Dominique A. Heger, Fortuitous Technology, (dom@fortuitous.com), Austin, TX, 2006

New Features in Linux 2.6 – Performance, Scalability, and Stability

Introduction

Over the last several years, the Linux operating system has gained acceptance as the operating system of choice in many scientific and commercial environments, respectively. Today, the performance aspects of the Linux operating system is regarded as being very competitive compared to the other big three UNIX flavors, AIX, HP-UX, and Solaris. This statement is particularly applicable to smaller SMP systems with up to 4 processors. Recently, there has been an increased emphasis on the performance of Linux in a mid to high-end enterprise-class environment, consisting of SMP systems that are configured with up to (or sometimes surpassing) 64 CPUs. Therefore, scalability and performance related aspects surrounding Linux 2.6 are paramount to support applications on large systems that are actually scalable to high CPU counts. The objective of this article is to highlight the performance improvements made for Linux 2.6, with an emphasis on scalability related features. The actual components that are briefly introduced include the Linux 2.6 CPU scheduler, the virtual memory, and I/O subsystem.

The Virtual Memory (VM) Subsystem

As the physical memory subsystem has a lower latency than the disk subsystem, one of the challenges faced by the VM subsystem is to keep the most frequently referenced portions of memory in the faster primary storage. In the event of a physical memory shortage, the VM subsystem is required to surrender some portion of physical memory by transferring infrequently utilized memory pages out to the backing store. Therefore, the VM subsystem provides (next to the large address space) 'resource virtualization' in the sense that a process does not have to manage the details of physical memory allocation. Further, the VM provides information and fault isolation, as each process executes in its own address space. In most circumstances, hardware facilities in the memory management unit (MMU) perform the memory protection functionality by preventing a process from accessing memory outside its legal address space (except of course for memory regions that are explicitly shared among processes). The virtual address space of a process is defined as the range of memory addresses that are presented to the process as its environment. At any time in the life cycle of a process, some addresses may be mapped to physical addresses and some may not.

Most modern computer architectures support more than one page size. As an example, the IA-32 architecture (not the operating system) supports either 4KB or 4MB pages. The Linux operating system used to only utilize large pages for mapping the actual kernel image. In general, large page usage is primarily intended to provide performance improvements for high performance computing (HPC) applications, as well as database applications that (usually) have large working sets. But, any memory access intensive application that utilizes large amounts of virtual memory may obtain performance improvements by using large pages. Linux utilizes 2MB or 4MB large pages, AIX® uses 16MB large pages, whereas Solaris large pages are 4MB in size. The large page performance improvements are attributable to reduced translation lookaside buffer (TLB) misses. This is due to the TLB being able to map a larger virtual memory range, therefore increasing the reach of the TLB. Large pages further improve the process of memory prefetching, by eliminating the necessity to restart prefetch operations on 4KB boundaries.

Physical page allocation in Linux 2.6 is based on a (relaxed) buddy algorithm. The mechanism itself resembles a lazy buddy scheme that defers the coalescing part. In general terms, the buffer release activity of the coalescing portion involves a two step process, where the first step consists of pushing a buffer back onto the free list, and hence make the buffer available to other requests. The second step involves referencing the buffer as free (normally in a bitmap) and coalescing the buffer (if feasible) with adjacent buffers. A regular buddy system implementation performs both steps on each release, whereas the lazy buddy scheme performs the first step, but defers the second step and only executes it depending on the state of the *buffer class*. The Linux 2.6 implementation of the discussed lazy buddy concept utilizes per CPU page lists. These pagesets contain two lists of pages (marked as *hot* and *cold*), where the hot pages

have been used recently, and the expectation is that they are still present in the CPU cache. On allocation, the pageset for the running CPU will be consulted first to determine if the pages are available, and if it is the case, the allocation will proceed accordingly. In order to determine when the pageset should be released or allocated, a low and high watermark based approach is being used. As the low watermark is reached, a batch of pages will be allocated and placed on the list. As the high watermark is reached, a batch of pages will be freed simultaneously. As a result of the new and improved implementation, the spinlock overhead (in regards to protecting the buddy lists) is significantly being reduced.

In the 2.4 Linux kernel, a global daemon (*kswapd*) acted as the page replacement activator. The *kswapd* was responsible for keeping a certain amount of memory available, and under memory pressure would initiate the process of freeing pages. In Linux 2.6, a *kswapd* daemon is available for each node in the system. The general idea behind this design is to avoid situations where a kswapd would have to free pages on behalf of a remote node. In Linux 2.4, a spinlock has to be acquired before removing pages from the least recently used (LRU) list. In Linux 2.6, operations involving the LRU list take place via *pagevec* structures, which allows to either add or remove pages from the LRU lists in sets of up to *PAGEVEC_SIZE* pages. In the case of removing pages, the *zone lru_lock* lock is acquired and the pages placed on a temporary list. Once the list of pages to be removed is assembled, a call to *shrink_list()* is initiated. The actual process of freeing pages can now be performed without acquiring the *zone lru_lock* spinlock. In the case of adding pages, a new page vector struct is initialized via *pagevec_init()*. Pages are added to the vector through *pagevec_add()*, and then committed to being placed onto the LRU list via *pagevec_release()*.

CPU Scheduler

In general, the thread scheduler represents the subsystem responsible for decomposing the available CPU resources among the runnable threads. The behavior and decision making process of the scheduler has a direct correlation to thread fairness and scheduling latency. Fairness can be described as the ability of all threads to not only encounter forward progress, but to do so in a relatively even manner. The opposite of fairness is starvation, a scenario where a given thread encounters no forward progress. Fairness is frequently hard to justify, as it represents an exercise in compromises between global and localized performance. Scheduling latency describes the actual delay between a thread entering the runnable state (*TSRUN*) and actually running on a processor (*TSONPROC*). An inefficient scheduling latency behavior results (as an example) in a perceptible delay in application response time. Hence, an efficient and effective CPU scheduler is paramount to the operation of the computing platform. It decides which task to run at what time and for how long. In general, real-time and time-shared jobs are distinguished, each class revealing different objectives. Both are implemented through different scheduling disciplines embedded into the scheduler.

Linux assigns a static priority to each task that can be modified through the *nice()* interface. Linux has a range of priority classes, distinguishing between real-time and time-sharing tasks. The lower the priority value, the higher the logical priority of a task, or in other words its *general importance*. In this context, the discussion always references the logical priority when elaborating on priority increases and decreases. Real-time tasks always have higher priority than time-sharing tasks. The Linux 2.6 scheduler (referred to as the *O(1)* scheduler) is a multi queue scheduler that assigns an actual run-queue to each CPU, promoting a CPU local scheduling approach. The *O(1)* label basically describes the asymptotic upper bound time complexity for retrieval. In other words, it refers to the property that no matter the workload on the system, the next task to run will be chosen in a constant amount of time. The previous incarnation of the Linux scheduler utilized the concept of *goodness* to determine which thread to execute next. All runnable tasks were kept on a single run-queue that represented a linked list of threads in a *TSRUN* (*TASK_RUNNABLE*) state. In Linux 2.6, the single run-queue lock was replaced with a per CPU lock, ensuring better scalability on SMP systems. The per CPU run-queue scheme adopted by the *O(1)* scheduler decomposes the run-queue into a number of *buckets* (in priority order) and utilizes a bitmap to identify the *buckets* that hold runnable tasks. Locating the next task to execute requires a read from the bitmap to identify the first bucket with runnable tasks, and choosing the first task in that bucket's run-queue. The per-CPU run queue consists of two vectors of task lists, labeled as the active and the expired vector. Each

vector index represents a list of runnable tasks (each at its respective priority level). After executing for a period of time (time slice dependent), a task moves from the active list to the expired list to insure that all runnable tasks get an opportunity to execute. As the active array becomes empty, the expired and active vectors are swapped by modifying the pointers. Occasionally, a load-balancing algorithm is invoked to rebalance the run-queues to ensure that a similar number of tasks are available per CPU. As already discussed, the scheduler has to decide which task to run next and for how long. Time quanta in the Linux kernel are defined as multiples of a system tick. A tick is defined as the fixed delta (*1/HZ*) between two consecutive timer interrupts. In Linux 2.6, the *HZ* parameter is set to 1000, indicating that the interrupt routine *scheduler _tick()* is invoked once every millisecond, at which time the currently executing task is charged with one tick. Setting the *HZ* parameter to 1000 does not improve the responsiveness of the system, as the actual scheduler timeslice is not affected by this setting. For systems that primarily execute in a number crunching mode, the *HZ=1000* setting may not be appropriate. In such an environment, the aggregate systems performance could benefit from setting the *HZ* parameter to a lower value (around 100).

Besides the static priority (*static_prio*), each task maintains an effective priority (*prio*). The distinction is being made to account for certain priority bonuses or penalties based on the recent sleep average (*sleep_avg*) of a given task. The sleep average accounts for the number of ticks a task was recently descheduled for. The effective priority of a task determines its location in the priority list (for the active as well as the expired array) of the run queue. A task is declared interactive when its effective priority exceeds its static priority by a certain level (a scenario that can only be based on the task accumulating *sleep average ticks*). The interactive estimator framework embedded into Linux operates automatically and transparently. In Linux 2.6, high priority tasks reach the interactivity state with a much smaller sleep average than do lower priority tasks. It is worthwhile to reemphasize that as the interactivity of a task is estimated via the sleep average, I/O bound tasks are potential candidates to reach the interactivity status, whereas CPU bound tasks are normally not perceived as interactive tasks. The actual timeslice can be defined as the maximum time a task can execute without yielding the CPU (voluntarily) to another task, and is simply a linear function of the static priority of normal tasks.

The priority itself is projected into a range of *MIN_TIMESLICE* to *MAX_ TIMESLICE*, where the default values are set to 10 and 200 milliseconds. The higher the priority of a task the greater the task's timeslice. For every timer tick, the running task's timeslice is decremented. If decremented to 0, the scheduler replenishes the timeslice, recomputes the effective priority and re-enqueues the task into either the active vector (if the task is classified as interactive) or into the expired vector (if the task is considered as being non-interactive). At this point, the system will dispatch another task from the active array. This scenario ensures that tasks in the active array will be executed first, before any expired task will have the opportunity to run again. If a task is descheduled, its timeslice will not be replenished at wakeup time, however it has to be pointed out that its *effective priority* might have changed due to any accumulated sleep time. If all the runnable tasks have exhausted their timeslice, and hence have been moved to the expired list, the expired and the active vector are swapped. This technique ensures that the scheduler *O(1)* does not have to traverse a (potentially large) list of tasks as it was required in the Linux 2.4 scheduler. The crux of the issue with the new design is that due to any potential interactivity, scenarios may arise where the active queue continues to have runnable tasks that are not migrated to the expired list. The ramification is that there may be tasks starving for CPU time. To circumvent the starvation issue, in the case the task that first migrated to the expired list is older than the *STARVATION_LIMIT* (which is set to 10 seconds), the active and the expired arrays are being switched.

While discussing performance enhancement techniques in the kernel, it has to be pointed out that the Linux 2.6 environment provides a NUMA (Non Uniform Memory Access) aware extension to the *O(1)* scheduler. The focus is on increasing the likelihood that memory references are local rather than remote on NUMA systems. The NUMA aware extension augments the existing CPU scheduler implementation via a node-balancing framework. Further, it is imperative to point out that next to the preemptible kernel support in Linux 2.6, the Native POSIX Threading Library (*NPTL*) represents the next generation POSIX threading solution for Linux, and hence has received a lot of attention from the performance community. The new threading implementation in Linux 2.6 has several major advantages, such as in-kernel POSIX signal handling. In a well-designed multi-threaded application domain, fast user space synchronization (*futex's*) can be utilized to implement the *pthread_mutex* object. In contrast to the Linux 2.4 *sched_yield* construct,

the *futex* framework avoids a scheduling collapse during heavy lock contention among different threads. The *NPTL* library is fully POSIX compliant.

I/O Scheduling and the BIO layer

The I/O scheduler in Linux forms the interface between the generic block layer and the low-level device drivers. The block layer provides functions that are utilized by the file systems and the virtual memory manager to submit I/O requests to block devices. These requests are transformed by the I/O scheduler and made available to the low-level device drivers. The device drivers consume the transformed requests and forward them (by using device specific protocols) to the actual device controllers that perform the I/O operations. As prioritized resource management seeks to regulate the use of a disk subsystem by an application, the I/O scheduler is considered an important kernel component in the I/O path. It is further possible to regulate the disk usage in the kernel layers above and below the I/O scheduler. Adjusting the I/O pattern generated by the file system or the virtual memory manager (VMM) is considered as an option. Another option is to adjust the way specific device drivers or device controllers consume and manipulate the I/O requests.

The various Linux 2.6 I/O schedulers can be abstracted into a rather generic I/O model. The I/O requests are generated by the block layer on behalf of threads that are accessing various file systems, threads that are performing raw I/O, or are generated by virtual memory management (VMM) components of the kernel, such as the kswapd or the pdflush threads. The producers of I/O requests initiate a call to *__make_request(),* which invokes various I/O scheduler functions such as *elevator_merge_fn*. The enqueuing functions in the I/O framework generally intend to merge the newly submitted block I/O unit (a *bio* in 2.6 or a *buffer_head* in 2.4) with previously submitted requests, and to sort (or sometimes just insert) the request into one or more internal I/O queues. As a unit, the internal queues form a single logical queue that is associated with each block device. At a later stage, the low-level device driver calls the generic kernel function *elv_next_request()* to obtain the next request from the logical queue. The *elv_next_request()* call interacts with the I/O scheduler's dequeue function *elevator_next_ req_fn*, and the latter has an opportunity to select the appropriate request from one of the internal queues. The device driver processes the request by converting the I/O submission into potential scatter-gather lists as well as protocol-specific commands that are submitted to the device controller. From an I/O scheduler perspective, the block layer is considered as the producer of I/O requests and the device drivers are labeled as the actual consumers.

From a generic perspective, every read or write request launched by an application results into either utilizing the respective I/O system calls or into memory mapping (*mmap*) the file into a process's address space. I/O operations normally result into allocating PAGE_SIZE units of physical memory. These pages are being indexed, as this enables the system to later locate the page in the buffer cache. Any cache subsystem only improves performance if the data in the cache is being reused. Further, the read cache abstraction allows the system to implement (file system dependent) read-ahead functionalities, as well as to construct large contiguous (SCSI) I/O commands that can be served via a single DMA operation. In circumstances where the cache represents pure (memory bus) overhead, I/O features such as direct I/O should be explored (especially in situations where the system is CPU bound).

In a general *write()* scenario, the system is not necessarily concerned with the previous content of a file, as a *write()* operation normally results into overwriting the contents in the first place. The write cache emphasizes other aspects such as asynchronous updates, as well as the possibility of omitting some write requests in the case multiple *write()* calls into the cache subsystem result into a single I/O operation to a physical disk. Such a scenario may occur in an environment where updates to the same (or a similar) inode offset are being processed within a rather short time-span. The block layer in Linux 2.4 was organized around the *buffer_head* data structure. The issue with this implementation is that it is a rather daunting task to create a truly effective and performance related block I/O subsystem if the underlying *buffer_head* structures force each I/O request to be decomposed into 4KB chunks. The new representation of the block I/O layer in 2.6 encourages large I/O operations. The block I/O layer now tracks data buffers by using *struct page* pointers. Linux 2.4 systems were prone to lose sight of the logical form of the writeback cache when flushing the cache subsystem. Linux 2.6 utilizes logical pages attached to inodes to flush dirty data.

This allows multiple pages (that belong to the same inode) to be coalesced into a single *bio* that can be submitted to the I/O layer (a process that works well if the file is not fragmented on disk).

The 2.4 Linux I/O scheduler

The default 2.4 Linux I/O scheduler primarily manages the disk utilization. The scheduler has a single internal queue, and for each new request, the I/O scheduler determines if the request can be merged with an existing request. If that is not the case, a new request is placed in the internal queue, which is sorted by the starting device block number of the request. This approach focuses on minimizing disk seek times. An aging mechanism implemented into the I/O scheduler limits the number of times an existing I/O request in the queue can be omitted and basically passed over by a newer request, which prevents any potential starvation scenarios. The *dequeue* function in the I/O framework represents a simple removal of requests from the head of the internal queue.

The 2.6 Deadline I/O Scheduler

The deadline I/O scheduler available in Linux 2.6 incorporates a per-request expiration based approach, and operates on five I/O queues. The basic idea behind the implementation is to aggressively reorder requests to improve I/O performance while simultaneously ensuring that no I/O request is being starved. More specifically, the scheduler introduces the notion of a per-request deadline, which is used to assign a higher preference to read than write requests. As already mentioned, the scheduler maintains five I/O queues. During the enqueue phase, each I/O request gets associated with a deadline, and is being inserted into I/O queues that are either organized by starting block (a sorted list) or by the deadline factor (a FIFO list). The scheduler incorporates separate sort and FIFO lists for read and write requests, respectively. The fifth I/O queue contains the requests that are to be handed off to the device driver. During a *dequeue* operation, in the case the dispatch queue is empty, requests are moved from one of the four I/O lists (sort or FIFO) in batches. The next step consists of passing the head request on the dispatch queue to the device driver (this scenario also holds true in the case that the dispatch-queue is not empty). The logic behind moving the I/O requests from either the sort or the FIFO lists is based on the scheduler's goal to ensure that each read request is processed by its effective deadline without actually starving the queued-up write requests. In this design, the goal of economizing on the disk seek time is accomplished by moving a larger batch of requests from the sort list (sector sorted) and balancing it with a controlled number of requests from the FIFO list. Hence, the ramification is that the deadline I/O scheduler effectively emphasizes average read request response time over disk utilization and total average I/O request response time.

To summarize, the basic idea behind the deadline scheduler is that all read requests are satisfied within a specified time period. On the other hand, write requests do not have any specific deadlines associated. As the block device driver is ready to launch another disk I/O request, the core algorithm of the deadline scheduler is invoked. In a simplified form, the first action being taken is to identify if there are I/O requests waiting in the dispatch queue, and if yes, there is no additional decision to be made on what to execute next. Otherwise, it is necessary to move a new set of I/O requests to the dispatch queue.

The scheduler searches for work in the following places, *BUT* will only migrate requests from the *first source* that results in a hit. (1) If there are pending write I/O requests, and the scheduler has not selected any write requests for a certain amount of time, a set of write requests is selected. (2) If there are expired read requests in the *read_fifo* list, the system will move a set of these requests to the dispatch queue. (3) If there are pending read requests in the sort list, the system will migrate some of these requests to the dispatch queue. (4) If there are any pending write I/O operations, the dispatch queue is populated with requests from the sorted write list. In general, the definition of *certain amount of time* for write request starvation is normally two iterations of the scheduler algorithm (this is tunable). After moving two sets of read requests, the scheduler will migrate some write requests to the dispatch queue. A set of requests (by default), can represent as many as 64 contiguous requests, but a request that requires a disk seek operation counts the same as 16 contiguous requests (this is tunable as well).

The 2.6 Anticipatory I/O scheduler

The anticipatory I/O scheduler's design attempts to reduce the per-thread read response time. It introduces a controlled delay component into the dispatching equation. The delay is being invoked on any new request to the device driver, thereby allowing a thread that just finished its I/O request to submit a new request. This basically enhances the chances (based on locality) that this scheduling behavior will result in smaller seek operations. The tradeoff between reduced seeks and decreased disk utilization (due to the additional delay factor in dispatching a request) is managed by utilizing an actual *cost-benefit* calculation method.

Overall, the 2.6 implementation of the anticipatory I/O scheduler is similar to, and may be considered as, an extension to the deadline scheduler. In general, the scheduler follows the basic idea that if the disk drive just operated on a read request, the assumption can be made that there is another read request in the pipeline, and hence it is worth the while to wait. The I/O scheduler starts a timer, and at this point, there are no more I/O requests passed down to the device driver. If a (close) read request arrives during the wait time, it is serviced immediately, and in the process, the actual distance that the kernel considers as *close* grows as time passes (the adaptive part of the heuristic). Eventually the *close* requests will dry out and the scheduler will hence decide to submit some of the write requests, basically converging back to what is considered as a normal I/O request dispatching scenario. In general, the anticipatory I/O scheduler is considered as a rather complex implementation, as the actual cost-benefit analysis that is necessary to provide adequate performance has to be conducted in an as efficient as possible manner.

The 2.6 Completely Fair Queuing Scheduler

The Completely Fair Queuing (CFQ) I/O scheduler can be considered as representing an extension to the better known Stochastic Fair Queuing (SFQ) scheduler implementation. The focus of both implementations is on the concept of *fair allocation of I/O bandwidth* among all the *initiators of I/O requests*. A SFQ based scheduler design was initially proposed for some network subsystems. The goal to be accomplished is to distribute the available I/O bandwidth as equally as possible among the I/O requests. The actual implementation utilizes *n* (normally 64) internal I/O queues, as well as a single I/O dispatch queue. During an enqueue operation, the *PID* of the currently running process (the actual I/O request producer) is utilized to select one of the internal queues (normally hash based) and hence, the request is basically inserted into one of the queues (in FIFO order). During dequeue, the SFQ design calls for a round-robin based scan through the non-empty I/O queues, and basically selects requests from the head of the queues. To avoid encountering too many seek operations, an entire round of requests is first collected, and secondly sorted and ultimately merged into the dispatch queue. Next, the head request in the dispatch queue is passed to the actual device driver.

A standard CFQ implementation on the other hand does not utilize a hash function. Hence, each I/O process receives an internal queue assigned to, which implies that the number of I/O processes determines the number of active internal queues. From an aggregate performance perspective, the CFQ scheduling concept works efficiently well under the circumstances that all I/O processes generate requests at basically the same rate. In Linux 2.6, the CFQ I/O scheduler utilizes a hash function (and a certain amount of request queues) and therefore resembles an SFQ implementation. The CFQ, as well as the SFQ implementations strives to manage per-process I/O bandwidth, and provide fairness at the level of process granularity.

The 2.6 noop I/O scheduler

The Linux 2.6 noop I/O scheduler can be considered as a rather minimal I/O scheduler that performs (as well as provides) basic merging and sorting functionalities. The main usage of the noop scheduler revolves around non disk-based block devices (such as memory devices), as well as specialized software or hardware environments that incorporate their own I/O scheduling and (large) caching

functionality, and hence require only minimal assistance from the kernel. Hence, in large-scale I/O configurations that incorporate RAID controllers and a vast number of contemporary physical (TCQ capable) disk drives, the noop scheduler has the potential to outperform the other three I/O schedulers (workload dependent).

Conclusion and Future Work

The Linux 2.6 kernel represents another evolutionary step forward, and builds upon its predecessors to boost (application) performance, through enhancements to the VM subsystem, the CPU scheduler and the I/O scheduler. In addition, this new version of the kernel delivers important functional enhancements in security, scalability, and networking.

References

"The Linux 2.6 Source Code" http://kernel.org

Arcangeli, A., "Evolution of Linux Towards Clustering", EFD R&D Clamart, 2003

Axboe, J., "Deadline I/O Scheduler Tunables", SuSE

Corbet, J., "A new deadline I/O scheduler". http://lwn.net/Articles/10874

Corbet, J., "Anticipatory I/O scheduling". http://lwn.net/Articles/21274

Corbet, J., "Porting drivers to the 2.5 kernel", LSO, 2003

Iyer, S., Drushel, P.,"Anticipatory Scheduling, A disk scheduling framework to overcome deceptive idleness in synch. I/O", Rice U.

Lee Irwin III, W., "A 2.5 Page Clustering Implementation", Linux Symposium, Ottawa, 2003

Patterson D.A, , Hennessy, J.L, "Computer Architecture – A Quantitative Approach" , 3d Edition, 2002

Nagar, S., Franke, H., Choi, J., Seetharaman, C., Kaplan, S., Singhvi, N., Kashyap, V., Kravetz, M., "Class-Based Prioritized Resource Control in Linux", 2003 Linux Symposium,

McKenney, P., "Stochastic Fairness Queueing", INFOCOM, 1990

Molnar, I., "Goals, Design and Implementation of the new ultra-scalable O(1) scheduler". (sched-design.txt).

Mosberger, D., Eranian, S., "IA-64 Linux Kernel, Design and Implementation", Prentice Hall, NJ, 2002

Wienand, I., "An analysis of Next Generation Threads on IA64", HP, 2003

Drepper, U. and I. Molnar, The Native POSIX Thread Library for Linux, 01/30/03

Franke, H., Russell, R. and M. Kirkwood, "Fuss Futex and Furwocks: Fast User level Locking in Linux", OLS 2003