Comparative evaluation of the recent Linux and Solaris kernel architectures

STERGIOS PAPADIMITRIOU
TEI of Kavala
Dept of Information Management
Agios Loukas, 65404 Kavala
GREECE

KONSTANTINOS TERZIDIS
TEI of Kavala
Dept of Information Management
Agios Loukas, 65404 Kavala
GREECE

Abstract: The paper compares core kernel architecture and functionality of two modern open source systems. The subsystems examined are scheduling, memory management, and file system architecture. These subsystems are common to any operating system (not just Unix and Unix-like systems), and they tend to be the most well-understood components of the operating system. One of the more interesting aspects concerning the Linux and Solaris Operating Systems (OS), is the amount of similarities between them. Ignoring the different naming conventions, both of them utilize similar approaches toward implementing the different concepts. Each OS supports time-shared scheduling of threads, demand paging with a not-recently-used page replacement algorithm, and a virtual file system layer to allow the implementation of different file system architectures. The paper concludes that both the Linux and the OpenSolaris kernel can offer robust and powerful computing environments both at the server application areas and as well at the desktop and workstation ones.

Key–Words: Scheduling, memory management, threads, file systems, paging

1 Introduction

In this paper we examine comparatively three core subsystems of two modern open source operating systems. The first one is the GNU-Linux operating system [3, 6]. The current 2.6 Linux kernel incorporates many advanced features [4, 5] and stands well compared to the also state of the art OpenSolaris kernel [1, 2, 12].

The three subsystems examined are *scheduling*, *memory management*, and *file system* architecture. These subsystems are common to any operating system (not just Unix and Unix-like systems), and they tend to be the most well-understood components of the operating system.

An interesting aspect revealed by our comparison is the amount of similarities between the two OSes. Even with different naming conventions, each OS takes fairly similar paths toward implementing the different concepts. Also the performance, scalability and robustness of the two systems are at similar levels. Both systems offer strong computing environments, capable of supporting demanding applications with similar performances, as for example illustrated in [10].

Both Linux and Solaris support time-shared scheduling of threads and state of the art schedulers [1, 3]. They can support effectively both batch, interactive and even real-time processing demands. Also, they offer extensive symmetric multiprocessing support and both own fully preemptive kernels. At the area of memory management both OSes implement demand paging with a not-recently-used page replacement algorithm [6, 7]. Also both Linux and OpenSolaris implement a virtual file system layer to allow the implementation of different file system architectures. Ideas that originate in one OS often find their way into the other. For instance, Linux also uses the concepts behind Solaris's slab memory allocator. The recent open sourcing of the OpenSolaris kernel code by Sun Microsystems, offers the potentiality for fruitful interchange of ideas between the Linux and OpenSolaris communities. The paper concludes that both systems are modern and effective UNIX realizations, capable of accomplishing effectively demanding application requirements and at the same time their UNIX philosophy offers to them many similarities [14].

The paper proceeds as follows: Section 2 studies the scheduling approaches of the two systems and compares them. Section 3 deals with their memory management systems and it takes the same comparative approach. The file system frameworks are discussed in section 4. Finally, the paper concludes with the results of this comparative study.

2 Scheduling and Schedulers

The basic unit of scheduling in Solaris is the *kthread_t* structure [1]; and in Linux, the *task_struct* structure [3]. Solaris represents each process as a *proc_t*, and each thread within the process has a *kthread_t*. Linux represents both processes and threads by *task_struct* structures. A single-threaded process in Linux has a single *task_struct*. A single-threaded process in Solaris has a *proc_t*, a single *kthread_t*, and a *klwp_t* structure. The *klwp_t* structure provides a save area for threads switching between user and kernel modes. Effectively, both operating systems schedule threads. In Linux a thread corresponds to a *task_struct* structure and in Solaris a thread is a *kthread_t*.

Scheduling decisions are based on *priority*. In Linux, the lower the priority value, the better, i.e. a value closer to 0 represents a higher priority. In Solaris, the higher the value, the higher the priority. For Linux, the priority range *0 - 99* corresponds to the *System Threads, Real-Time* Scheduling Class (*SCHED_FIFO, SCHED_RR*) and the priority range *100 - 139* to *User priorities* (*SCHED_NORMAL*). In Solaris, the priority range *0 - 59* corresponds to the *Time Shared, Interactive, Fixed, Fair Share Scheduler* class, the range *60-99* to the *System Class*, the range *100-159* to the *Real-Time* and finally the highest priority range *160-169* to the *Low level Interrupts* [3, 1].

Both OSes favor interactive threads/processes. Interactive threads run at better priority than compute-bound threads, but tend to run for shorter time slices. Both Solaris and Linux use a per-CPU *runqueue*.

Linux uses an *active* queue and an *expired* queue. Threads are scheduled in priority from the active queue. A thread moves from the active queue to the expired queue when it uses up its time slice (and possibly at other times to avoid starvation). When the active queue is empty, the kernel swaps the active and expired queues.

Solaris uses a *dispatch queue* per CPU. If a thread uses up its time slice, the kernel gives it a new priority and returns it to the dispatch queue.

The *runqueues* for both operating systems have separate linked lists of runnable threads for different priorities. Both Solaris and Linux use a *separate list* for each priority. Linux uses an arithmetic calculation based on run time versus sleep time of a thread (as a measure of *interactiveness*) to arrive at a priority for the thread. Solaris performs a table lookup.

Both OSes schedule the one next thread to run, instead of attempting to derive a schedule for a whole group of n threads. Also, both have mechanisms to take advantage of caching (warm affinity) and load balancing. For hyperthreaded CPUs, Solaris has a mechanism to help keep threads on the same CPU node. This mechanism is under control of the user and application.

One of the big differences between Solaris and Linux is the capability to support multiple *scheduling classes* on the system at the same time. Both OSes support Posix *SCHED_FIFO*, *SCHED_RR*, and *SCHED_OTHER* (or *SCHED_NORMAL*). *SCHED_FIFO* and *SCHED_RR* typically result in *realtime* threads. Both Solaris and Linux implement kernel preemption in support of realtime threads. Solaris has support for a *fixed priority* class, a *system class* for system threads (such as page-out threads), an *interactive* class used for threads running in a windowing environment under control of the X server, and the *Fair Share Scheduler* in support of resource management.

The ability to add new scheduling classes to the system comes with a price. Everywhere in the kernel that a scheduling decision can be made (except for the actual act of choosing the thread to run) involves an indirect function call into scheduling class-specific code. For instance, when a thread is going to sleep, it calls scheduling-class-dependent code that does whatever is necessary for sleeping in the class. On Linux the scheduling code simply does the needed action. There is no need for an indirect call. The extra layer means there is slightly more overhead for scheduling on Solaris, but more supported features.

3 Memory Management and Paging

In Solaris, every process has an "address space" made up of logical section divisions called *segments*. The segments of a process address space are viewable via pmap(1). Solaris divides the memory management code and data structures into *platform-independent* and *platform-specific* parts. The platform-specific portions of memory management is in the HAT, or *Hardware Address Translation*, layer [7, 1].

Linux uses a memory descriptor that divides the process address space into logical sections called *memory areas* to describe process address space. Linux also has a *pmap* command to examine process address space. Linux divides machine-dependent layers from machine-independent layers at a much higher level in the software. On Solaris, much of the code dealing with, for instance, page fault handling is machine-independent. On Linux, the code to handle page faults is pretty much machine-dependent from the beginning of the fault handling. A consequence of this is that Linux can handle much of the paging code more quickly because there is less data abstraction (layering) in the code. However, the cost is that a change in the underlying hardware or model

requires more changes to the code [3, 5]. Solaris isolates such changes to the HAT and pmap layers respectively. Segments, regions, and memory areas are delimited by:

- Virtual address of the start of the area.
- Their location within an object/file that the segment/region/memory area maps.
- Permissions.
- Size of the mapping.

For instance, the text of a program is in a segment/region/memory area. The mechanisms in the two OSes to manage address spaces are very similar, but the names of data structures are completely different.

Both operating systems use a variation of a Least Recently Used (LRU) algorithm for page stealing/replacement. They both have a daemon process/thread to do page replacement. Solaris has a *pageout daemon* that runs periodically and in response to low-free-memory situations. Paging thresholds in Solaris are automatically calibrated at system startup so that the daemon does not overuse the CPU or flood the disk with page-out requests.

Linux also uses an LRU algorithm that is dynamically tuned while it runs. On Linux, there can be *multiple kswapd daemons*, as many as one per CPU. Both OSes use a global working set policy (as opposed to per process working set). Linux uses several page lists for keeping track of recently used pages. The different linked lists of pages to facilitate an LRU-style algorithm. Linux divides physical memory into (possibly multiple sets of) three ”zones:” one for DMA pages, one for normal pages, and one for dynamically allocated memory. These zones seem to be very much an implementation detail caused by x86 architectural constraints. Pages move between ”hot”, ”cold” and ”free” lists. Frequently accessed pages will be on the ”hot” list. Free pages will be on the ”cold” or ”free” list.

Solaris uses a *free list*, *hashed list*, and *vnode page list* to maintain its variation of an LRU replacement algorithm. Instead of scanning the vnode or hash page lists, Solaris scans all pages with a *”two-handed clock”* algorithm. The two hands stay a fixed distance apart. The front hand ages the page by clearing reference bit(s) for the page. If no process has referenced the page since the front hand visited the page, the back hand will free the page (first asynchronously writing the page to disk if it is modified). Both operating systems take NUMA locality into account during paging. The I/O buffer cache and the virtual memory page cache is merged into one system page cache on both OSes. The system page cache is used for reads/writes of files as well as mmapped files and text and data of applications.

Although the memory management systems of the two systems are similar, there are some subtle differences. A brief example to highlight differences is page fault handling. In Solaris, when a page fault occurs, the code starts in a platform-specific trap handler, then calls a generic as_fault() routine. This routine determines the segment where the fault occurred and calls a ”segment driver” to handle the fault. The segment driver calls into file system code. The file system code calls into the device driver to bring in the page. When the page-in is complete, the segment driver calls the HAT layer to update page table entries (or their equivalent). On Linux, when a page fault occurs, the kernel calls the code to handle the fault. You are immediately into platform-specific code. This means the fault handling code can be quicker in Linux, but the Linux code may not be as easily extensible or ported.

4 File Systems

The *data abstraction layer* in both Linux and Solaris hide file system implementation details from applications. In both OSes, the developer uses *open*, *close*, *read*, *write*, *stat*, etc. system calls to access files, regardless of the underlying implementation and organization of file data. Solaris calls this mechanism *VFS* (”virtual file system”) and the principle data structure is the *vnode*, or ”virtual node.” [11] Every file being accessed in Solaris has a vnode assigned to it. In addition to generic file information, the vnode contains pointers to file-system-specific information.

Linux also uses a similar mechanism, also called *VFS* (for *virtual file switch*). In Linux, the file-system-independent data structure is an *inode*. This structure is similar to the *vnode* on Solaris [8]. Note that there is an inode structure in Solaris, but this is file-system-dependent data for UFS file systems. Linux has two different structures, one for *file operations* and the other for *inode operations*. Solaris combines these as *vnode operations*.

VFS allows the implementation of many file system types on the system. This means that there is no reason that one of these operating systems could not access the file systems of the other OSes. Of course, this requires the relevant file system routines and data structures to be ported to the VFS of the OS in question. Both OSes allow the stacking of file systems.

5 Conclusions

Solaris, and Linux are obviously benefiting from each other. With Solaris going open source, we can expect this to continue at a faster rate. Solaris uses more data abstraction layering, and generally could support additional features quite easily because of this. However, most of the layering in the kernel is undocumented. Probably, source code access will change this.

The application level interface to these systems is very similar, typical of the modern UNIX system programming interface [12, 13]. The recent open sourcing of the OpenSolaris kernel code by Sun Microsystems, offers the potentiality for valuable interchange of ideas between the Linux and OpenSolaris communities. The paper concludes that both systems are modern and effective UNIX realizations. They are capable of accomplishing effectively demanding application requirements and at the same time their UNIX philosophy offers to them many similarities [14].

References:

[1] Richard McDougall and Jim Mauro, *Solaris(TM) Internals: Solaris 10 and OpenSolaris Kernel Architecture (2nd Edition) (Solaris Series)*, Sun Microsystems Press, 2006

[2] Richard McDougall, Jim Mauro, Brendan Gregg, *Solaris Performance and Tools*, Sun Microsystems Press, 2006

[3] Daniel Plerre Bovet and Marco Cesati, *Understanding the Linux Kernel*, O' Reilly, 2005

[4] Christian Benvenuti, *Understanding LINUX Network Internals*, O' Reilly, 2005

[5] Jonathan Corbet, Alessandro Rubini, Creg Kroah-Hartman, *LINUX Device Drivers*, 3nd Edition, O'Reilly 2005

[6] Linux Kernel Development (2nd Edition) (Novell Press) by Robert Love (Paperback - Jan 12, 2005)

[7] J. L. Bertoni, *Understanding Solaris Filesystems and Paging*, Technical Report TR-98-55, Sun Microsystems Research, November 1998, http://research.sun.com /research/techrep/1998/abstract-55.html.

[8] Rmy Card, Theodore Tso, Stephen Tweedie, *Design and Implementation of the Second Extended Filesystem*, First Dutch International Symposium on Linux. Amsterdam (December, 1994)

[9] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. [PDF] A Fast File System for UNIX *ACM Transactions on Computer Systems*, 2(3):181-97, August 1984.

[10] *MySQL AB New OLTP Benchmark Results for MySQL on Solaris 10*, April 21, 2006, URL: http://www.mysql.com/news-and-events/press-release/release_2006_19.html

[11] Sun Microsystems File System Performance: The Solaris OS, UFS, Linux ext3, and ReiserFS, www.sun.com/software/whitepapers/ solaris10/fs_performance.pdf, 2004

[12] Rich Teer, *Solaris System Programming*, Addison-Wesley, 2005

[13] Stevens, W. Richard, Fenner, Bill, and Rudoff, Andrew M. 2004, *UNIX Network Programming, Volume 1, Third Edition, The Sockets Networking API*, Addison-Wesley, Reading, MA

[14] Vahalia, U., *UNIX Internals - The New Frontiers*, Prentice Hall, 1996