PAPER

Shared Page Table: Sharing of Virtual Memory Resources*

Young-Woong KO†, ***Nonmember and*** **Chuck YOO**†, ***Regular Member***

SUMMARY Traditionally, UNIX has been weak in data sharing. By data sharing, we mean that multiple cooperative processes concurrently access and update the same set of data. As the degree of sharing (the number of cooperative processes) increases, the existing UNIX virtual memory systems run into page table thrashing, which causes a major performance bottleneck. Once page table thrashing occurs, UNIX performs miserably regardless of the hardware platforms it is running on. This is a critical problem because UNIX is increasingly used in environments such as banking that require intensive data sharing. We consider several alternatives to avoid page table thrashing, and propose a solution of which the main idea is to share page tables in virtual memory. Extensive experiments have been carried out with real workloads, and the results show that the shared page table solution avoids the page table thrashing and improves the performance of UNIX by an order of magnitude.

key words: *shared memory, thrashing, shared page table, virtual memory*

1. Introduction

One of the design goals of the original UNIX operating system was to support interactive programs that execute as processes; a process consists of three segments: text, data, and stack [2], [11], [21]. When processes execute the same program, the read-only text segment of the program is shared. However, the original UNIX design assumes that there is little data sharing between processes. By data sharing, we mean that multiple cooperative processes concurrently access and update the same set of data and that the data is shared in terms of interprocess communication (IPC). This assumption was reasonable considering the original design goals of UNIX and didn't pose any problem in traditional UNIX environments that do not need data sharing.

However, as machines running UNIX are getting faster and more powerful, UNIX is being used in environments that the original UNIX design didn't anticipate. An example of the new environments for UNIX is financial institutions that provide services to hundreds or thousands of people simultaneously. The financial data is managed by database management systems (DBMSs) that spawn processes in order to access the data. Several hundreds or even thousands of processes are not unusual, and mainframes used to be the only type of machine that could support the financial institutions. Due to the advance in processor technology, powerful multiprocessor servers running UNIX now have the processing power to handle such heavy workload.

Because these new environments demand new requirements from UNIX (such as security or high availability), it is often found that UNIX lacks certain functionalities to support the new environments. One such functionality that UNIX lacks is data sharing among many processes. For instance, the architecture of commercial DBMSs such as Oracle uses a large number of processes that work together in parallel in order to serve queries from different users [24]. The processes share a large-sized database critical to users and to the success of the day-to-day operation. Considering that the majority of servers are being used for DBMS-related applications [25], data sharing becomes a key requirement for UNIX. Thus how to provide an efficient data sharing facility is an important problem for UNIX because proprietary operating systems have thus far dominated in mission-critical environments.

To remedy the lack of data sharing in the previous UNIX systems, two IPC facilities have been introduced: memory mapped file interface (mmap) [9], [22] and shared memory facility [2]. The mmap interface allows a process to map a file object into its address space. By mapping the same file, processes can share data in the file through the mapped file interface. While mmap has many advantages such as the elimination of read() and write() system calls for doing I/O, this mapped file interface has several disadvantages in terms of data sharing [20]. First, the crash semantics of files is not well defined in UNIX. Since a mapped file is maintained in the file system buffer, a system crash leaves the state of the file on the disk unpredictable. This is not acceptable for mission-critical data. Second, because a mapped file can come from a remote file system like NFS, the implementation of mmap becomes complicated, which adds more overhead. Third, sharing semantics is limited by the kernel. An application program cannot use its domain specific knowledge to allow flexibility in accessing shared data. One popular use of the mapped file interface is to map shared libraries through dynamic linking [6].

Manuscript received December 25, 2001.
Manuscript revised August 23, 2002.

†The authors are with the Department of Computer Science, Korea University, Seoul, Korea.

*This work was supported by grant No.97-01-00-09-01-3 from the Basic Research program of the Korea Science & Engineering Foundation.

In comparison with mmap, the shared memory facility does not provide functionalities such as simplified file I/O because it does not use UNIX file system name space. However, due to its simplicity, the shared memory facility is being widely used in mission-critical environments. Shared memory enables processes to share the same physical memory pages. A process can attach a shared memory segment to the virtual address within its address space and then access the segment as if it were part of its normal address space without the need for any additional system calls. Shared memory therefore provides a very fast mechanism for sharing data between processes.

This paper identifies a deficiency in existing UNIX virtual memory system [7], [18] to support shared memory in a large scale. The deficiency is called page table thrashing [5], and it is in contrast to page trashing because page trashing can be avoided if enough physical memory exists. Page table thrashing can happen even with enough physical memory. To solve page table thrashing, this paper introduces the concept of sharing page tables and shows that shared page tables indeed prevent the performance loss caused by the page table thrashing. This paper also shows that shared page tables can be implemented with minimal kernel changes. We didn't want to redesign the existing virtual memory system or the architecture and the changes resulting from implementing the shared page table algorithm should minimally affect the rest of the kernel. Since the virtual memory system is in the heart of the kernel, we could not afford that our changes cause major changes in other subsystems like file system.

The remainder of this paper is organized as follows. Section 2 explains how page table thrashing occurs. Section 3 discusses possible solutions to solve page table thrashing. In Sect. 4, we describes the shared page table algorithm in detail. Section 5 explains the design and implementation of shared page tables in Sun's Solaris operating system. In Sect. 6, we present how experiments have been carried with real workloads and show the results. Section 7 concludes the paper.

2. Page Table Thrashing

When a process attaches a shared memory segment, the kernel allocates the page tables to map the pages that belong to the segment. Note that page tables are allocated on a per-process basis. If many processes are running with shared memory attached, a large number of page tables will be allocated. Suppose that the size of a shared segment is 128 MByte. The size of the page tables to map this segment is about 128 KByte with 4 KByte page size. If 100 processes are running on the system, 13 MByte physical memory needs to be reserved just for page tables (see Fig. 1). The actual physical memory requirement is much higher because of various kernel data structures (such as page structures

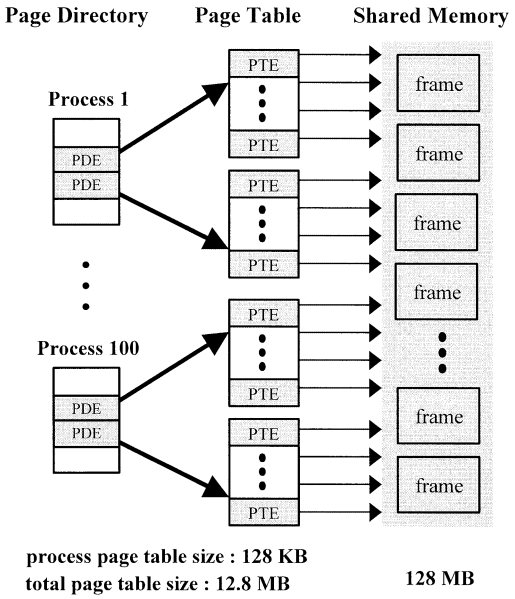

Fig. 1 Two level paging layout with shared memory.

for page table) associated with page tables. In addition to the physical memory, the kernel also has to allocate the same size of kernel virtual space. The size is too big as a chunk that a single kernel data structure occupies, and the kernel virtual space in UNIX is already tight enough so that the allocation of this amount of virtual space is very difficult.

It is not quite unusual that mission-critical environments run hundreds or thousands processes with shared memory attached. Note that the page tables are neither pageable nor swappable because page tables are typically allocated statically. Without enough page tables allocated at the boot time, the kernel can run into the shortage of page tables in the middle of processing mission-critical operations, and if that happens, the virtual memory system has to steal page tables from other processes.

The page table stealing is very expensive because all the pages mapped in the stolen page table have to be flushed before it is given to other process. If one page table maps 1024 pages, stealing one page table generates 1024 flushes (paging I/O). After flushing is done, the page table is placed in the new hierarchy of the page tables and it is remapped with different physical pages. The whole system can experience thrashing when processes steal page tables from each other. This high virtual memory overhead caused by page table stealing is called page table thrashing. A process is in page table thrashing if it is spending more time stealing page tables than executing. Thrashing is more likely on large-scale multiprocessor servers where concurrent processes run simultaneously.

3. Possible Solutions

3.1 Large Allocation of Page Tables

One may think that allocating a large amount of page tables may solve the page table shortage. This sounds like a straightforward solution, but it has three disadvantages. First, it is very hard to estimate how many page tables would be large enough to accommodate all situations. Second, in a monolithic kernel like UNIX, the kernel virtual space is a part of the user virtual space, so that there is a virtual space limitation in which kernel data structures like page tables can occupy. As a result, it is not possible to allocate an arbitrarily large amount of page tables. Finally, once allocated, the physical pages for page tables cannot be used for other purposes, so main memory may be wasted if too many page tables are allocated. Thus, we didn't regard this approach as a good solution.

3.2 Multithreading

Multithreading [14] has emerged as a new paradigm for multiprocessing in order to exploit the parallelism of multiprocessor systems. Many operating systems support or will support multithreading in various forms. Briefly speaking, there are two types of multithreading: kernel-level and user-level. Kernel-level multithreading means that the kernel is restructured with locks to prevent different processes from concurrently manipulating its data structures. User-level multithreading means that a process has several threads of control. Each thread has its own local variables and call/return stack. The instructions and global data are shared among all threads, and a change in a global variable by one thread will be seen by the other threads. Threads also share most of the operating system state of a UNIX process such as open files. The detailed description of threads or how to build multithreaded application are beyond the scope of this paper, and interested readers are referred to significant research literature [1], [14].

One solution to avoid the page table stealing is to use user-level multithreading where an user application uses threads instead of cooperating processes. Shared memory is replaced with global data which is shared by all the threads. Because the arbitrary number of threads can run on a single address space, one set of page tables can serve all the threads running in the address space, which can prevent page table shortage.

However, this approach requires that user applications have to be re-architected and re-implemented with user-level threads in order to take advantage of multithreading. This task of rewriting user applications with threads is not an easy task and involves major efforts. Since threads execute the same program independently, it is possible for threads to affect each other in sometimes surprising ways. Another factor is that there is yet no proven data to show the advantage of using thread paradigm for real-live user applications. The last point is that this approach dictates how user applications need to be written. We believe that operating systems have to provide reasonable services for a wide range of applications instead of forcing a certain style of behaviors from applications.

3.3 Bigger Pages

Another solution is to make the page size bigger, such as 256 KBytes. Then the number of page table entries to map a large segment would be much smaller, and so the page table shortage would not be a problem even though many processes are attaching it. However, there are several reasons why page sizes are hard to be increased. First, the page size is closely related to many architectural issues such as MMU. It is very difficult to change kernel without hardware and/or architectural changes. The second reason is that the big page size will increase the working set size due to fragmentation. In [19], the page size of 32 KBytes causes a significant increase in average working set size. The increased working set will need larger cache in order to reduce cache misses. Finally, bigger pages have a disadvantage in page allocation. When the kernel allocates big pages, since a page has to be contiguous in physical memory, it would have higher chance to sleep waiting for the memory available. Without larger cache and more memory, the overall system throughput may decrease using bigger pages. Finally, this approach causes more internal fragmentations, and forces the system to do all operations, such as copy-on-write, zero-fill on demand, and paging I/O at the new large page size.

3.4 Multiple Page Sizes

Another way to decrease page table stealing is by using multiple-page sizes [8], where each translation lookaside buffer (TLB) entry may map a page of variable size. For example, large contiguous regions in a process address space, such as program text, may be mapped using a small number of large pages (e.g., 64 KByte pages) rather than a large number of small pages (e.g., 4 KByte pages). There are several microprocessor architectures that support multiple page sizes, including R4000, Alpha, and UltraSPARC. For example, the UltraSPARC microprocessor provides hardware support to select 8 KByte, 64 KByte, 512 KByte, or 4 MByte pages. Multiple page sizes approach shows greatly increasing performances gains in large-page shared memory application. However, current operating systems have difficulties in supporting multiple page sizes and end up using only page size although TLB can support multiple page sizes.

3.5 Sharing Writable Segments

The virtual memory system in the Sprite operating system allows sharing writable segments [13]. The main purpose is that when a process is created by fork(), the child process can share the parent's data segment as well as the text segment. Each segment has its own page tables and backing store, and each process has pointers to the segments that belong to the process. Thus Sprite has a different process structure from UNIX which changes the semantics of the data segment in UNIX. This approach can be called shared segments, and forces user application to protect their global data, because all the data segments are shared by the forked process. Threads would be a better approach in sharing data segments.

3.6 Prototype Page Table Entry

In Windows NT [4], when a page is shared between processes, a level of indirection is inserted into the page tables. The indirection is called prototype page table entry. The page table entries that map the same page points to a prototype page table entry. By doing this, maintaining page consistency is simplified since a single prototype page table entry maps a page. However, page tables are still allocated per process basis, and so page table stealing can still take place.

3.7 Shared Text Segment Mechanism

Shared text segment mechanism allows multiple processes to share the same physical page for sharing read-only text segments. Different instantiations of the same program share text pages and dynamically linked libraries such as libc, libX and libnsl. This approach, however, do not allow sharing of page tables; each instantiations of a program has their own copy of page tables. Thus, it makes redundant copy of the same page tables that maps the same read-only text segment. On the other hand, shared data segment with shared page table, allows multiple processes to share the same page tables that map the same segment. Thus, it can address the redundancy in shared text segment mechanism.

4. Shared Page Table Algorithm

The key idea of the shared page table algorithm is to create "master page table (MPT)" for a shared memory segment, and share the MPT among the processes that attach the segment. Suppose that the size of a shared segment is 128 MByte. The size of the MPT to map this segment is about 128 KByte with 4 KByte page size. If 100 processes are running on the system, each process uses MPT to access the segment. Consequently, page tables to map 100 processes are only 128 KByte. In this way, we dramatically reduce page tables and kernel virtual memory for shared memory segment. Figure 2 shows the page table layout with master page table.

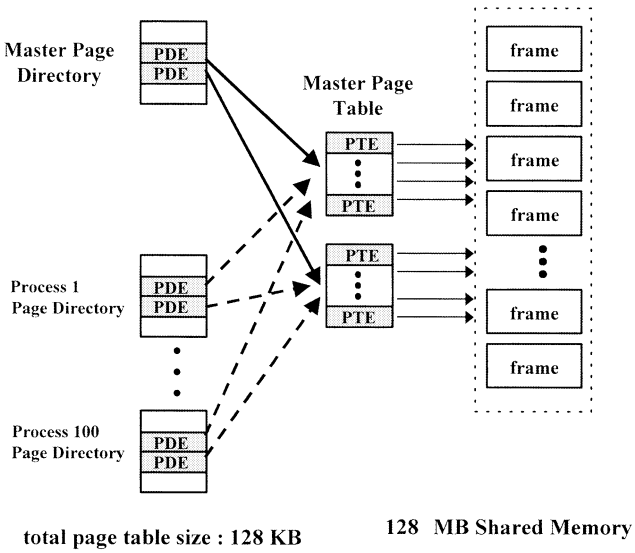

Fig. 2 Page table layout with master page table.

In this section, we present the shared page table algorithm and its usage model, with particular emphasis on how it is possible to share page tables. The algorithm is composed of following four operations: (1) creating a master page table (2) attaching the shared memory segment (3) detaching the shared memory segment (4) destroying the master page table.

4.1 Creating a Master Page Table

In the shared page table algorithm, if a process attempts to create a newly shared memory segment, users need to invoke shmget() system call. Specifically, the IPC_SHARE_PAGE_TABLE flag in the shmflg, which is an argument to the shmget() call, must be set to indicate the system to use the shared page table algorithm. The first thing that the shmget() function does is allocating one page of physical memory for a master page directory and initializing master page directory entry. After this, according to the size of shared memory, we create a number of master page tables and associate them with master page directory entry. Finally, we initialize the share count to zero. The share count is the number of processes that attach shared memory segment. In our algorithm, newly created address translation data structure (master page directory and master page table) is not process-dependent. It only preserve shared memory address mapping. As a result, when the shmget() function succeeds, there exist master page directory, master page table group. Figure 3 shows the relationship among master page directory, master page table and shared memory.

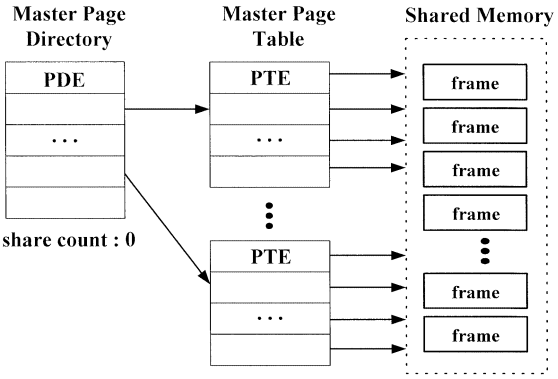

Fig. 3 Master page directory and master page tables.

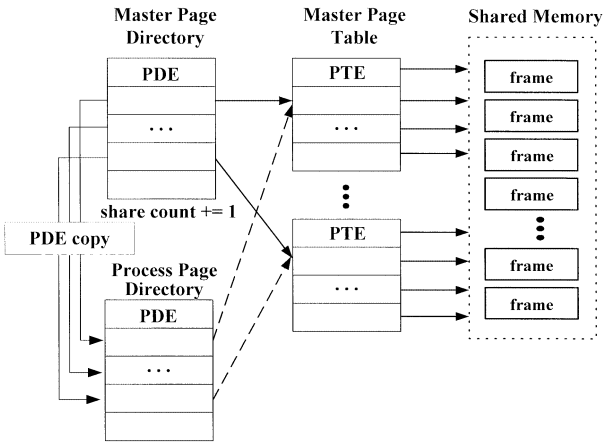

Fig. 4 Mapping the shared memory to the process's user segment.

4.2 Attaching the Shared Memory Segment

If a process wants to attach shared memory segment to the process's segment, users need to invoke shmat() system call. Specifically, in the shared page table algorithm, the SHM_SHARE_PAGE_TABLE flag in the shmflg, which is an argument to the shmat() call, must be set to indicate the system to use the shared page table algorithm. The shmat() call maps the shared memory segment to the process's segment. In the shared page table algorithm, the shmat() call simply copies master page directory entry into the page directory of the calling process. After this, the share count is increased. As shown Fig. 4, although as many as processes attach shared memory segment to their address space, the number of page table is not increased.

4.3 Detaching the Shared Memory Segment

If one process wants to detach shared memory segment, the shmdt() function is called. In the shared page algorithm, the shmdt() function simply invalidate the page directory entry of the calling process and decrease the share count. If the share count becomes zero, the master page directory and group of master page tables are destroyed. Figure 5 shows the diagram of detaching the shared memory segment from the process's user segment.

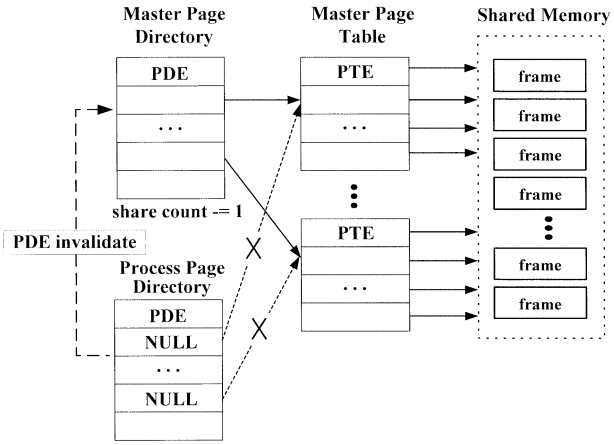

Fig. 5 Detaching the shared memory from the process's user segment.

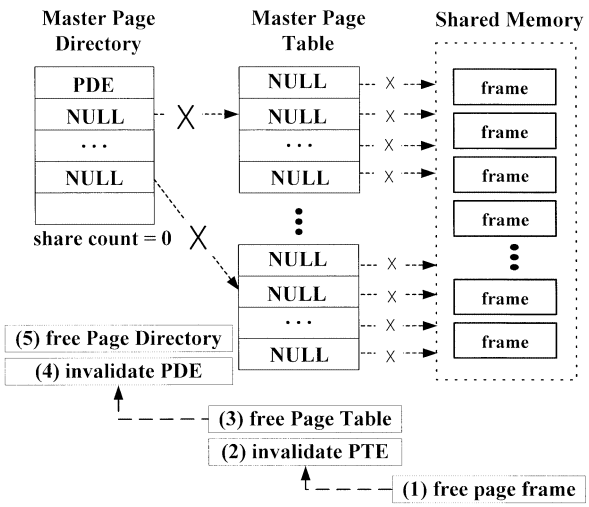

Fig. 6 Destroying the shared memory from the process's user segment.

4.4 Destroying the Master Page Table

If one process calls shmdt() function, and the share count of the master page directory equals zero, then there is no reference to the shared memory segment. In this case, the master page directory and master page tables are invalidated and freed. Figure 6 shows the destroying procedure as follows: (1) free all page frames that compose the the shared memory (2) invalidate the master page table entry (3) free master page tables (4) invalidate master page directory entry (5) free master page directory.

5. Implementation in a Commercial Operating System

To demonstrate the effectiveness of the shared page table algorithm, we have implemented the shared page table in the Solaris 2 operating system from Sun Microsystems. Our choice of operating system is based on the following reasons. First, it is a widely-used general purpose operating system, which gave us the opportunity to examine the impact of our approach on the performance of real mission-critical applications such as DBMS, on-line transaction processing (OLTP), and on-line analytical processing (OLAP). Second, the Solaris operating system covers different processor architectures. Finally, we had access to the complete source code for the Solaris.

We briefly describe the virtual memory system in the Solaris operating system and how the shared page table algorithm is implemented.

5.1 Virtual Memory System in the Solaris Operating System

The Solaris operating system was derived from Berkeley UNIX [11], and has moved to be a SVR4-based system [2] with many extensions like multithreading [14]. Because, the virtual memory system of Solaris has the same structure as the virtual memory system of SVR4 with several enhancements including a multithreaded HAT layer [23], the principles described in this paper can thus be applied to other SVR4-based operating systems.

In Solaris, address spaces are constructed out of mappings to virtual memory objects, and the main memory is used to cache the contents of virtual memory objects. The most common type of virtual memory object is a file, and the vnode [10] is the fundamental structure to name file objects. Vnodes provide a file system independent abstraction that allows access to a file object. Each memory page is named by a backing store for that page. In the case of a file object, the name of a page is a pair of (vnode, offset) where vnode is the file that the page comes from and offset is the location in the vnode.

A second common virtual memory object is anonymous memory. Unlike file mapping, the backing store of anonymous memory is swap space whose name is not known to users. Each anonymous page is assigned a name from the pool of swap space that is maintained by the kernel. Anonymous pages are used for many purposes within the kernel, including uninitialized data and stack segments of processes. Each anonymous page is represented by an opaque handle called an anon slot. The anon slot contains the name of the backing store. Typically several anon pages are allocated together, and in order to describe such a cluster of anon pages as a unit, the pointers to the anon slots are stored in an array called anon map [3].

Each mapping which comprises a process' address space is represented by a segment. A segment provides many services that need the cooperation of vnode operations [15] such as handling faults on an address within the segment. The segment resolves faults by filling a page from the backing store that the segment maps through a vnode operation. There are several types of segments in the system. Among them, the vnode segment type (segvn) is most heavily used. The segvn provides shared and private mappings to files and anonymous memory. Shared mapping always writes the current data of its mapped object. For the private mapping, the first write access to a page in segvn causes a copy-on-write operation that creates a private page. The page tables are allocated and freed through the HAT layer. Additionally, the HAT layer is responsible for loading and unloading translations into page tables. The other details of the Solaris virtual memory system are described in [6], [12].

The memory management unit that we used is the SPARC reference MMU (SRMMU) [17]. It is designed to be a simple-chip implementation that can provide general-purpose memory management to efficiently support a large number of processes running a wide variety of applications.

The primary functions of the SRMMU are to perform virtual-to-physical address translation and to provide memory protection. The SRMMU uses three levels of page tables in main memory to store address translation information. When an address in a segment is accessed for the first time, the SRMMU will fault on the address because the page table is not yet set up and the page table entry is not valid. The segment driver then tries to handle this pagefault. In the case of segvn, it acquires a free page and fills the page from the vnode. The segvn then calls the HAT layer to set up the page table to map the page. The HAT layer first makes a page table entry and loads the entry into the proper page table. Once the page table is set up, the SRMMU will do a table walk to get the physical address of the requested page of any subsequent address references.

5.2 Implementing the Shared Page Table Algorithm in the Solaris Operating System

To implement the shared page table algorithm, instead of having segvn allocate the page tables per process, we create a new segment type called segshm. The role of segshm is to create and maintain the master copy of page tables. When a shared memory is created, segshm allocates master page tables and physical pages of the given size that compose the shared memory, and map the pages into the L1PT and its associated level 2 and 3 page tables (L2PT and L3PT). Note that the pages of shared memory are pre-mapped before the faults.

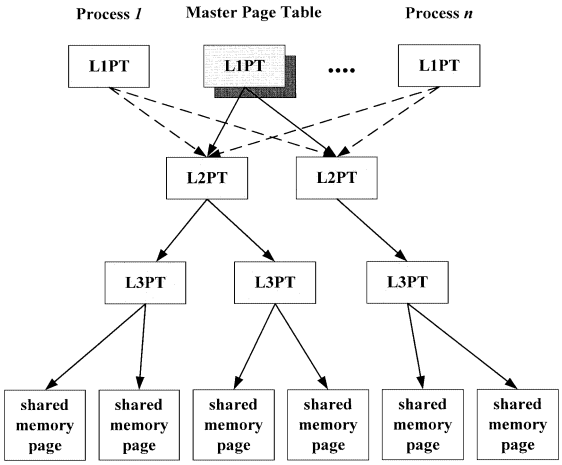

Fig. 7 Page table layout by segshm.

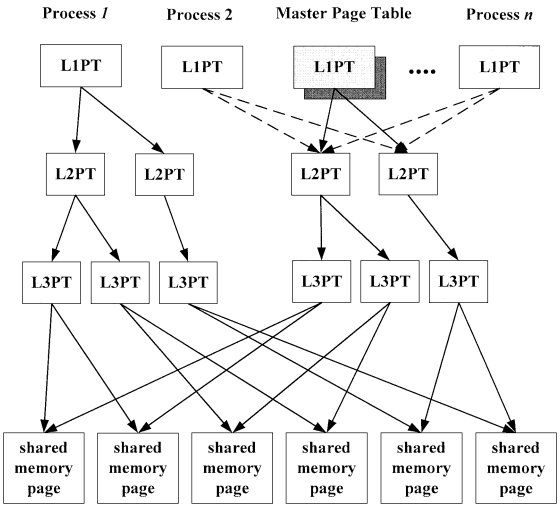

Fig. 8 Page tables with segvn and segshm.

When a process attaches the shared memory, the entries in the master L1PT that were created by segshm are copied to the L1PT of that process starting from the attach address. Once the L1PT entries are copied, references to any address in the shared memory from the process attaching the shared memory are through the same level 2 and level 3 page tables that the master L1PT is pointing to. Figure 7 shows how processes attaching the shared memory share the master page tables. Note that the unit of sharing is an entry in L1PT, and this constrains the attach address to the size which a L2PT maps. With the shared page tables, only one set of page tables is needed to support an arbitrary number of processes, without allocating a large amount of page tables.

When a shared memory is detached, the entries in the caller's L1PT are invalidated and thus the caller process is separated from the shared memory, while the master copy still exists. When the shared memory is destroyed, the mapping in the master page tables is destroyed and the page tables are freed. The pages belonging to the shared memory are also freed at this time.

In addition to the saving in page table allocation, sharing page tables has other advantages. In systems where page tables can be cached, segshm could have a higher cache hit than segvn because the MMU table walk would hit the cache without going to memory. Another advantage is the reduced overhead of maintaining page consistency. Because a page may be mapped more than once, the HAT layer maintains page consistency to indicate whether the page has been modified or referenced. This is necessary for the paging algorithm. With per-process page tables, the HAT layer has to go through the page table of every process which maps the page, and synchronize with the page table entries. Sharing page tables simplifies this step because it has a single page table entry for a page.

5.3 Interaction with Process File System

The process file system (called /proc) allows inspection and modification of a process' image through normal file system calls such as lseek(), read(), and write() [9]. In SVR4 the concept has been refined to a general interface to UNIX process model abstraction. This facility permits monitoring of reference and modified bits of pages accessed by on each process in the system through per-process page tables. When processes use shared page tables, however, it will not be possible to observe an individual process' use of the shared pages because there is one set of page tables and the reference and modified bits reflect the aggregate of the page accesses.

In order to provide the compatibility with the semantics of /proc, we decided to support both shared and per-process page tables. In other words, the processes that need to preserve /proc semantics are allowed to use the per-process page tables, while other processes can use shared page tables simultaneously. The per-process page table is constructed by invoking segvn and the shared page table by segshm. Figure 8 depicts how the shared memory pages are accessed through the shared and per-process page tables. Process 1 uses per-process page tables and the other processes shared page tables.

Supporting both segvn and segshm is implemented by using anon map structure of a shared memory. An anon map structure exists per shared memory segment. Remember that an anon map structure is an array of anon slots, each of which points to an anonymous page that is allocated to the shared memory. There are three possible states of anon map depending on how segvn and segshm are interleaved. The first case is an empty anon map. This is when segshm is invoked before any segvn pagefaults. The second is that the anon map is partially filled, which means that segshm is invoked

when pagefaults of some addresses in shared memory have been resolved by segvn. The last one is that anon map is completely filled. To handle these cases, when the master page tables are created, segshm examines the anon slots in anon map of the shared memory segment one by one to see if the page was already allocated (by segvn). If the page was allocated, it is brought into memory because the page may not be in memory, and segshm maps the page into the master copy of page table. If the page was not allocated, segshm allocates a page, maps the page, and mark the anon slot. Once segshm finishes allocating and mapping the pages, subsequent segvn will map the already allocated pages into the per-process page tables.

5.4 Protection

The protection for accessing pages is usually provided by the protection bits in the page table entries. With per-process page tables, each process can have different protections for the same set of physical pages. This permission mechanism can be used for per-process protection of page. However, shared page tables cannot allow the per-process protection mechanism based on page tables because it has only one set of page tables. Instead, it depends on the access permission mechanism of kernel. UNIX grants read, write, and execution permission of shared memory for three different memberships of processes: owner, group and others. Accessing the pages that belong to a shared memory segment is controlled by the process' permission on the shared memory segment. Since the page table entries in the master page tables allows read and write accesses to the pages, any attempt to use shared page tables without write permission is denied. If a process needs per-process protection, it can use per-process page tables. The process can coexist with other processes using shared page tables as explained earlier in Sect. 5.3.

Applications that exhibit large amounts of read-write sharing do not need per-protection mechanism. In database systems, for example, many server processes could share a large pool of read-write page representing a buffer cache with shared page tables (e.g., Oracle uses shared memory for its Shared Global Area, which is how Oracle does its caching of data, indexes, stored procedures).

6. Performance Measurements

We have implemented the shared page table algorithm in Sun's Solaris 2 operating system. After implementation, we have carried out extensive experiments. The goal of our experiments was to see the effect of page table stealing and the improvement resulting from shared page tables. For our experiments, we used a benchmark similar to the TPC benchmarks. The TPC benchmarks are the industry standard benchmarks that are published by the Transaction Processing Performance Council (TPC) that consists of the major vendors in the industry. The TPC benchmarks use the shared memory facility extensively and stresses the virtual memory system heavily.

The workload simulates a banking system; the bank has one or more branches with each branch having multiple tellers. There are many individuals, each represented by an account. The branch, teller, and account information are stored in databases. A transaction in the benchmark randomly reads an account, updates the account, and propagates the update to the teller and branch balances. The performance was measured as transactions per second (TPS).

In general, the TPC benchmarks are difficult to set up, and there are several severe restrictions to follow, including scaling of database and response time constraints. For example, scaling of databases requires that per TPS, the ratio of branch and teller and account numbers is 1:10:100,000. In the experiments in this paper, we followed most of the specifications in the TPC benchmarks except a few things like mirroring. Rather than spending time to conform to the specification, we focused on identifying the performance differences, solely based on kernel changes within a given configuration.

We used three configurations to run the benchmark. The first configuration is a SPARCstation 10 workstation with a single 50 MHz superSPARC processor, 64 MByte of primary memory, and 3 GByte of local disk space. The database was scaled to 250 TPS. The second configuration is a Sun 600 series tightly coupled multiprocessor machine, a medium-range server. Sun 600 series used in our experiments is equipped with four 50 MHz superSPARC processor, 128 MByte of primary memory, and 5 GByte of local disk space. The database was scaled to 250 TPS. The shared memory size used in both configurations was 40 MByte. The third configuration is a Sun 1000 series tightly coupled multiprocessor machine, a high-end server. Sun 1000 series is equipped with eight 50 MHz superSPARC processor, 128 MByte of primary memory, and 10 GByte of local disk space. The scaling of the database was 350 TPS. The shared memory size used in this configuration was 60 MBytes. We ran the benchmark with kernels using shared page table algorithm and traditional page table allocation algorithm (non-shared page table) for shared memory, and the number of processes attaching shared memory varied from 10 to 100.

The reason to use different classes of machines was to evaluate the effect of shared page tables in each class, not to compare performance between the machine classes because each configuration has differences. Compared with other experiments done in the literature [16] which used a 10 TPS database on one disk, we believe that our experiments are much more extensive and a close approximation to a realistic workload, which

provides meaningful evaluation of shared page tables. The benchmark consists of 180 seconds of ramp-up to fill the shared memory pages, 300 seconds of steady-state run followed by 90 seconds of ramp-down. TPS is measured during the steady-state period.

First, we were surprised at the sharp performance degradation with the non-shared page table algorithm as the process load increases. This was shown to be true for all three configurations. The performance started to degrade at about 40 processes and kept degrading. In order to do instrumentation of the kernel, we put counters inside the kernel. As the benchmark was running, we read the counter values and verified that the degradation was indeed due to the page table shortage. When page table stealing took place, the system time increased substantially, and the system response time was also very slow. When the kernel with shared page tables was running, the degradation disappeared as shown in Figs. 9, 10 and 11, and the performance stayed flat over the range of processes. The performance improvement at 100 processes in Solaris implementation is 81%, 126%, and 210% respectively in configurations 1, 2, and 3.

With a small number of processes (10 to 30), the shared page table algorithm performed a little better than the non-shared page table algorithm. We think that it is because the segshm driver is more efficient than segvn, i.e., segshm has more streamlined code path than segvn that handles a variety of cases like local files and NFS files. In Solaris, comparing the slope of the curves shows that the performance degradation in configuration 3 is more rapid than that in configuration 2, and configuration 2 more rapid than configuration 1. When more current processes are active in parallel, taking away page tables from active processes incurs bigger penalty in the performance. If page table stealing takes place in other environments like window systems, the performance impact is much less significant because only a few of them are active at a time.

To gain further insight into the amount of memory size, we carried out the experiments to compare the amount of memory consumption for page table between per-process page table algorithm and shared page table algorithm. In our experiments, the number of page table for text/data segment is fixed with 10 and the

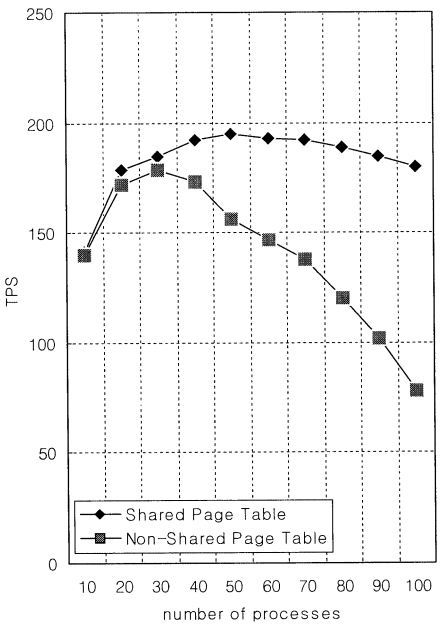

Fig. 10 TPS of configuration 2.

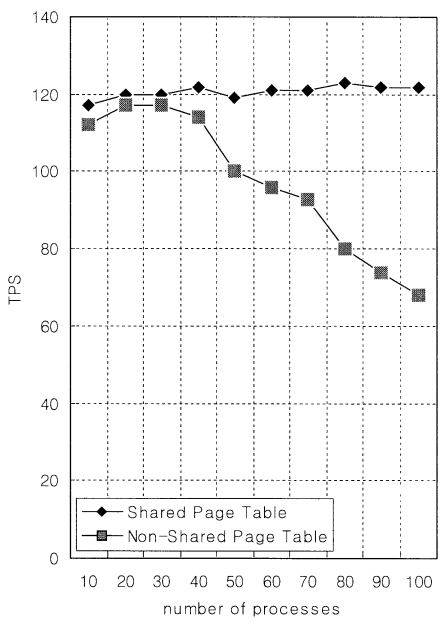

Fig. 9 TPS of configuration 1.

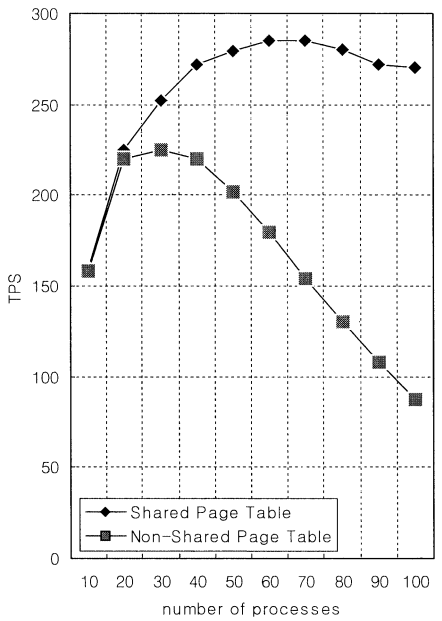

Fig. 11 TPS of configuration 3.

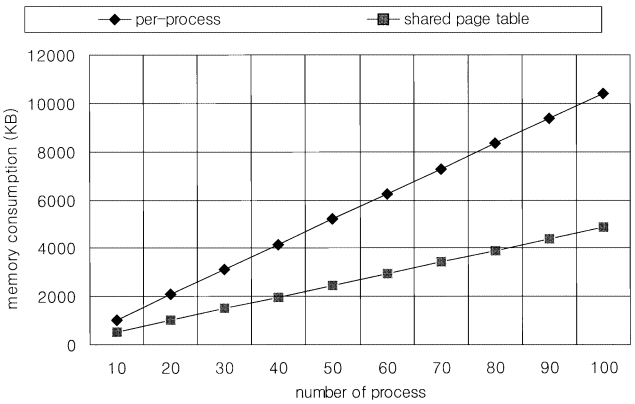

Fig. 12 The amount of memory consumption for page table.

shared memory size is 60 MByte. Because a page table can cover 4 MByte of memory address, 15 page tables are needed for shared memory in our experiments. As a result, in per-process page table algorithm, a process needs 104 KByte of memory; 40 KByte (10 page tables * 4 KByte) for text/data segment, 60 KByte (15 page tables * 4 KByte) for shared memory and 4 KByte for 1 page directory. In shared page table algorithm, a process needs 108 KByte of memory; 40 KByte (10 page tables * 4 KByte) for text/data segment, 60 KByte (15 page tables * 4 KByte) for shared memory, 4 KByte for 1 page directory and 4 KByte for 1 master page directory.

As the number of process attaching shared memory is increased, if we use the per-process page table algorithm, the amount of memory consumption is increased in proportion to the sum of page table size of text/data segment and shared memory. On the other hand, if we use the shared page table algorithm, the amount of memory consumption is increased in proportion to the text/data segment only. Figure 12 shows the amount of memory consumption for page tables when the number of processes attaching shared memory varied from 10 to 100.

7. Conclusion

We have investigated the data sharing facility in UNIX and its deficiency in mission-critical environments. The main problem is the page table stealing. As a solution, we have presented the design and implementation of the shared page tables in UNIX. Our experiments show that the shared page tables increase the data sharing performance dramatically by avoiding page table stealing. The performance improvement was observed consistently in different classes of machines. The contribution of this paper is to introduce the concept of sharing mapping resources such as page tables and to prove that sharing page tables provides crucial performance improvement for cooperating processes. Although this paper explores shared page tables in the context of UNIX, we believe that the concept of sharing virtual memory resources can be applied to other operating systems or different MMU.

In the beginning of this project, there were three design goals in addition to improving data sharing performance. One was that a solution should fit within the existing virtual memory architecture. We didn't want to change the existing virtual memory system or the architecture. The second goal was that the changes resulting from implementing a solution should minimally affect the rest of the kernel. Since the virtual memory system is in the heart of the kernel, we could not afford that our changes cause major changes in other subsystems like the file system. Finally, we wanted to have a simple solution. The alternative solutions that were proposed in Sect. 3 were examined in view of above design goal, and we found that they are not appropriate. Instead, we believe that the shared page tables satisfy the design goals nicely. As we have described, our solution is very simple and fits within the existing virtual memory architecture. Moreover it minimally affects the rest of the kernel.

There are still several issues to be investigated in the virtual memory system for intensive data sharing applications. One example is how to reduce the TLB misses. TLB is a fast buffer containing recently used virtual-to-physical address translations. Most contemporary computers that employ virtual memory use TLB to increase the address translation speed. As cycles-per-instruction get smaller with faster RISC processors, the penalty caused by TLB misses becomes non-negligible. Detailed design and implementation need to be done and the experiment results are yet to be seen.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A new kernel foundation for UNIX dvelopment," Proc. 1986 Summer USENIX Conference, 1986.
[2] M. Bach, The Design of the UNIX Operating System, Prentice-Hall, 1986.
[3] H. Chartock and P. Snyder, "Virtual swap space in SunOS," Proc. Autumn 1991 EUUG Conference, 1991.
[4] H. Custer, Inside Windows NT, Microsoft Press, 1993.
[5] P. Denning, "Virtual memory," ACM Computing Surveys, vol.2, no.3, pp.153–189, Sept. 1970.
[6] R. Gingell, M. Lee, X.T. Dang, and M.S. Weeks, "Shared libraries in SunOS," Proc. 1987 Summer USENIX Conference, 1987.
[7] R. Gingell, J.P. Moran, and W.A. Shannon, "Virtual memory architecture in SunOS," Proc. 1987 Summer USENIX Conference, 1987.
[8] I. Subramanian, C. Mather, K. Peterson, and B. Raghunath, "Implementation of multiple pagesize support in HP-UX," Proc. USENIX 1998 Annual Technical Conference, USENIX Assoc., June 1998.
[9] T. Killian, "Processes as files," Proc. 1984 Summer USENIX Conference, pp.203–207, 1984.
[10] S. Kleiman, "Vnodes: An architecture for multiple file systems types in Sun UNIX," Proc. 1986 Summer USENIX

Conference, pp.238-247, June 1986.
[11] S. Leffler, M.K McKusick, M.J. Karels, and J.S. Quarterman, 4.3 BSD UNIX Operating System, AddisonWesley, 1989.
[12] J. Moran, "SunOS virtual memory implementation," Proc. Spring 1988 EUUG Conference, 1988.
[13] M. Nelson, "Virtual memory for the sprite operating system," Technical Report UCB/CSD 86/301, Computer Science Division (EECS), University of California, Berkeley, 1986.
[14] M. Powell, S.R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks, "SunOS multithread architecture," Proc. 1991 Winter USENIX Conference, 1991.
[15] D. Rosenthal, "Evolving the vnode interface," Proc. 1990 Summer USENIX Conference, pp.107–117, June 1990.
[16] M. Seltzer and M. Olson, "LIBTP: Portable, modular transactions for UNIX," Proc. 1991 Winter USENIX Conference, 1991.
[17] SPARC Internal, ed. SPARC Architecture Manual Version 8. 1991.
[18] E. Sznyter, P. Clancy, and J. Crossland, "A new virtual memory implementation for Unix," Proc. 1986 Summer USENIX Conference, pp.81–88, 1986.
[19] M. Talluri, S.I. Kong, M.D. Hill, and D.A. Patterson, "Tradeoffs in supporting two page sizes," Proc. International Symposium on Computer Architecture, pp.415–424, 1992.
[20] A. Tevanian, R.F. Rashid, M. Young, D.B Golub, M.R. Thompson, W.J. Bolosky, and R. Sanzi, "A UNIX interface for shared memory and memory mapped files under mach," Proc. 1987 Summer USENIX Conference, pp.53–67, 1987.
[21] K. Thompson, "UNIX implementation," The Bell System Technical Journal, vol.57, no.6, pp.1931–1946, 1978.
[22] UNIX System Laboratories, Operating System API Reference, UNIX SVR4.2, UNIX Press, Prentice-Hall, Englewood Cliffs, NJ, 1992.
[23] D. Williams, Personal communication, 1993.
[24] H. Yoo and T. Rogers, "UNIX Kernel support for OLTP performance," Proc. 1993 Winter USENIX Conference, pp.241–247, 1993.
[25] H. Yoo and K. Ko, "Operating system performance and large servers," ACM SIGOPS, 1994.

Young-Woong Ko received both a bachelor and a master degree in computer science from Korea University, Seoul, Korea, in 1997 and 1999, respectively. He is currently pursuing the Ph.D. degree in computer science at Korea University. His research interests include operating system, soft real-time scheduling and multimedia scheduling.

Chuck Yoo received the B.S. degree in electronics engineering from Seoul National University, Seoul, Korea and the M.S. and Ph.D. in computer science in University of Michigan. From 1990 to 1995, he worked as researcher in Sun Microsystems Lab. He is now a Professor in College of Information and Communications, Korea University, Seoul, Korea. His research interests include high performance network and multimedia streaming for thin client.