A Comparative Evaluation of Core Kernel Features of the Recent Linux, FreeBSD, Solaris and Windows Operating Systems

Stergios Papadimitriou, Lefteris Moussiades

***Abstract*—The paper compares core kernel architecture and functionality of four modern operating systems. The subsystems examined are process / thread architecture, scheduling and interrupt handling. Linux, Solaris and FreeBSD have a lot of similarities, owning Unix roots, but also have some notable differences. However, Windows is significantly different, being a radical non-Unix design. The paper compares some aspects of the Unix-like approaches of Linux/Solaris/FreeBSD with Windows, emphasizing the consequences of their different design decisions, and presents some comparative performance results, using Java benchmarks.**

***Index Terms*—Scheduling, process, threads, interrupts**

I. Introduction

In this work we examine comparatively core subsystems of three modern Unix-like operating systems and contrast them with the radical non-Unix approaches of Windows. The first one is the GNU-Linux operating system [3], [6]. The current 3.x Linux kernel incorporates many advanced features [4], [5] and stands well compared to the also state of the art Solaris 11 kernel [1], [2], [7] and the FreeBSD kernel [11]. Also, some aspects of modern Windows (Windows 7) [10] are compared with Unix designs.

The subsystems examined are *process* and *thread* structure, *interrupt structure* and *scheduling*. These subsystems form the core components of any operating system.

The Linux, Solaris and FreeBSD OSes take fairly similar paths toward implementing the different concepts. Also the performance, scalability and robustness of the three systems are at similar levels. Both systems offer strong computing environments, capable of supporting demanding applications with similar performances.

Both Linux, Solaris, FreeBSD and Windows support preemptive scheduling of threads with 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 own fully preemptive kernels. [6], [11].

The paper concludes that Linux, FreeBSD and Solaris 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 [9]. Linux seems the most efficient, both from the point of view of the user experience (which is subjective) and from actual performance results. This efficiency can be explained from the fact that Linux generally uses less intermediate layers of code before accessing hardware. Windows implements different design solutions, and it is interested to study them and explore their benefits and drawbacks.

The paper proceeds as follows: Section II highlights some important aspects of the basic architecture of these systems. Section III exploits the interrupt processing architecture that is of outstanding importance for the efficient support of higher layers and for meeting real-time processing demands. Section IV studies the scheduling approaches of the four systems and compares them. Finally, the paper concludes with the results of this comparative study.

Manuscript received Oct 02, 2015; revised Nov 04, 2015.

Stergios Papadimitriou and Lefteris Moussiades are with the Technology Education Institute of East Macedonia and Thraki, Dept of Informatics and Computer Engineering, Agios Loukas, 65404 Kavala, GREECE, emails: sterg@teiemt.gr, lmous@teiemt.gr

II. Basic Architecture

Both Linux, Solaris, FreeBSD and Windows are monolithic kernels. This organization permits efficient execution since kernel components interact with direct procedure calls without involved with message passing and the associated switches between user and kernel mode code. Windows started as a microkernel design [10], but performance demands forced the movement of components designed to operate in user mode (for benefit of reliability and modularity) again in kernel space. The recent Windows kernel has a modular design with distinct kernel components in different modules. However, all the modules operate in kernel space and therefore routines of one module can directly call routines from other modules. This way, Windows kernel avoids a great deal of overhead (as also Linux, FreeBSD and Solaris kernels). The modular but yet monolithic design of Windows kernel is referred sometimes as *macrokernel* design [10].

In contrast to Linux/FreeBSD/Solaris, Windows presents a client-server view of the operating system to applications. Applications interact with a subsystem, mostly with the Windows subsystem process (*Crss*). That subsystem keeps the *Executive Process* (*EPROCESS*) block, the *Process Environment Block (PEB)*, and a parallel structure for each process that is executing a Windows program.

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 kernel state for user threads, i.e. is a save area for threads switching

ISBN: 978-988-19253-0-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

between user and kernel modes. Effectively, all operating systems schedule threads. In Linux a thread corresponds to a *task_struct* structure and in Solaris a thread is a *kthread_t*. Multithreaded processes in Linux are implemented as normal processes (i.e. with *task_struct* structures) that share the same address space and resources. Linux also implements a clever scheme for efficiently identifying the current process descriptor by maintaining the proper pointer at the current stack of the processor.

FreeBSD like Solaris implements a multithreaded process design [11]. Each FreeBSD process keeps a linked list of its threads. These threads are scheduled by the kernel, and they own their own kernel stacks onto which they can execute system calls simultaneously. The process state in FreeBSD supports threads that can select the set of resources to be shared, i.e. the concept of *variable-weight* processes is implemented [11]. Linux implements also variable-weight processes, but with a different mechanism, by allowing to specify with the *clone* system call, which resources the newly created tasks will share.

The thread structure of FreeBSD represents just the information needed to run in the kernel (as the *klwp_t* of Solaris does): information about scheduling, a stack to use when running in the kernel, a *thread state block*, and other machine dependent state. FreeBSD elegantly divides the kernel state in two primary structures: the *process structure* and the *thread structure*. The process structure contains information that must always remain resident in main memory, whereas the thread structure tracks information that needs to be resident, only when the process is executing, such as its kernel runtime stack.

FreeBSD has the *rfork* system call that behaves like Linux's *clone* system call. Also it organizes sleep and turnstile queues in a data structure that is hashed by an event identifier. FreeBSD assigns bottom-half interrupt priorities and top-half interrupt priorities. It assigns for each thread a user mode execution priority and a kernel mode execution priority and separates wait channel priority from user mode priority. Also, it assigns threads sleeping in the kernel a higher priority, because they typically hold shared kernel resources when they awakened. FreeBSD provides restartable system calls.

FreeBSD initially assigns a high execution priority to each thread and allows that thread to execute for a fixed *time slice*. Threads that execute for the duration of their time slice have their priority lowered, whereas threads that give up the CPU (usually because they perform I/O) are allowed to remain at their priority. Inactive threads have their priority raised.

Windows implements threads distinctly, as Solaris and FreeBSD do, and not as Linux that views threads as normal processes with shared resources. Windows processes are represented by an *executive process* (*EPROCESS*) block. Threads are represented by *executive thread* (*ETHREAD*) blocks. The EPROCESS is a multithreaded process (like Solaris *proc_t*), therefore it links to one or more ETHREAD (like Solaris *kthread_t*) blocks. The part of the EPROCESS structure that needs to be accessed from user-mode code, is isolated to the *Process Environment Block* (*PEB*) (like Solaris *klwp_t*) that exists in the process address space. Thus, PEB in Windows plays a similar role as *klwp_t* of Solaris, i.e. a glue for switching user and kernel modes of a process.

The following GroovyLab code exercises the efficiency of the underlying multithreading support (GroovyLab, is a MATLAB like environment that compiles Groovy like code for the Java Virtual Machine. The project's site is: https://code.google.com/p/jlabgroovy/).

Listing 1: A benchmark for testing multithreading performance

```
  tic()
  sM=100;   nthreads=1000000
  java.util.concurrent.Future <?>[] futures =
    new java.util.concurrent.Future [nthreads];
    // how many rows the thread processes
  int   rowsPerThread = (int)(sM / nthreads)+1
  int threadId = 0;  // the current threadId
  while (threadId < nthreads)  {  // for all threads

 futures[threadId] =
  edu.emory.mathcs.utils.ConcurrencyUtils.submit(
    new Runnable() {
          public void run()  {
    double sm=0.0
    for (k in 1..1000) sm+=k
   }
 });
        threadId++;
  }  // for all threads
ConcurrencyUtils.waitForCompletion(futures);
  tm=toc()
   }
  }
```

The recent Linux kernel 3.11.6 on which OpenSuSE 13.1 is based outperformed clearly Windows 8 at the above benchmark. Linux presented an execution time of 7.1 sec, while Windows finished at 17.1 sec. We performed some experiments with different number of threads, and Linux finishes at about half the time, persistently at all the experiments.

We performed also experiments to test the input / output efficiency, using both the traditional Java I/O classes and with the NIO (New IO) Java framework for asynchronous I/O. At the plain input streams and at the random access files tests, Linux outperformed Windows significantly. However, at the buffered input streams and at the memory mapped files, the performance of the two operating systems is similar. Actually, at the buffered input stream benchmark with Java 8, Windows outperforms Linux. We present two representative tables, one with results obtained with Java 7 (Table 1) and one with the recent Java 8 (Table 2).

Table 3 presents some results on multithreading performance of the OSes. It displays the user and system (sys) time for the corresponding tests (rows) and operating systems (columns). Linux performs faster at this bench and FreeBSD follows closely. Windows performs much slower than the other OSes.

The *thread creation* benchmark (row 1 of Table 3) creates repetitively a large number of threads (specifically one million, i.e. 1000000). Each thread performs a trivial task, actually it increases a global variable with a non-thread safe increment operation. As expected the final value of the counter depends on race conditions, and takes non-deterministically a value near 1000000, actually $<= 1000000$.

The next row presents a similar benchmark for thread creation. The only difference is that the counter increment operation is protected with a mutex. Therefore, the total

ISBN: 978-988-19253-0-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

number of iterations is now reliably correct, i.e. is always 1000000. Surprisingly, execution times at Linux at that bench that implements critical section is a little faster than the increment of the shared counter in the wrong way without mutex.

The *memory map* benchmark (third row) maps a file over the virtual memory space. Linux performs again faster. At the *memory allocation* benchmark (fourth row), also Linux is the most efficient OS in allocating heap space, with the classic malloc() C library function.

The *forkWithNoWrite* benchmark exercises the efficiency of fork based process creation. Since Windows lacks a fork() call, only Unix-like OSes are tested. The bench creates a large number of forked processes (specifically 100000) that do not perform any computation. The *forkWithWrite* bench is similar with the only difference that these processes increment a global counter. Again, Linux is clearly the faster Unix-like OS. Also, perhaps it is interesting to note that the *clang* Linux C/C++ compiler (based on the Low Level Virtual Machine (LLVM) intermediate reprentation), performs slightly faster than the standard gcc/g++ compiler, although the difference is not significant. We whould note here that Linux presents very small user times and generally Linux process forking outperforms clearly the other OSes. However, when it comes to implement multithreading for applications, with frameworks as pthreads, Java or C++11, the other OSes use their specialized thread structures. Thus, the difference in the efficiency of thread support illustrated at the *Thread Creation* and *Thread Creation and mutex* benchmarks between Linux and FreeBSD is not significant, actually FreeBSD has slightly smaller system times.

III. Interrupt Structure

Modern networking imposes significant demands for efficient processing of interrupts from network cards that can service very high transfer rates. Also, many "Internet of Things" style applications further demand efficient interrupt processing. This section highlights some important aspects of the rather advanced interrupt architecture offered by all the OSes.

Although interrupt controllers perform a level of interrupt prioritization, Windows imposes its own interrupt priority scheme, on top of the interrupt prioritization performed by interrupt controllers, known as *interrupt request levels (IRQLs)*. The IRQLs aren't the same as the interrupt requests (IRQs) defined by interrupt controllers. The kernel represents IRQLs internally as a number from 0 through 31 on x86 and from 0 to 15 on x64 and IA64. Higher numbers represent higher priority interrupts. Although the kernel defines the standard set of IRQLs for software interrupts, the Hardware Abstraction Layer (HAL) maps hardware interrupt numbers to the IRQLs.

Windows uses a type of device driver called a *bus driver*, to determine the presence of devices on a bus type (e.g. PCI, USB etc.) and what interrupts can be assigned to a device. The bus driver reports this information to the *Plug and Play manager* which decides, after taking into account the acceptable interrupt assignments for all other devices, which interrupt will be assigned to each device. Then it calls a *Plug and Play interrupt arbiter*, which maps interrupts to IRQLs.

Both Windows and Linux use a system table called *Interrupt Descriptor (or Dispatch) Table (IDT)*. The IDT associates each interrupt or exception vector with the address of the corresponding interrupt or exception handler. The IDT must be properly initialized before the kernel enables interrupts. At system boot the IDT is filled with pointers to the kernel routines that handle each interrupt and exception. Each processor in Windows has a separate IDT, so that different processors can run different ISRs, if appropriate. Windows statically assigns interrupts to processors in a round-robin manner.

A common practice is to store all the context required to resume a nested kernel control path in the kernel mode stack of the current process. With this design, we cannot reschedule from an interrupt handler. This is because interrupts can be arbitrarily nested, stacking multiple contexts on the kernel stack. If we reschedule, the stack with the saved interrupt frames is lost, and there is a problem at restoring those contexts after the interrupt handler finishes.

Windows and Linux use the kernel stack of the current process for interrupt frames and therefore they cannot reschedule from an interrupt handler. Windows uses the kernel stack of the currently executing thread for servicing interrupts. Linux does the same when two-page stacks (i.e. 8Kb) are used. However, for one page stack (i.e. 4 Kb) Linux uses separate *exception*, *hard IRQ stack* and *soft IRQ* stacks. These are multiple kernel mode stacks: for exceptions (one per-process), for hard IRQ (one per CPU) and for soft IRQs (one per CPU). Making a difference from Windows, Linux implements an interrupt model without priority levels. Because each interrupt handler may be deferred by another one, there is no need to establish additional predefined priorities among hardware devices beyond those defined by the hardware interrupt arbiter. This simplifies the kernel code and improves its portability.

Also, for Windows, code running at *Deferrable Procedure Call (DPC)* dispatch level or above can't wait for an object. That wait operation would necessitate the scheduler to select another thread to execute. However, the scheduler synchronizes its data structures at DPC dispatch level and cannot therefore be invoked to perform a reschedule.

Linux implements an elaborate scheme to balance the interrupt load among the processors at a multiprocessor environment. Specifically, a special kernel thread, the *kirqd*, exploits the IRQ affinity of a CPU by modifying the *Interrupt Redirection Table* entries of the I/O APIC. The later table controls the distribution of interrupt processing among CPUs. Therefore, Linux obtains generally a balanced load of the interrupt workload over multiprocessors.

Linux provides three different and complementary ways to perform deferred processing:

1) *Softirqs*
2) *Tasklets*
3) *Work queues*

Softirqs is the most efficient mechanism. They are statically allocated (i.e. defined at compile time). They can run concurrently on several CPUs, even if they are of the same type. Therefore, to fullfill these concurrency requirements, they must be *reentrant* functions and must explicitly protect their data structures with spin locks.

ISBN: 978-988-19253-0-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

Tasklets are built upon the softirq mechanism but they can be dynamically allocated (i.e. at runtime). That makes them proper for use with dynamically loaded kernel modules. Contrary, to softirqs, they can be *non-reentrant* functions. Therefore, they are easier to write, but they are imposed to serialization, i.e. tasklets of the same type cannot run in parallel on several CPUs. However, tasklets of different types can.

Both softirqs and tasklets execute in *interrupt context*, therefore they cannot sleep. This imposes significant limitation on what the programmer can perform within a softirq-tasklet.

The *work queue* mechanism was introduced in Linux to overcome this limitation. Work queues schedule deferrable work within the process context of special kernel threads. Thus this deferable code, is schedulable, and can therefore sleep. However, work queues induce the overhead of scheduling and of the necessary context switches. Softirqs and tasklets are a more efficient mechanism since they are executed totally within kernel space, as simple C routine calls. We note that although work queues imply a context switch, that context switch is efficient, since switching takes place totally within kernel space.

Practically, work queues are quite efficient, but when we have to service thousands of interrupts per second, using tasklets or even the more efficient softirqs makes sense.

Linux also supports an elaborate distribution of IRQ signals coming from the hardware devices in a round-robin fashion among all the CPUs [3]. The Linux approach to interrupt handling stays closer to hardware from the other OSes. A competent C programmer can readily write interrupt handlers without having to familiarize with complicated layering e.g. the ”interrupt objects” of Windows. Linux implements an interrupt architecture that although rather low-level, it exploits fully the parallelism on an SMP system and is simple to use.

The Linux kernel delivers exceptions caused by programming errors to the application as Unix style signals.

Solaris uses the notion of the *interrupt thread* for servicing interrupts. This has similarities to the *work queues* of Linux. Upon recept of the interrupt, the current thread is pinned and the interrupt thread executes. When the interrupt thread completes, the interrupted thread is unpinned and resumes execution. This mechanism avoids a full context switch for servicing interrupts. Solaris interrupt threads can sleep also: If the interrupt thread blocks, it is given full thread state and it is placed on a sleep queue, and the interrupted thread will be unpinned. Thus, for interrupt code that doesn't sleep, Solaris interrupt threads can be more efficient than Linux work queues.

The term *pinned* refers to a mechanism employed by the Solaris kernel that avoids context switching out the interrupted thread. The executing thread is pinned under the interrupt thread. The interrupt thread ”borrows” the LWP from the executing thread. While the interrupt handler is running, the interrupted thread is pinned to avoid the overhead of having to completely save its context; it cannot run on any processor until the interrupt handler completes or blocks on a synchronization object. Once the handler is complete, the original thread is unpinned and rescheduled.

Solaris (as Windows also) assigns priorities to interrupts to allow overlapping interrupts to be handled with the correct precedence. This interrupt prioritization performs similarly to the IRQL based prioritization performed by the Windows kernel. We should note that Linux does not prioritize interrupts explicitly, but by allowing nesting of interrupts enforces the interrupt prioritization that the interrupt controller implements. However, interrupt handlers using priority levels alone cannot block, since a deadlock could occur if they are waiting on a resource held by a lower priority interrupt.

Solaris supports the *cross-call mechanism* for multiprocessor systems. It is a facility with which one CPU can send an interrupt to one or more other CPUs on the system (or to all of them) to force the CPU into a handler to take a specific action.

FreeBSD implements thread context for interrupt processing and fits it within the prioritization of the rest tasks of the workload. It uses also high-priority *interrupt threads* as Solaris does. The highest priority threads of class *ITHD* serve the time critical demands for interrupt processing, tasks that on single processor systems were usually performed within the interrupt service routine by disabling the CPU interrupts. Following the ITHD scheduling class, comes in priority the *REALTIME* class. Therefore, with the prerequisite of small and bounded overhead for the ITHD thread processing, the engineer can design and implement on FreeBSD real time processing workloads. The *KERN* class follows the REALTIME class in priority. This class performs deferred interrupt processing, i.e. is the top-half kernel processing of the interrupt requests. Time consuming parts of the interrupt service tasks should be implemented within the threads of the KERN class. The class that runs the ”normal” user applications, the *TIMESHARE* follows in priority. The kernel adapts dynamically the priority of the threads of this class in order to provide better response to the interactive tasks. The priorities of threads running in the timeshare class are adjusted by the kernel based on resource usage and recent CPU utilization. A thread has two scheduling priorities: one for scheduling user-mode execution and one for scheduling kernel mode execution. Finally, the *IDLE* class consumes the CPU time, when no useful task exists.

IV. Scheduling and Schedulers

Scheduling decisions are usually 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]. The overall scheme of Solaris prioritization clearly resembles the corresponding FreeBSD scheme. Both Windows, Solaris and Linux use a per-CPU *runqueue*.

All OSes favor interactive threads/processes. Interactive threads run at better priority than compute-bound threads, but tend to run for shorter time slices. Traditionally, time-slice

ISBN: 978-988-19253-0-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

based scheduling uses heuristic criteria to favor interactive tasks. This is suboptimal and problematic.

Linux introduces a new type of scheduling the *Completely Fair Scheduling* that avoids the problem of computing the proper time slices [6] The Completely Fair Scheduler (CFS) tries to approximate the concept of perfect multitasking, i.e. the fair sharing of processor time by the tasks. Suppose, for example, that we have N tasks of equal priority to schedule. The CFS should try to approximate the ideal situation of having them running as if each task was alone at the CPU, of course running at $\frac{1}{N}$ speed.

Implementing such an ideal protocol requires to have the processes run for infinitely small durations. However, in practice is not efficient to run the tasks for very small durations, due to context switching overheads and the effect of switching on caches. The larger are the time slices (i.e. time quantums) the smaller is the overall overhead, and the better the overall system throughput, at the expense of increasing task latencies. CFS intelligently settles this tradeoff by fixing the *targeted latency*, i.e. the maximum affordable latency for a set of tasks. Then CFS computes, based on the latency constraint, a schedule that shares the processor time fairly and optimizes switching overhead, i.e. it performs only the necessary task swaps to maintain the latency constraints.

The CFS also avoids starving low priority tasks. With CFS irrespective of how much heavily loaded is a system, even the less important tasks receive some time slice.

The switching rate for tasks with the timeslicing based Windows and Solaris scheduling is dependent on the load and cannot fullfill systematically the required latencies. In contrast, the Completely Fair Scheduler (CFS) of Linux, designs the scheduling starting with the constraint of maintaining the required latencies.

Time quantum based scheduling systems usually perform a priority boost after a release from a wait operation. Since usually the interactive I/O bound tasks are blocked, this strategy favors them. Also, they adaptively decrease the priority of compute bound tasks on every successive time quantum that they take. The Completely Fair Scheduler automatically favors I/O bound tasks, since their blocking time provides them an advantage to take their turn. In contrast, the traditional priority boost mechanisms of Windows, FreeBSD and Solaris are rather heuristic and their effective operation is based on an elaborate tuning of their parameters.

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 all operating systems have separate linked lists of runnable threads for different priorities. Both Solaris, FreeBSD and Linux use a *separate list* for each priority. The previous O(1) scheduler of Linux used 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.

All OSes schedule the one next thread to run, instead of attempting to derive a schedule for a whole group of n threads. Also, they 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. The other OSes also support similar processor task affinity schemes.

Both OSes support POSIX *SCHED_FIFO*, *SCHED_RR*, and *SCHED_OTHER* (or *SCHED_NORMAL*). *SCHED_FIFO* and *SCHED_RR* typically result in *realtime* threads. Solaris, FreeBSD 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.

One of the big differences between Solaris and Linux is the capability of Solaris to support multiple *scheduling classes* on the system at the same time. 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.

Similar to Linux, FreeBSD avoids the dynamic selection of scheduling policy for performance reasons. However, it allows the selection of the scheduler when the kernel is built [11].

V. Conclusions

The paper aimed to provide an insight to some key design approaches and components of Linux, Solaris, FreeBSD and Windows using a comparative view. All the four OSes are monolithic kernels (although Windows started as microkernel). Also, they allow the kernel to be extended flexibly with dynamically loadable modules. Windows imposes more layers between the hardware and the user space applications (e.g. the HAL, the interrupt objects). Microsoft virtualizes part of the hardware and builds other components upon that virtualization. Clearly, this is an advantage from the point of view of portability to different architectures but requires learning additional Microsoft APIs and being dependent on them. From the four OSes, Linux is generally the closest to the hardware, and combined with the clever and highly optimized implementation approaches, realize the fastest overall performance.

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.

The application level interface of Linux, FreeBSD and Solaris is very similar, typical of the modern UNIX system programming interface [7], [8]. Also, FreeBSD and Solaris share similar approaches at their internal design of low level process/thread structures and primitives, distinct from Linux that has its own design of multithreading. 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

ISBN: 978-988-19253-0-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

TABLE I
THE RESULTS OF BENCHMARKING I/O OPERATIONS WITH JAVA 7

I/O test	Linux	Windows
Plain Input Stream	4145	10328
Buffered Input Stream	197	219
Random Access File	5992	13517
Memory Mapped File	164	203

[9]. The presented performance results favor Unix-like OSes compared to Windows. Finally, from the three Unix based OSes, Linux performs better at the presented benchmarks, as can be expected with the less layering that has at its design.

However, the performance of Linux that we measured seems close to that of the recent FreeBSD kernel. Also, both kernels can be easily build from source, in our experience it is easier to build from source the FreeBSD kernel. An illustrative large benchmark that illustrates the similar performance of the two OSes, is the building from source of the LLVM-clang-cling sources (*https://github.com/llvm-mirror/clang*,*https://github.com/vgvassilev/cling*). The *gmake* based building process is a heavy one, the total source code is about 356.6MB. FreeBSD 11 completes that build in 4203.839 sec (user time), 177.652 sec (system time) with a ZFS filesystem, and in $4061.268u, 149.202s$ with a UFS filesystem. The sligthy slower performance of ZFS is perhaps due to the fact that the virtual memory system of FreeBSD currently does not integrate the caching of ZFS, and ZFS performs its own buffering. OpenSUSE 13.2 Linux with a custom build kernel, 4.1.0-2, and ext4 filesystem performs slightly better with respect to user time, i.e. $3937.621secs$ but a little slower with respect to system time i.e. $190.67secs$.

That review work tried to enlight some aspects of the internal operation of the compared OSes. Clearly, a lot of interest work remains, to better understand the advantages and disadvantages of the rather sophisticated cores of these modern OSes.

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* (3nd Edition) (Novell Press) by Robert Love, 2010
[7] Rich Teer, *Solaris System Programming*, Addison-Wesley, 2005
[8] 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
[9] *Advanced Programming in the UNIX Environment, Third Edition, W. Richard Stevens, Stephen A. Rago*, Addison-Wesley, 2013
[10] Mark E. Russinovich, David A. Solomon, Alex Ionescu, *Windows Internals*, 6th edition, 2012, Microsoft Press, Vol. I and II
[11] Marshall Kirk McKusick, George V. Neville-Neil, Robert N.M. Watson, *The design and implementation of the FreeBSD operating system*, Addison-Wesley, 2nd edition, 2014

TABLE II
THE RESULTS OF BENCHMARKING I/O OPERATIONS WITH JAVA 8

I/O test	Linux	Windows
Plain Input Stream	3945	11001
Buffered Input Stream	294	78
Random Access File	5893	14126
Memory Mapped File	55	62

TABLE III
THE RESULTS OF BENCHMARKS WITH THE PTHREAD LIBRARY (USER AND SYSTEM TIME IN SECS)

Benchmark	Linux	FreeBSD
Thread Creation	2.11, 9.5	3.00, 8.72
Thread Creation and mutex	2.98, 9.193	3.04, 9.2
Memory Map	2.5, 9.35	3.28, 8.21
Memory Allocation	2.44, 10.15	3.21, 8.18

Benchmark	Solaris	Windows 8
Thread Creation	7.0, 8.04	7.6, 32.07
Thread Creation and Mutex	7.119, 7.999	8.1, 33.1
Memory Map	7.6, 8.3	7.3, 38.53
Memory Allocation	7.5, 8.26	7.718, 36.68

Benchmark	Linux	FreeBSD	Solaris
forkWithNoWrite	0.08, 2.12	2.79, 11.23	4.71, 18.8
forkWithWrite	0.078, 2.376	2.9, 11.7	4.86, 19.4

ISBN: 978-988-19253-0-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)