file starting at the specific block. Actual write to disk is performed when a full cluster is in the cache, or sequential write pattern is broken. This technique improves the sequential read and write.

Advisory and mandatory locks on files are included in Solaris. UFS supports shared and exclusive locks. File locks can be on record granularity (bytes range with files). The system call *fcntl()* provides the locking functions[Stev92][Programmer].

In the case that the system does not shutdown properly, a utility program called *fsck* is used to maintain file system consistency[Vaha96]. This program examines the whole file system, looks for inconsistencies, and repairs them if possible. The result of *fsck* is a limited form of crash recovery—it returns the file system to a consistent state, not recovering all the modification before the system crash.

UFS is designed on a Unix server-based approach. The typical workload is edit-compile-link cycle of software development. This workload will have lots of small files, and those files are accessed as a group. In view of this access pattern, the UFS adopts a design that favors small files, and relatively bad support for large files. Database systems usually process a huge repository of data. This obviously leads to some discrepancy.

There are several issues why UFS gives a bad support for large files[Bert98]. The first one is non-preallocation. Disk blocks are allocated at request, and one block each time. This fragments large files. Although the disk allocation algorithm tries to put data block in one file close to each other, it changes cylinder groups after each one megebytes to avoid flooding a cylinder group by one file. This further separates the large file data. The last issue is about the indirect pointer in inode. To ensure proper crash recovery, indirect pointer has to be initialized. This requirement holds recursively for double indirect pointer and triple indirect pointer. This decreases write performance.

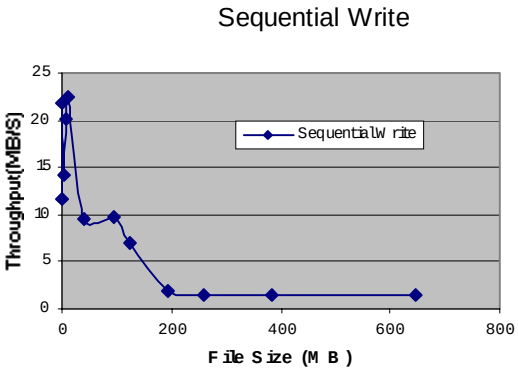

Figure 1 write throughput v.s. File Size

Figure 1 shows a test[1] on write throughtput for large files. In this test data are written into the file sequentially from the beginning to the end. As we can see from the figure, as the file sizes increases, the througput decreases dramatically. We see two steps in the figure. Those are likely the indirect pointer and double indirect pointer.

The record granularity of locks on files gives database systems more flexibility on concurrent access to file data. With *fsck*, the file system will return to a consistent state. But this does not satisfy the ACID requirement from database system.

Many people observe this problem, and come up with some solution. Log-structured file system (LFS)[ROSE91] bundles writes and sequentially write modification to disks. Dr. Seltzer gave a thorough discussion on UFS and LFS performance and transaction support[SELT93]. Veritas file system uses a extent-based disk allocation algorithm, and does metadata logging [VERT]. Veritas file system are available for high-end Sun Servers.

3.2 Buffer Cache Management

[1] All succeeding tests are done on CS lab nova machines, which has 640MB memory

To reduce disk traffic and eliminate unnecessary disk I/O, Solaris uses memory as the *buffer cache* to cache the recently access file blocks[Bert98]. Buffer cache is integrated with the virtual memory paging system, thus can grow at time. If a read request hits in the buffer cache, the buffer is returned to the request without actually accessing the disk. The operating system will prefetch several disk blocks into buffer cache if it detects sequential access. To prevent saturation of virtual memory system, UFS implements a free-behind policy when reading large sequential files. A block is replaced when no free buffer can be found for a request. The replacement algorithm is called *global clock-hand LRU replacement scheme*. Memory is treated as a common pool, with little per-process or per-file information used by the paging routines. The replacement process scans memory pages, first turning off reference bits, and checking them later to see which pages have been referenced. The scanning is done by a thread, called the *pageout_scanner*, which maintains two pointers, called clock hands, into a list of the system's pages. One pointer indicates the next page for which the reference bit is to be reset, and the other indicates which page is to be checked for replacement. The two clock hands are advanced in unison. If the scan finds a page that has not been referenced since the reference bit was reset, and it is not locked into memory, that page is placed on a list, and the *pageout* thread frees it after doing any necessary disk operations. There is a minimal check done to ensure that at least a few pages per process are never freed by this scanning. File write is write-behind, which means modified data will be written to disk only at its replacement. To maintain the consistency of the file system in case of crash, metadata writes are synchronized.

A potential problem for integrating file buffer cache with virtual memory system is that a large file might force out some program segments, resulting in a severe penalty. UFS uses a *priority paging* [Prio98] algorithm to address this problem. It will flush some file buffer when the free memory drops to some threshold. This technique achieves a 10-300% performance gains on different workloads [Prio98].

Caching data in kernel causes extra overhead for data access. It is advantageous only if the data will be requested again later. If data will be only accessed once, or data set is too big to fit into the memory, caching in kernel buffer is actually harmful. Database systems' data sets are usually much bigger than the memory size. And database systems usually have their own buffer for the data block they accessed. Thus kernel buffer caching is not useful for database systems. Database systems have a limited number of data access patterns – either scan the file or access the record through an index. Chou etc. classified the database access into nine categories[Chou85]. For each category a work set is chosen and a replacement algorithm is applied. For those access patterns, LRU is not necessary the best algorithm for all the cases.

Another issue related to kernel caching is database system logging. Database system uses logging to enforce a transaction model. Those log records have to be written to disk before the write can finish. This effectively disables write-behind. Those records will not be read again unless a recovery is processing. At any case, caching of log records will not be useful.

Buffer replacement policy has been a research area for database systems because it is critical to their performance, and is possible because of the limited number of data access patterns. While it is not touched very much by operating system research because of the un-predictability of memory access in operating systems. Reiter etc. proposed the domain separation algorithm[Reit76]. Chou etc. presented the DBMIN algorithm [Chou85]. Carey etc introduced Priority-LRU and Priority-DBMIN policies[Care89]. Nicola etc. analyzed the Generalized Clock Buffer replacement scheme[Nico92]. O'Neil etc proposed LRU-k algorithm[Onei93]. Johnson etc. gave the "Two Queue" algorithm[John94].

3.3 UFS I/O interfaces

To address the problems presented above, and provide support for vaious programs, UFS exports a wide range of interfaces for I/O operations [Vmsizing] [Bert98] [Programmer]. These include:

- Asynchronous I/O: aioread/aiowrite/aiowait
  aioread/aiowrite sends a disk request to the operating system, and returns immediately. aiowait can be used to sychronously detect a completion of asynchronous request. The completion can also be notified by a signal SIGIO
- Synchronous I/O: sync/fsync/fdatasync, O_DSYNC|O_RSYNC|O_SYNC
  fsync will force the modified disk metadata/data written to disk before it returns. fdatasync won't force out the metadata. sync will schedule a flush of all dirty pages, but will return before the flush is actualy done.
- Memory-Mapped files: mmap/madvise/memcntl/msync
  mmap maps a file to the user process's memory space. Missed file blocks are handled through virtual memory page fault mechanism. This eliminates the kernel buffer cache for the file blocks. But mmap cannot be used to files with sizes greater than the virtual memory system.
- Vector I/O: readv/writev
  readv/writev will read from/write to a contiguous range in the file into/from several non-contiguous buffers. Each read/write still need a disk operation. But it only does one metadata update.
- Direct I/O: directio
  With directio on, operating system won't perform any buffer caching for file block accessed. It cannot be used to mmaped files and files with holes. Reads/writes have to be sector aligned(512 byte). No prefetching is performed by the system.
- Raw device: /dev/rdsk
  Raw device allows a program to directly access the storage devices just like directio, but bypasses the file system interfaces.

Database systems can use those interfaces for different data access/operation pattern. Asynchronous I/O interfaces can be used when database system need to do some work on current fetched data, and start fetching data for next operation. Synchronous I/O interfaces can be used to ensure the data is written to the disk, like in the logging case. To bypass the kernel buffer caching, database systems can use mmap, directio and raw device interfaces. Raw device interfaces eliminates the file system overhead, thus achieves the close to hardware performance. mmap is good for small read-only files. For large file, mmap disables the priority paging algorithm, and causes thrashing on virtual memory system.

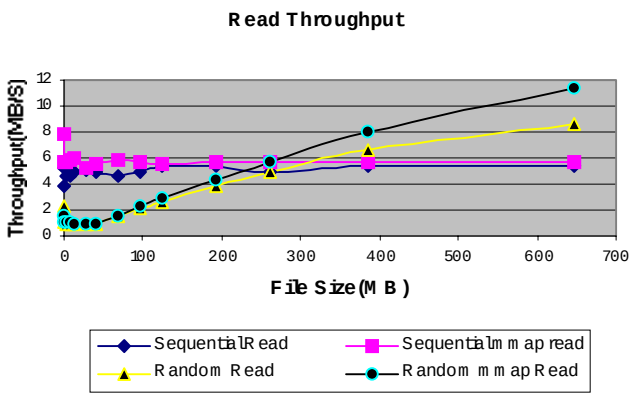

Figure 2 Read Throughput v.s. File Size

Some tests are performed to study different I/O interfaces provided by Solaris.

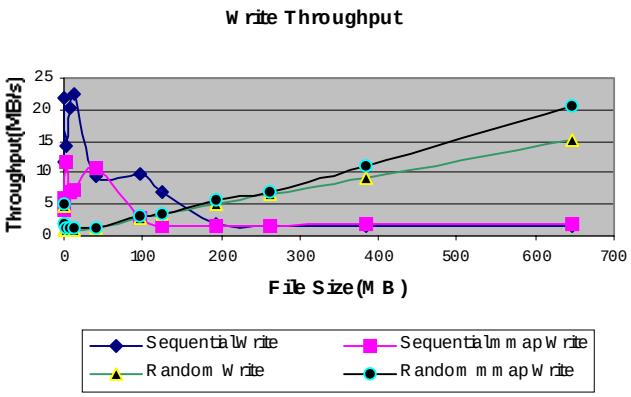

Figure 2 Write Throughput v.s. File Size

The first test we took is comparing mmap and read/write with sequential and random access pattern on various file sizes. We can see from figure 2 that for sequential reads access, mmap beats read() for all cases. This win should come from the elimination of kernel buffer access. For large files, mmap beats read() marginally. This comes from two points. The first is large files read is dominated by disk activities. Kernel buffer access overhead becomes less important. The second is the benefit of clustering for read. For sequential writes, write() outperforms mmap for small files. This must relates to write clustering, and the extra virtual memory overhead for mmap like page table lookup, page fault handling etc. mmap interface has another drawback that it may replace program segments because priority paging does not apply to it. For random access, all interfaces start with a small throughput, and increases linearly as the file size grows after a threshold about 50megabytes. This must relate to the clustering and kernel buffer. mmap does a little better than read/write in these caese. The write throughput is higher than read throughput in our study because we write the file twice. The second write on the file will write on the kernel buffer, thus increases the throughput. From our result we can conclude: mmap is better than read() for small file reads access. write() outperforms mmap for file writes. Caching wins for random accesses.

We also did some test on asynchronous interface, compared with standard read/write interface. In this test we read a block to memory, sort it, and write it back to its original place.

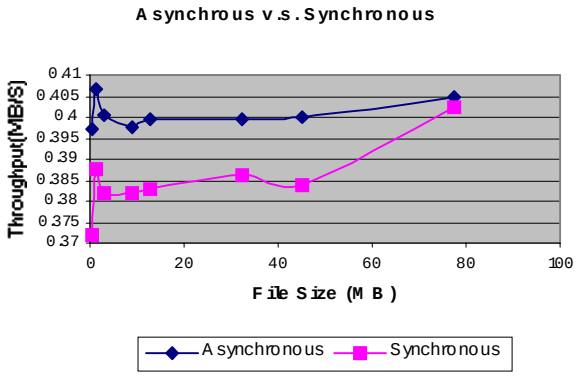

Figure 3 Asynchronous v.s. Synchronous

At all file sizes, asynchronous interface beats synchronous interface. For this test, we used aiowait to synchronously wait for the completion of the I/O operation. If asynchronous wait is used, we will expect a higher gain for asynchronous interface for handling operations on disk data.

3.4 Summary on UFS, its buffer cache, and its I/O interfaces

We examined UFS layout, its buffer cache, and its I/O interfaces. UFS fails to provide a good support for large file, and its default buffer cache behaviour does not fit into database systems' requirement. UFS provides a variety of I/O interfaces to allow database systems to bypass those problems. Some of them are effective.

XFS[XFS] achieves a throughput of >4GB/s using directio and extent-based allocation. It also supports guranteed rate I/O, and lots of advanced feature like journaling, 64-bits. It is available on IRIX system.

3.5 Processes management in Solaris

Solaris provides the traditional processes management. It also supports threads[Eykh92].

Solaris supports three different kinds of threads: *kernel threads*, *lightweight processes* and *user threads*. The kernel thread is a fundamental lightweight object that can be independently scheduled and dispatched to run on one of the system processors. It need not be associated with any process and may be created, run and destroyed by the kernel to execute specific functions. Lightweight processes provide multiple threads of control within a single process. They are kernel-supported user thread. Each LWP is bound to its own kernel thread, and the binding remains effective throughout its lifetime. LWPs are scheduled independently and may execute in parallel on multiprocessors. User threads are implemented by the *threads library*. They can be created, destroyed, and managed without involving the kernel. The threads library provides synchronization and scheduling facilities.

Applications using multithreads can achieve the same parallelism as applications using multiple processes, with much less overhead. Switch between threads is much cheaper than processes context switch since it does not need to change the virtual address. Creation/termination of a process is more expensive than creating/deleting a thread. Communication and synchronization between processes have a higher cost than those between threads. But multiprocess applications have their advantages over multithread applications. Each process has its own address space. Process does not have to protect its data explicitly from other processes. While in a multithreaded process, those threads share the same address space. Every object in this address space has to be protected since more than one thread can access this object. Some forms of synchronization have to be provided to avoid the data corruption. If the number of threads goes to a higher number, this may become a problem.

A simple test is taken on Solaris 7 to examine multithread throughput. The test is similar to the asynchronous one. The program reads a block from a file, sorts it, and writes it back to its original places. Multithread version will make each thread working on a part of the file. Figure 4 shows the results.

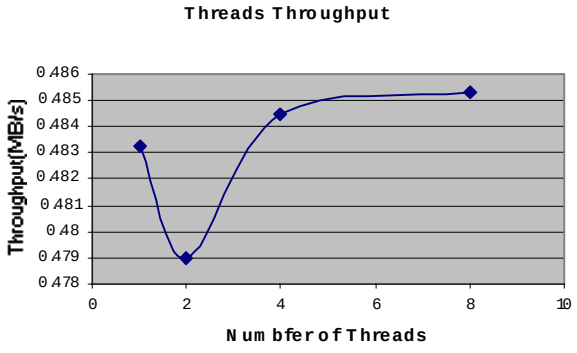

Figure 4 Multithreads throughput

From figure 4 we can see as the number of threads increases, the throughput increases/decreases a little bit. But basically they are about the same. Seems this means multithreads on a single task single processor will not increase the throughput very much. Better use threads for hetero-services and use them on multiprocessor systems.

3.6 Processes Scheduling in Solaris

Solaris implements a fully preemptive system[Khan92]. User applications as well as kernel activities are all preemptible. Processes(threads) are divided into two categories: The *time-sharing* class and the *real-time* class. The real-time class has higher priorities than any time-sharing process. Only superuser processes can enter the real-time class. Real-time applications have fixed priority and time quantum. The only way to change them is by making a *priocntl* system call explicitly. The time-sharing class is the default class for a process. It changes process priorities dynamically and use *round-robin* scheduling for processes with the same priority. The time slices given to a process depends on its scheduling priority. By default the lower priority of the process, the large its time slice. The time-sharing class uses event-driven scheduling. Operating system changes the priority of a process in response to specific events related to that process. The scheduler penalizes (reduces its priority) the process each time it uses up its time slice. On the other hand, the scheduler boosts the priority of the process if it blocks on an event or resource, or if it takes a long time to use up its quantum.

A *priority inversion* problem might occur if a lower-priority thread holds a resource needed by a higher priority process, thus blocks that higher priority process. Solaris uses a technique called *priority inheritance* to address it. When a high-priority thread blocks on a resource, it temporarily transfer its priority to the lower-priority thread that owns the resource. When the lower-priority thread release the object, it surrenders its inherited priority. This will not eliminate the priority inversion problem. But it allows the blocked process to restart quickly.

With a fully preemptive kernel and priority inheritance, Solaris is able to make the processes dispatch faster. In Khanna's originally measure[Khan92], the average dispatch latency is 2ms. This number should have decreased many times with modern hardware.

3.7 Interprocess Communication

Solaris provides a large numbers of primitives for interprocess communication and synchronization[Vaha96][Programmer]. Traditional IPC primitives like *signals, pipes*, and *semaphores* are fully supported, with the addition of lots of new facilities. *Message queue*(msgget/msgsnd/msgrcv) provides a way for multiprocess programs to exchange message through shared message queue. *Shared memory* allows different processes access the same physical memory region. The system call shmctl() can be used to pin the shared memory to RAM. This is particularly useful for database systems because when they implement buffer pool with shared memory, they need the knowledge that the buffer pool IS residing in the memory.

For short-term access to shared data, using semaphore is not acceptable because if a process holding a hot lock is being swapped out of the processor, all the rest processes have to wait until the process holding the lock back to processor again, thus causing a convoy problem. To address this problem, light-weighted locks are provided. *Atomic test-and-set* primitive is supported, and other locks are built on top of it. *Spin-locks* do a busy wait on an allocated lock. This can be used for short-term locks. Solaris implements the *adaptive locks*. When a thread want to access an allocated lock, if the owner of the lock is in sleep, this thread goes to sleep, otherwise it does a spin. *Read-writer locks* synchronize read and modification of shared data, while *condition variable* provides an event-driven synchronization.

3.8 Summary of Solaris's processes management and IPC

Solaris puts much effort to decrease the dispatch latency. It also supports threads to improve the system throughput and multitasking. Ample IPC and synchronization primitives are provided to facilitate the processes communication.

We have examined the Solaris system for its file system, its buffer cache, its processes management, and its interprocess communication primitives. Now look at the other side of the story. How does database systems evolve? Do they take advantage of all the facilities provided by the effort of operating system? We choose PostgreSQL as an example for database system, and look at how it handles client requests, how it manages its transactions, and how it uses the operating system facilities.

4 PostgreSQL

4.1 System Overview

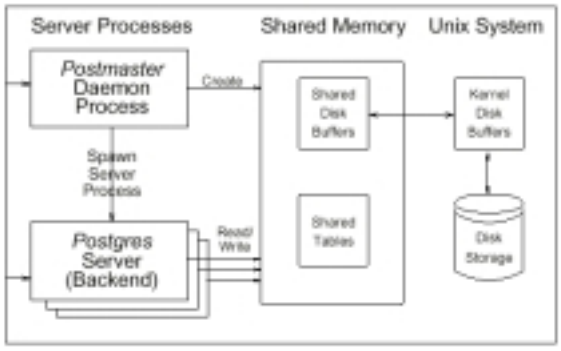

Figure 6 Inter-Server communication in PostgreSQL

PostgreSQL[Momj00] is an open source object-oriented relational database system. It is originally called Postgres[Ston86][Ston90], and was developed in University of California at Berkeley. It moves out to the open source community in 1996, and becomes PostgreSQL. Now it is maintained by about fifteen programmers, and is offering commercial support to some degree. We choose this because it is open source project. We can examine it from the source code level, thus gives us more inside how it works.

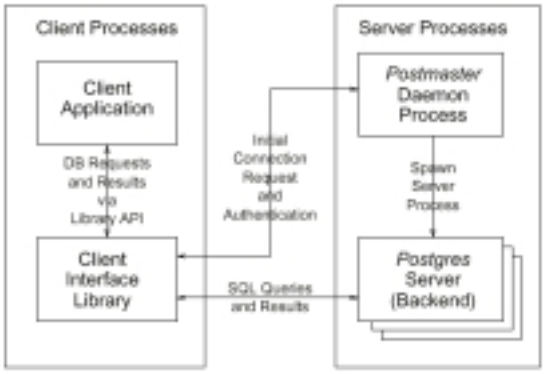

Figure 5 Clients/Server communication in PostgreSQL

PostgreSQL uses the client/server model [Lane00]. Communication between clients and server is through Unix socket. A postmaster daemon is started at the server, and listening to all the client requests. Clients connect to the server at a well-known port number. Once the postmaster receives a valid request, it spawns a new postgres backbend server process to serve that request. PostgreSQL will divide a relation into 2GB chunks if the relation has a size bigger that 2GB for the reason that most operating systems only support user files with 2GB up limit. Read/write relation data are through standard read/write interface. Database data are brought into a shared buffer pool created by shared memory. Unix kernel usually provides additional buffering. Buffer pool is considered as global. LRU algorithm is used for replacement policy. Dirty pages will be flushed to disk only at the replacement.

PostgreSQL supports transactions processing[Lane00][. Logging is used to ensure the ACID properties of transactions. It choose a very different way to implement its logging. Each transaction has a log record, which has only 2 bits, showing the transaction is in progress, aborted, or committed. It implements a non-overwrite storage management. Modifications to tuples are appended to the table. Older tuples will be removed sometime later by a vacuum maintenance command issued periodically. Before a transaction commits, it has to force its modified data page to disk before write the log record. This is done by the fsync() I/O interface. Concurrent accesses to database tables are guaranteed using lock. It implements a multilevel concurrent access control. Readers never wait for the writers. Writers will block each other only when updating the same row. Waiting on locks is handled with per-process IPC semaphore. Locking is a strict two phase locking — Once a transaction releases one locks, it cannot acquire any other locks. Locks can be applied to tables, pages, or records(rows). Short-term locks are used to protect data structure in shared memory. Those locks are implemented using spin-locks based on platform-specific atomic test-and-set instructions.

4.2 Analysis

PostgreSQL tries to work on most Unix/Linux systems. To achieve this it can only use features presented on most system. This generality makes PostgreSQL less efficient at some systems. Specifically, at Solaris system, at least several optimizations can be achieved. First, using threads instead of processes to handle requests. Threads have smaller creation overhead, thus can serve user requests quicker. Current model suffers a long startup time for small transactions. Another advantage of threads model is a thread consumes much less resource than a process. For the same system, it can sustain much more threads than processes, thus can serve much more requests. The reason why PostgreSQL people do not adopt thread model, partly because of rewrite the system in thread model will have too much work. Another reason is those people think there is no standard compatible thread implementation for all the systems they support. A further reason they give is PostgreSQL supports user defined type and operation. With a process model, a malicious user program will crash that particular process only, without crashing the whole system. And this advantage does not apply to thread model. A second optimization will be the shared memory. Locking the shared memory by shmctl() is only available at Solaris, and only the superuser can make this call. Currently PostgreSQL has to be run as a regular user. This effectively disables the “pinning” of shared memory. The third is I/O interface. PostgreSQL uses the standard read/write to read/write from/to a file. As we see from our previous analysis and test, there are other interfaces which can achieve higher throughput. The fourth is the buffer pool. We already know LRU will not work well all the time. Flushing on replacement slows down the buffer allocation. A better way may be using a separate process (thread) to flush the dirty pages to disk on the background. Modification of PostgreSQL on Solaris to adopt those optimizations is an interesting thing to do.

5 Summary and Conclusion

We revisited the operating system support for database system by looking inside a modern operating system—Sun Solaris. We studied its file system, its buffer cache, and its ample I/O interfaces. We further went over its threads support, its processes scheduling and interprocess communication. After that we used PostgreSQL as an exmaple of database system. We checked its client-server and inter-server communication, its transaction support and its storage system. We pointed out several optimizations that PostgreSQL system can take under Solaris to improve its performance.

In conclusion, operating systems do not solve all the complains from database systems by default. But they provide lots of facilities to help database systems to address those problems. Database systems are taking advantage of those services from operating systems. But partly for historic reason, partly for portability reason, some database systems have not exploited all the facilities. The discrepancy between operating system services and database systems requirement will exist as long as they target at different services. But the gap will decrease as operating systems pay more attention to database systems.

Reference:

[Bert98] Bertonni, L. Jonathan, "Understanding Solaris Filesystem and paging", *Sun Microsystems Technical report TR-98-55*, 1998, http://www.sun.com/research/techrep/1998/abstract-55.html
[Care89]Michael J. Carey, Rajiv Jauhari, Miron Livny: "Priority in DBMS Resource Scheduling". *VLDB* 1989: 397-410
[Chou85] Hong-Tai Chou, David J. DeWitt: "An Evaluation of Buffer Management Strategies for Relational Database Systems". *Very Large Data Bases (VLDB) Conference* 1985: 127-141
[Eykh92] Eykholt, J.R., etc "Beyond Multiprocessing: Multithreading the SunOS kernel" *Proceedings of the summer 1992 USENIX technical conference* Jun 1992 pp.11-18
[Khan92] Khanna, S. "Realtime scheduling in SunOS5.0", *proceedings of the winter 1992 USENIX technical conference*, Jan, 1992
[John94]Theodore Johnson, Dennis Shasha: "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm". *VLDB 1994:* 439-450
[Lane00]Lane, Tom, "Transaction processing in PostgreSQL", *a talk on Open Source Database Conference* Oct. 2000, http://www.postgresql.org/osdn/index.html
[Mcku84] Marshall K. McKusick, William N. Joy, Samuel J. Le_er, and Robert S. Fabry. "A Fast Filesystem For Unix". *ACM Transactions on Computer Systems*, 2(3):181-197, 1984.
[Momj00] Momjian, Bruce, "PostgreSQL: Introduction and Concepts", Addison-Wesley, 2000, ISBN 0-201-70331-9, http://www.postgresql.org/docs/aw_pgsql_book/index.html
[Nico92] Victor F. Nicola, Asit Dan, Daniel M. Dias: "Analysis of the Generalized Clock Buffer Replacement Scheme for Database Transaction Processing". *SIGMETRICS 1992*: 35-46
[Onei93] Elizabeth J. O'Neil, Patrick E. O'Neil, Gerhard Weikum: "The LRU-K Page Replacement Algorithm For Database Disk Buffering". *SIGMOD Conference 1993*: 297-306
[Prio98] Richard Mc Dougall, Triet Vo, Tom Pothier , "Priority Paging". http://www.sun.com/sun-on-net/performance/priority_paging.html
[Programmer] Solaris Unix Programmer's Manual 1997
[Reit76]Allen Reiter: "A Study of Buffer Management Policies for Data Management Systems". *Technical Summary Report 1619, Mathematics Research Center*, University of Wisconsin, Madison, 1976

[Rose91] Mendel Rosenblum and John K. Ousterhout. “The Design and Implementation of a Log-Structured File System”. *In Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pp. 1-15. ACM, 1991
[Selt93]Seltzer, I. Margo, “File System Performance and Transaction Support”, *Ph.D. Thesis for Computer Science in University of California at Berkeley,* 1993
[Stev92]Steve, W.Richard, “Advanced Programming In the Unix Environment”, *Addison-Welsley*, 1992, ISBN 0-201-56317-7
[Ston81]Stonebraker, M., ‘‘Operating System Support for Database Management,’’ *Communications of the ACM 24* 7 (July 1981), 412-418.
[Ston86] Stonebraker, M. and Rowe, L. A. 1986. “The design of POSTGRES” *In Proceedings of the ACM-SIGMOD Conference on Management of Data*, pp. 340--355.
[Ston90] Michael Stonebraker, Lawrence Rowe, and Michael Hirohama. "The implementation of postgres." *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125--142, 1990.
[Vaha96] Vahalia, Uresh , “Unix Internals—The new Frontiers”, 1996, *Prentice-Hall inc.* ISBN 0-13-101908-2
[VERT]Veritas file system white paper, 1996 (http://www.veritas.com)
[Vmsizing] “The Solaris Memory System – sizing, tools and architecture”, Sun Microsystems, 1998. http://www.sun.com/sun-on-net/performance/vmsizing.pdf
[XFS] XFS technical information, SGI corporate, http://www.sgi.com/ software/xfs/techinfo.html