Operating System Kernels

Patrick Bitterling
Computer Systems and Telematics
Institute of Computer Science
Freie Universität Berlin, Germany
bitterli@inf.fu-berlin.de

Abstract—**A Kernel is the backbone of most operating systems. This paper provides information about kernel architectures with their features and advantages. The three most actual desktop operating systems are distributions of Linux, (free)BSD and Windows Vista. The kernels (even when they have the same architecture) of these OS differs greatly in there subsystems. So this paper shows how the subsystems of these kernels work. Following subsystems are viewed: process management and scheduling, system calls, interrupts, kernel synchronization, time and memory management management and additionally a short look into the networking stack. But not all operating systems have a kernel. So there must be alternatives which are shown in the last part of section 3.**

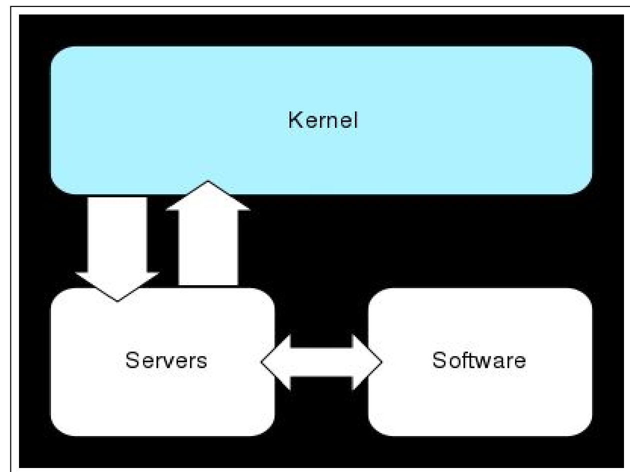
I. Introduction

In the past there are huge metal machines that are feed with magnetic tapes to perform some work. Today you can start your fair-size computer or even smaller laptop with pressing on a button. In a short time you have nice graphical user interface and a lot of applications, that can be used for the desired work. Between these two scenarios past more then fifty years has elapsed. In the beginning there are no operating systems for computers and the first operating systems did not include any kernel. The time elapsed and the computer had to support more hardware. But OS were developed further and it was time for the kernel-based operating systems. The first kernels were microkernels. The microkernel is a minimal computer operating system kernel. The first operating system kernels were small, because the computer memory was limited. Computers became more efficient and the operating system kernel had to control a larger number of devices. The address spaces increased from 16 to 32 bits, so the kernel design was not limited by hardware architecture and the size of the kernel began to grow. With the the grow of the kernel there was a new upcoming design the monolithic kernel. The monolithic kernel is used in operating systems with Linux-kernel and a lot of known BSD-derivatives as FreeBSD. It was also used by MS DOS and the MS Windows 9x series (till ME). Between this two designs exists the Hybrid kernel, which is found in the Windows NT series (includes XP and Vista) and in Mac's OS/2 and OS/X. In section 2 the different kernel architectures are explained. This include the benefits of these architectures. Section 3 gives a overview of the subsystems of three operating system kernels which is followed by a comparison of these subsystems.

II. The different kernel architectures:

These section discusses the differences between the microkernel, the monolithic kernel and the hybrid kernel.

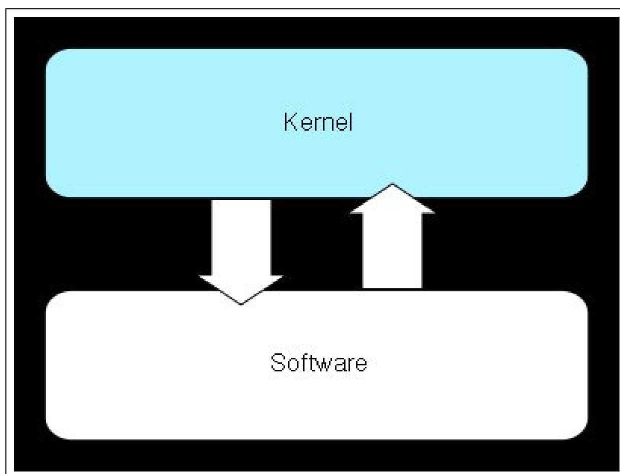
A. *The microkernel*



The design of a microkernel determines that it has only functions for a simple inter-process communication, memory management and scheduling. All these functions are operation in the kernel mode but the rest runs in the user mode. The microkernel is not implemented as one huge process. The functionality of the Microkernel is divided in several processes the so called Servers. In best case only these Servers get more privileges which need them to do their tasks. All servers are separated from the system and each process has his own address space. The result is that the microkernel cannot start functions directly. It has

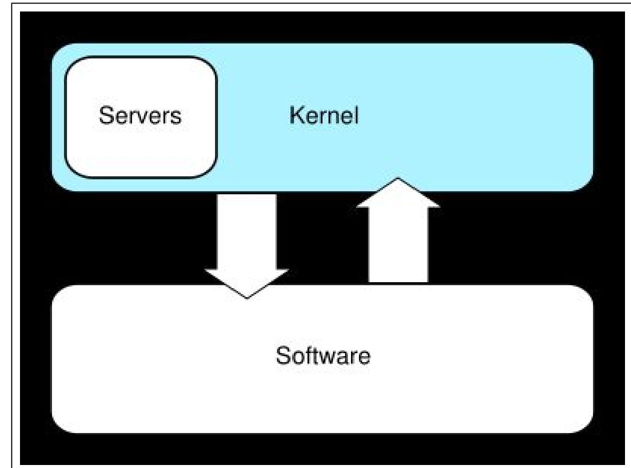
to communicate via "Message Passing" which is an inter-process communication mechanism which allows Servers to communicate to other Servers. Because of this implementation errors affect only the process in which it occurs. These modularization allows to exchange servers without jamming the hole system. The communication via inter-process communication produce more overhead then a function call and more context switches the a monolithic kernel. The result of the context switches is a major latency, which results in a negative performance.

B. The monolithic kernel



In contrast to the microkernel the monolithic kernel includes more functions, so there are more services that run in kernel-mode like all device drivers, dispatcher,scheduling, virtual memory, all inter-process communication (not only the simple IPC as in a microkernel), the (virtual) file system and the system calls, so only applications run in user-mode. The monolithic kernel is implemented as one process, which runs in one single address space. All kernel services are running in one kernel address space, so the communication between these services is more simple, because the kernel processes the ability to call all functions directly like a program in user-space. The ability to perform system calls results in better performance and simpler implementation of the kernel. The crash or bug in one module which is running in kernel-mode can crash the hole system.

C. The hybrid kernel



The hybrid kernel design is something between the microkernel and the monolithic kernel (that is the reason for the name). The hybrid kernel runs the same processes in kernel-mode microkernel. Additional the hybrid kernel runs the application IPC and the device drivers in kernel mode. In the user-mode is used for UNIX-Server, File-Server and applications. The goal of this architecture is to get the performance benefits of a monolithic kernel, with the stability of a microkernel The result is a microkernel-like structure in monolithic kernel and a controversy about the need of an extra category for this kernel or only have the two categories microkernel and monolithic kernel.

III. Kernel Subsystems

This section gives a overview of the Windows Vista, Linux 2.6.28 and FreeBSD 7.0 kernel subsystems. The overview begins with all Linux kernel subsystem, following by the FreeBSD and Vista kernel subsystem. At last there is a short look into the network protocol stack.

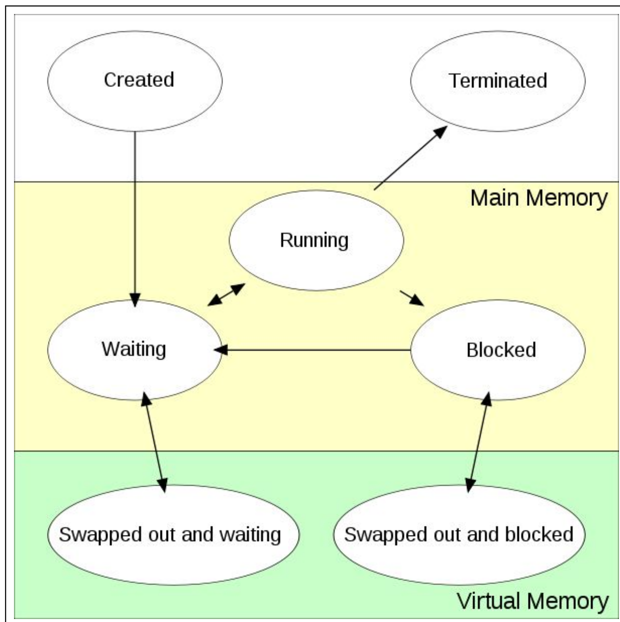
A. The Linux kernel architecture

The Linux kernel is a monolithic kernel, but it also a modular kernel. The Linux kernel possesses the ability to load and unload kernel-code. So the Linux kernel can load drivers for new devices on demand.

B. Linux process management

Linux kernel saves all pieces of information in a data structure.[1] These data structure contains information like open files, the address space of the process, status of the process. To identify a process each process contains a process identifier called PID. The PID is a number of the type (in most cases) int but uses only the length of short int (32768) to be compatible with

older UNIX or Linux versions. Linux, FreeBSD and Windows Vista using a five-state process management model.



Each process is in one of the following five states: RUNNING the task is running or in runqueue and awaits to run, INTERRUBTIBLE the process sleeps (blocked) and waits on a condition or a wake up signal to switch in the RUNNING state, UNINTERRUBTILE − like INTERRUBTIBLE but the process cannot wake up via signal to switch in the RUNNING state, ZOMBIE the task has finished but the parent has made the wait4() system call to free the process descriptor, STOPPED the execution of the process was stopped and the process has no possibility to switch back to RUNNING state. One important aspect of the process is the executed code which is read from a executable file and runs in within the address space. In general the execution of the program is placed in user-space till a system call or exception switch it to the kernel-space. The kernel is now in the process context which means that the kernel proceed a job in the name of this task. Then this process switched back to user-space if not the scheduler starts a process with a higher priority. All processes are organized in a hierarchy and each process is a descendant of the init-process, which has the PID of 1 and start as the last step of the booting system. Each process has no or several child-processes, so each process has exact on parent-process. To create a new process Linux uses the functions fork() and exec(). The function fork() creates a copy of the actual task that PID and PPID (parent-PID) differs from the original task. The copy of all resources would be to costly. So there are only the the resources duplicated which are used by both processes (called copy-on-write). The exec() function loads code in the address space and execute it. (Often a fork() is used before exec()). In Linux threads are only processes that are both share some resources and address space. For a computation in the background the kernel can spawn kernel-threads that are only working in the kernel-space and have no address space. To end a process the system call exit() is used or when a process must end because of an exception or a signal. When a process ends the parent gets a signal, free resources and calls schedule() to switch to another process. At last the process descriptor is freed. If parent process ends before the child process the child process must bound to another process, because of the process hierarchy.

C. *Linux process scheduling*

Since the 2.6.23 kernel Linux uses a Completely Fair Scheduler (CFS) [2], [3]. On real hardware only one single task can run at once. Most fair would be running each process with 1/n of the processors physical power. So while one task runs, the other tasks that are waiting for the CPU are at a disadvantage, because the current task gets an unfair amount of CPU time. In CFS this fairness imbalance is expressed and tracked via the per-task p->wait_runtime (nanosec-unit) value. ẅait_runtimeïs the amount of time the task should now run on the CPU for it to become completely fair and balanced. CFS's task picking logic is based on this p->wait_runtime value. It always tries to run the task with the largest p->wait_runtime value. In other words, CFS tries to run the task with most need for more CPU time. In practice the task runs only the amount of time until the p->wait_runtime value is low enough so that another task needs to run. To decide which task should run they are sorted in a time ordered red-black-tree. The CFS has additionally classes with variable priorities. When they are no more runnable task in one class the the CFS runs tasks from the next class.

D. *The Linux system calls*

System calls are function calls (most times) with the return value long to indicate of a success or failure.[1] If a failure accrues a error code number is written in errno which is a global variable. Each system call has his own unique number. When an application makes a system call it transmitted a number to the kernel, which has a table with all system calls (dependent on the architecture). To perform a system call Linux uses a software-interrupt. The exception arrange that the system switches in the

kernel mode. As a new feature the sysenter for x86-processors allows a faster system call. When the system call finishes the process switched back in user-space and continues the process.

E. Linux interrupt handling

All device that can make an interrupts have an interrupt-handler, which is part of the driver and are C-functions.[1] The interrupt-handler is also called interrupt-service-routine (ISR). First the action of ISR is that it affirm the interrupt so the device can continue to work. Interrupts are complex an so divided in the top-halves and the bottom-halves. The top-halves are for time critical work and are started when an interrupt accrues. All rest work which is not time critical is done in the bottom-halves. The Linux provides the possibility the activate or deactivate interrupts (all or individual). For the bottom-halves work Linux provides actually three different mechanisms Softirqs, Tasklets and Work-Queues.

F. Linux Kernel synchronization

Shared-Memory-Applications are applications using the same resources.[1] These resources must be protected against race conditions. Race conditions occur when the output and/or result of the process is unexpectedly and critically dependent on the sequence or timing of other events (two signals racing each other to influence the output first). Race conditions lead to errors which are difficult to localize. One protection mechanism is Locking. Each resource is locked by one process , so other processes cannot use them as long as it is locked by this process. Locking resources can lead to deadlocks. Deadlocks occur when a process 1 sets a lock for a resource A and process 2 on the resource B. Process 1 wants to lock resource B but cannot and waits. Process 2 wants to lock resource A but cannot and waits too. Each process cannot continue because the resource it needs is never be freed. Linux possesses mechanisms against race conditions and dead locking. Atomic operations are a collection of instructions which cannot be interrupted. Spinlocks are locks and processes which want the lock are in a busy-loop to get that lock. A modification is the Reader-Writer-Spinlock that allows multiple processes to read but not to write on the same resource. Another mechanism is the semaphore it is also called sleeping lock. The first process sets a lock on resource. If a second process needs that resource this process is sent to sleep an awakes when the resource is freed. There are also Read-Write-Semaphore as is the case with spinlocks. One similar method is the completion-variable. The completion-variable is used after a a task has finished to wake other tasks. New in the 2.6 kernel is the seq-lock, which is a lock and a sequence counter. It favors write-processes and solves the problem of writer starvation.

G. Linux Timer and Time-Management

For events like refreshing the system-uptime, the time of the day, rescheduling the time and timer-interrupt are of great importance.[1] Since the 2.5 kernel the timer-interrupt uses a frequency of 1000Hz. This time period is also called "tick". A architecture independent mechanism to trigger and program timer-interrupts is the system-timer. To execute Kernel-Code to a specific time the kernel can set, the so called Dynamic Timers or Kernel-Timer. To delay the execution of code uses the Busy-Loop, which could have a accuracy of micro seconds. For longer delays the task should be send to sleep an wake up in a defined time of seconds.

H. Linux Memory management

To organize memory, it is divided into blocks, called pages.[1] Pages are depending on the architecture and can support different page sizes. (usually 4 kByte on 32Bit-Architecture and 8 kByte on 64Bit-Architecture). Hardware limitation does not allow to treat all pages equal due to the physical address. Linux uses three different zones. The ZONE_DMA for DMA-enabled Pages, the ZONE_NORMAL for regular Pages and ZONE_HIGHMEM for pages that are not permanently mapped in the address space of the kernel. Allocating and freeing data structures is one of the most common operations inside a kernel. Instead of allocating memory for every single data structure it exists a Free-List (Slab-layer). Whenever space for a data structure is needed it pick instead the allocated space of one objects of the Free-List. The user-space uses instead stacks which have a large and dynamic space. The page-cache is a cache consisting of pages. To analyze if a page is already cashed Linux uses a Radix-Tree. The Radix-Tree is a binary tree which allows a fast search for the desired page. The buffer-cache is not separated from the page-cache. These two caches are not separated because the synchronization of these is complicate. The memory of the cash is limited and pages must be write back. Pages are written back when pages are to old or the memory is nearly fueled.

I. The BSD kernel architecture

The freeBSD kernel is a monolithic kernel, which can load and unload kernel modules as does a Linux kernel.

J. The BSD process management

Each thread of execution is a process.[4] For each process there is a struct called proc, which has all relevant pieces of information like the PID PPID, the priority of the process and the state of the process. There are five different stats: SIDL,SRUN,SSLEEP,SSTOP and SZOMB. The function fork() creates a copy of the actual task that PID and PPID (parent-PID) differs from the original task. The copy of all resources would be to costly so there are only the the resources duplicated which are used by both processes (called copy-on-write). The parent-child relationship induces a hierarchical structure on the set of processes in the system. A process can overlay itself with the memory image of another program, passing to the newly created image a set of parameters, using the system call execve . Usually a process terminated via the exit system call, sending 8 bits of exit status to its parent. When the parent process terminated before the child process, the child process must bind to another process because of the hierarchical process structure.

K. The BSD process scheduling

FreeBSD uses a time-share scheduler, the process priority is periodically recalculated based on various parameters, like the amount of CPU time it has used, the amount of memory resources it holds or requires for execution.[5] The scheduler a multilevel feedback queue which prefers short jobs and I/O-bound processes. At the base level queue the processes circulate in round robin fashion until they complete and leave the system. Each process starts with a value, the nice value. It is a number between -20 and 20. The second value is the realtime-priority which is a number between 0 and 99. The timeslice a value that decided how long a process runs before it gets preempted. A process must not spend all it's timeslice in one turn it can reschedule. A process without timeslice cannot run and gets a new timeslice when all other process spend their timeslice. The result is that most I/O-bound process can preempt a processor-bound process, but processor-bound processes have a greater timeslice. Some tasks require more precise control over process execution called real-time scheduling, which must ensure that processes finish computing their results by a specified deadline or in a particular order. The FreeBSD kernel implements real-time scheduling with a separate queue from the queue used for regular time-shared processes. A process with a real-time priority is not subject to priority degradation and will only be preempted by another process of equal or higher real-time priority. The FreeBSD kernel also implements a queue of processes running at idle priority. A process with an idle priority will run only when no other process in either the real-time or time-share-scheduled queues is runnable and then only if its idle priority is equal or greater than all other runnable idle priority processes.

L. BSD system calls

System calls are function calls with the return value long to indicate of a success or failure.[6] If a failure accrues a error code number is written in errno which is a global variable. Each system call has his own unique number. When an application makes a system call it transmitted a number to the kernel, which hast table with all system call (dependent on the architecture). The exception arrange that the system switches in the kernel mode. When the system call finishes the process switched back in user-space and continues the process. BSD can also emulate Linux system calls.

M. BSD interrupt handling

In freeBSD interrupt handlers are high-priority threads which run with interrupts enabled and may block on mutex. These scheme is called interrupt threads. Interrupt threads simplifies the locking in the kernel. The freeBSD kernel is preemptive and allows to preempt a interrupt thread by a higher-priority interrupt thread. Interrupt threads run real-time kernel priority.

N. BSD kernel synchronization

The OS must assure that not two (or more) processes use the same resources at the same time, FreeBSD uses look mechanisms and atomic operations.[7], [8] To short a resource for a short amount of time, freeBSD uses mutexes, which are spinlocks. A mutex is a lock and is owned by the process, who first claims the resource. No other process can get the mutex until it is released by the other process. These locks can acquired recursively, but should not send to sleep. Shared/Exclusive locks are reader/writer locks that allow multiple processes to read but only one process to write. These locks can also send to sleep. Some variables are protected via atomic operations. Because these variables can only access by these operations they do not need to be locked.

O. BSD Timer and Time-Management

Clock-Interrupts are interrupt occuring in fixed intervals.[9] The amount of time between two clock-interrupts is a tick. The tick most BSD system is 10ms (100Hz). FreeBSD provides two levels that coherent

with all time-related work. The hardclock level does jobs like the incrementation of time of day .It runs the current process's virtual and profile time (decrease the corresponding timers) and Schedule softclock interrupts if any callouts should be triggered. The other level is the clock software interrupt level which is the real-time timer in processes. It liable for retransmitting dropped network packets. It is responsible for the watchdog timers on peripherals that require monitoring and used when the process-rescheduling event occurs.

P. BSD Memory management

To organize memory FreeBSD uses pages, too.[10], [5] As in Linux the address space is divided in different zones. The using of free-list for data structures (slab-layer) have is origins in BSD (Linux adopted it). In user-space freeBSD also provides stacks as a large and dynamic memory. freeBSD uses a LRU replacement system for the decide which pages are swapped. If more memory is needed freeBSD can swap a hole process. freeBSD uses a page hash table instead of a radix-tree. So freeBSD has to look in a double linked list to decide if a page is already in the cache.

Q. The Windows Vista kernel architecture

Windows Vista differs in the kernel architecture from Linux and FreeBSD. Instead of a monolithic kernel, Windows Vista uses a so called hybrid kernel.

R. Windows Vista process management

The unit for a process in Vista is a thread.[11] Vista has a container which contains at least one thread. Each process has a virtual address space executable code, open handles to system objects, a security context, a unique process identifier, environment variables, a priority class, minimum and maximum working set sizes, and at least one thread of execution. The process started with a single thread (primary thread) but can create additional threads from any of its threads. All threads of a process share its virtual address space and system resources. In addition, each thread maintains exception handlers, a scheduling priority, thread local storage, a unique thread identifier, and a set of structures the system will use to save the thread context until it is scheduled. The thread context includes the thread's set of machine registers, the kernel stack, a thread environment block, and a user stack in the address space of the thread's process. Threads can also have their own security context, which can be used for impersonating clients.

S. Windows Vista process scheduling

Because a process contains one or more threads, Windows Vista has to schedule these threads. Microsoft Windows supports preemptive multitasking.[11] On a multiprocessor computer, the system can simultaneously execute as many threads as there are processors on the computer. A job object allows groups of processes to be managed as a unit. Job objects are namable, securable, sharable objects that control attributes of the processes associated with them. Operations performed on the job object affect all processes associated with the job object. Because Windows Vista is multitasking system each process runs for short time and not until it is finished. Each process gets a time slice. A time slice is the amount of time in which a process can run. To decide which thread should run first each thread has a priority level. The priority levels range from zero (lowest priority) to 31 (highest priority). Only the zero-page thread can have a priority of zero. (The zero-page thread is a system thread responsible for zeroing any free pages when there are no other threads that need to run.) The system treats all threads with the same priority as equal. The system assigns time slices in a round-robin fashion to all threads with the highest priority. If none of these threads are ready to run, the system assigns time slices in a round-robin fashion to all threads with the next highest priority. If a higher-priority thread becomes available to run, the system ceases to execute the lower-priority thread (without allowing it to finish using its time slice), and assigns a full time slice to the higher-priority thread. The priority of each thread is determined by the priority class of its process and the priority level of the thread within the priority class of its process. Windows Vista has a function which can temporarily increase the priority level of a thread, which hast a priority between 0 and 15. This is called Priority Boost and ensures that threads do not starve (wait forever to get processor time). After raising a thread's dynamic priority, the scheduler reduces that priority by one level each time the thread completes a time slice, until the thread drops back to its base priority.

T. Windows Vista system calls

A system call in Windows Vista is function in the Windows API.[12] The system locks the EAX-register, places a number into the register. Then the sysenter instructions is called to switch to the kernel mode. Because of the number in the EAX-register the kernel can perform the needed task and switch back the user mode.

U. Windows Vista interrupts

Hardware and software interrupts differ on each computer.[13] The kernel treats interrupts on all machines in a similar way by virtualizing the interrupt processing mechanism. The hardware abstraction layer (HAL) is responsible for providing a virtual interrupt mechanism to the kernel. Interrupts have a priority the interrupt request level (IRQL). Windows provides interrupt masking. Interrupt masking is a mechanism that masks all interrupts with the same or lower IRQL that the actually interrupt has. If an interrupt occurs at a higher IRQL, then the higher priority interrupt is serviced immediately. When an interrupt occurs, it is handled by a function called an interrupt service routine (ISR) . Data structures called an interrupt dispatch tables (IDT) track which interrupt service routine(s) will handle the interrupts occurring at each IRQL. A separate interrupt dispatch table is associated with each processor. So each processor can potentially associate different interrupt service routines with the same IRQL, or one processor can to handle all interrupts. The kernel supplies the interrupt service routines for many system interrupts such clock ticks, power failure, and thread dispatching. Other interrupt service routines are provided by the device drivers that manage peripheral hardware devices such as network adapters, keyboards, pens, and disk drives. A device driver associates its interrupt service routine with an IRQL by constructing an interrupt object and passing it to the kernel. The kernel then connects the interrupt object to the appropriate interrupt dispatch table entry. When an interrupt occurs at the device's IRQL, the kernel locates the device driver's interrupt service routine using the interrupt object. More than one interrupt object can be associated with each IRQL, so multiple devices could potentially share the same IRQL. When a device driver is unloaded, it simply asks the kernel to disconnect its interrupt objects from the interrupt dispatch table. Interrupt objects increase device driver portability by providing a way to connect and disconnect interrupt.

V. Windows Vista kernel synchronization

Because Windows Vista is multitasking OS, it needs synchronization objects in order to access shared data.[14] The kernel is responsible for creating, maintaining, and signaling synchronization objects. Although the Executive creates a number of complex synchronization objects, each of these contains at least one kernel synchronization object. When a process waits on a synchronization object, the kernel changes its dispatcher state from running to waiting. When the object finally becomes unlocked, the kernel changes the dispatcher state of the process that was waiting from waiting to ready. Usually processor spent no time in polling the object in a loop. The process that is waiting does not execute at all until the object is unlocked. The kernel also has a type of lock called a spin lock which does loop until it becomes unlocked. Spin locks are only used in very special cases in the kernel and in device drivers. Similar to the Linux atomic operations, Windows Vista has the kernel transaction manger (KTM)[15]. The KTM enables applications to use atomic transactions on resources by making them available as kernel objects.

W. Windows Vista Timer and Time-Management

Windows Vista uses a timer interrupt frequency of 100 Hz. (66.6 on multiprocessors).[16] For a better timer interrupt resolution Windows uses a high precision event timer (HPET). The HPET is hardware timer and has a higher precision then the real time clock.

X. Windows Vista Memory Management

The virtual address space of each process can be smaller or larger than the total physical memory available on the computer.[17] The subset of the virtual address space of a process that resides in physical memory is known as the working set. If the threads of a process attempt to use more physical memory than is currently available, the system pages some the memory contents to disk. The total amount of virtual address space available to a process is limited by physical memory and the free space on disk available for the paging file. To maximize its flexibility in managing memory, the system can move pages of physical memory to and from a paging file on disk. When a page is moved in physical memory, the system updates the page maps of the affected processes. When the system needs space in physical memory, it moves the least recently used pages of physical memory to the paging file. Manipulation of physical memory by the system is completely transparent to applications, which operate only in their virtual address spaces. Because the RAM is limited Windows Vista more processes can share the same page until one process wants to write to this shared page. (This is called Copy-on-Write Protection)

Y. The network protocol stack

A protocol stack is a software implementation of a computer networking protocol suite.[18] The suite is the definition of the protocols, and the stack is the software implementation of them. Most protocols are used for one purpose. The modularization makes design

and evaluation easier. Because each protocol module usually communicates with two others, they are commonly imagined as layers in a stack of protocols. The lowest protocol is designed to realize physical interaction of the hardware. Higher layer adds more features. User applications are the topmost layers. The OSI (open system interconnection) model is the best known. In practical implementation, protocol stacks are often divided into three major sections: media, transport, and applications. A particular operating system or platform will often have two well-defined software interfaces: one between the media and transport layers, and one between the transport layers and applications. The media-to-transport interface defines how transport protocol software makes use of particular media and hardware types ("card drivers"). For example, this interface level would define how TCP/IP transport software would talk to Ethernet hardware. Examples of these interfaces include ODI and NDIS in the Microsoft Windows and DOS environment (NdisWrapper in Linux and Project Evil for FreeBSD). The application-to-transport interface defines how application programs make use of the transport layers. For example, this interface level would define how a web browser program would talk to TCP/IP transport software. Examples of these interfaces include Berkeley sockets and System V streams in the Unix(-like) OS, and Winsock in Microsoft Windows.

Z. *Alternatives for a kernel-based OS*

A kernel provides many practical features, but is not required to run a computer. Historical the first operating systems do not contain a kernel. The OS was implemented for only one type of machine (all with the same hardware specification). The OS do not support any hardware abstraction. Today these sort of implementation is used in embedded systems and some game consoles. There is another reason for using a kernel-less architecture. KLOS (kernel-less operating system) uses another architecture because each service call results in a larger overhead because of the context switch.[19] The heart of KLOS is the Event Core. It contains the only part which runs in the privileged mode (Ring 0) the rest of the Event Runs in the unprivileged mode (Ring 3). The Ring-0 Event Stubs running in privileged mode. There is one event stub for every hardware interrupt and exception that the underlying processor supports. KLOS implemented new protections to secure the stability of the OS. The service calls do not need to access the Ring 0 and produce so less overhead. Test results (see paper of KLOS) confirm the better performance of the service call mechanism of KLOS with a amount of protection that is nearly as good as other current operating systems.

IV. The kernel subsystem comparison

At first it is useful to compare the Linux kernel and the FreeBSD kernel. Because both are UNIX-like OS, they share some similarities. The structure of a process and his creation are nearly identical. The differences in the scheduling are significant. The new CFS of Linux is not automatically better as the scheduler in FreeBSD. The FreeBSD 7.0 was about 15% better in MySQL the Linux 2.6.23 kernel.[20] The interrupts in FreeBSD 7.0 and Linux differs. Interrupt threads greatly simplifies locking in the kernel and can easily preemted by higher priority threads. FreeBSD and Linux use similar techniques for kernel synchronization and memory management. Windows Vista has more differences. It begins with a other understanding of processes which results in other scheduling. Windows Vista uses more objects to handle things like the interrupt object or the kernel object by atomic transmissions.

V. Conclusion

The section 2 reveals the reasons why current operating systems using monolithic or hybrid kernel. Both architectures promise more speed then microkernel. Windows Vista uses the hybrid kernel architecture to gain also some of the security of the microkernel. The kernel comparison shows that each system has developed unique subsystems. The greater problem is to determine which kernel has the best subsystems. Benchmarks using one application are pointless because similar applications can perform better work on another operating systems. The second problem is that especially Linux is in a fast development. All patch versions of the Linux kernel a released within 22 till 107 day, so most benchmarks become fast out of date. The benchmark above profs that new developed CFS is not as good as the FreeBSD scheduler. But in the meantime exists a lot more kernel patches that reduces the overhead of the CFS which may beat now the FreeBSD scheduler in MySQL. But the history shows that operating systems benefit from other operating systems. For example the slab layer that Linux adopted from BSD. Linux is used a sandbox for some new concepts that may find a place in a BSD-derivate kernel.

References

[1] R. Love, *Linux-Kernel-Handbuch*. ADDISON-WESLEY, 2005.

[2] T. Gleixner, “The completly fair scheduler,” http://www.linux-foundation.jp/uploads/seminar20080709/lfjp2008.pdf, July 2008.

[3] http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt.

[4] M. W. Lucas, *Absolute FreeBSD*. No Starch Press, October.
[5] M. K. McKusick, *Design and Implementation of the FreeBSD Operating System*. Addison-Wesley Professional, August.
[6] http://www.freebsd.org/doc/en/books/developers-handbook/x86-system-calls.html.
[7] http://www.usenix.org/events/bsdcon02/full_papers/baldwin/baldwin_html/node5.html#SECTION00053000000000000000.
[8] http://www.freebsd.org/doc/en/books/arch-handbook/locking.html.
[9] www.kernelchina.org/bsdkernel/freebsd.ppt.
[10] http://www.freebsd.org/doc/en/books/arch-handbook/vm.html.
[11] http://msdn.microsoft.com/en-us/library/ms684841(VS.85).aspx.
[12] http://www.codeguru.com/cpp/w-p/system/devicedriverdevelopment/article.php/c8035.
[13] http://technet.microsoft.com/de-de/library/cc767887(en-us).aspx.
[14] http://msdn.microsoft.com/en-us/library/ms686353(VS.85).aspx.
[15] http://msdn.microsoft.com/en-us/library/aa366295.aspx.
[16] http://widefox.pbwiki.com/Clock.
[17] http://msdn.microsoft.com/en-us/library/aa366779(VS.85).aspx.
[18] http://en.wikipedia.org/wiki/Protocol_stack.
[19] A. Vasudevan, *KLOS: A High Performance Kernel-Less Operating System*.
[20] http://www.scribd.com/doc/551889/Introducing-Freebsd-70.