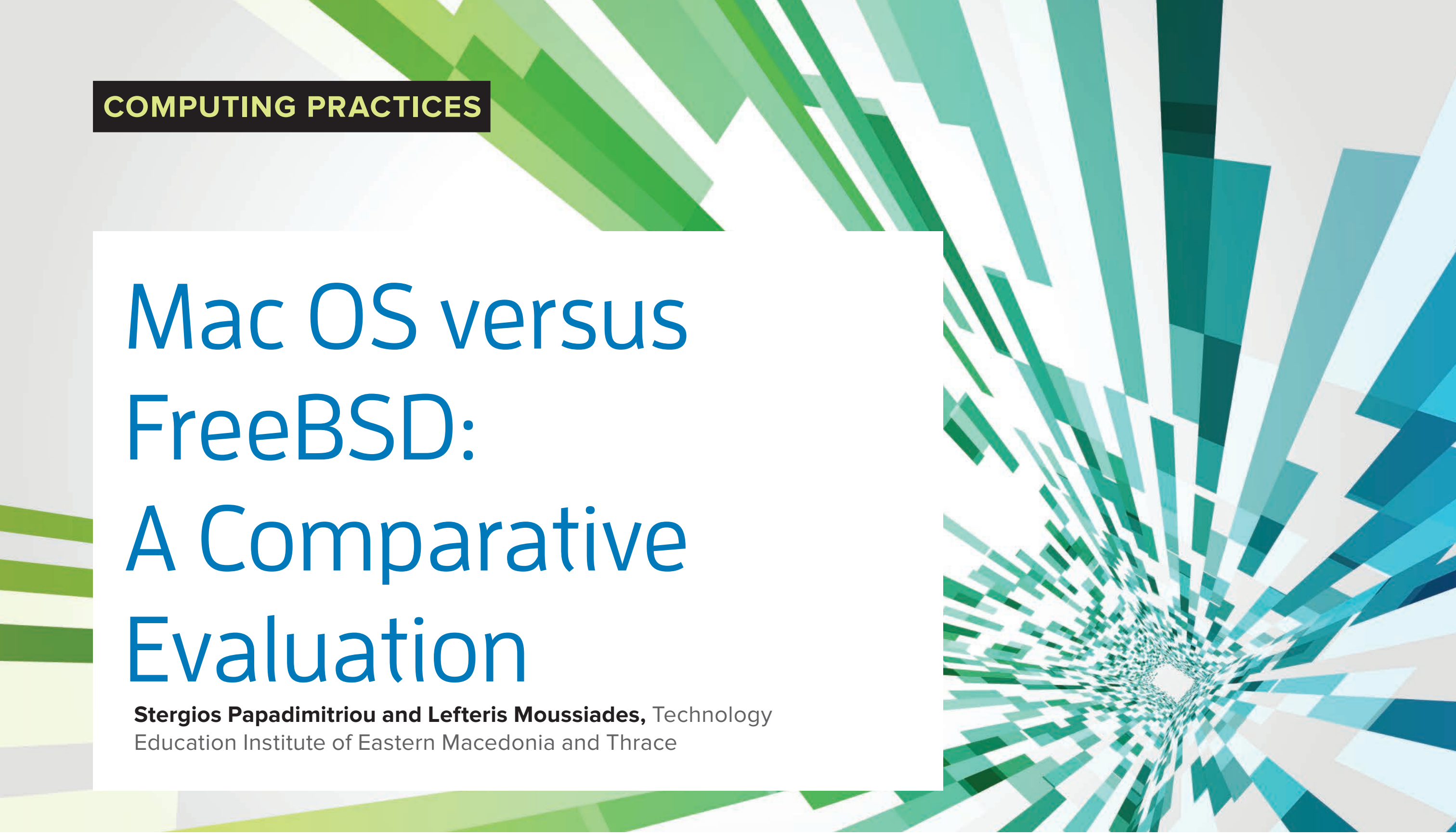

Mac OS versus FreeBSD: A Comparative Evaluation

Stergios Papadimitriou and Lefteris Moussiades, Technology Education Institute of Eastern Macedonia and Thrace

FreeBSD (an open source Unix-like OS) and Apple's Mac OS use similar BSD functionality but take different approaches. FreeBSD implements a traditional compact monolithic Unix kernel, whereas Mac OS builds the BSD Unix functionality on top of the Mach microkernel. The authors provide an in-depth technical investigation of both approaches.

FreeBSD—an open source Unix-like OS—and Apple's Mac OS both implement similar BSD Unix functionality but use radically different approaches. BSD stands for Berkeley Software Distribution and refers to the version of Unix developed at the University of California, Berkeley.

Mac OS uses the xnu kernel—xnu stands for "xnu is not Unix."[1–3] It consists of the Mach core, the BSD layer, and the I/O Kit (an object-oriented device driver development framework).[3] Specifically, xnu uses a modified Mach microkernel[2,3] for the lower-level functionality. On top of the Mach primitives, it implements BSD functionality. In essence, it layers BSD on top of the Mach infrastructure.

FreeBSD uses a custom modular kernel,[4] which is organized as a traditional Unix monolithic kernel. The FreeBSD kernel creates and manages processes, provides functions to access the filesystem, and supplies communication facilities.

In this article, we compare the core kernel architecture and functionality of the two OSes, providing some theoretical insights along with supportive performance benchmarks that highlight the relative advantages and disadvantages of each approach.

BASIC ARCHITECTURE

Both xnu and FreeBSD are monolithic kernels. This organization allows for efficient execution because kernel components interact with direct procedure calls without involving message passing and the associated switches between user and kernel mode code. The Mach kernel is actually a microkernel design, but performance demands forced Apple to implement the BSD component within the same kernel address space and with direct function calls that utilize Mach's functionality. Although it is well optimized, the elegant infrastructure of ports and message passing that Mach implements cannot compete with the efficiency of pure function calls.

Most OSes present a complete model on top of which user mode processes can be implemented. Instead, Mach provides a bare-bones model on top of which a full OS can be structured. Mac OS's xnu is one specific implementation of BSD on top of Mach.

The xnu kernel has a modular design with distinct kernel components in different modules. However, because all the modules operate in the same kernel space, routines of one module can directly call routines from other modules. Actually, many modifications of the original Mach code made by Apple replace Mach's message passing with direct function calls. This way, the xnu kernel avoids a great deal of overhead (as do the pure monolithic kernels of Linux, Windows, FreeBSD, and Solaris). The modular yet monolithic design of the xnu kernel is sometimes referred to as a hybrid design.

The xnu kernel includes the Mach microkernel, which provides a powerful port-based interprocess communication (IPC). It also implements core primitives as threads/tasks and schedules threads. Virtual memory (VM) is another core functionality implemented with Mach. On top of Mach and at the same kernel address space (for efficiency), the BSD layer implements the virtual file system (VFS), specific file systems, and networking, and provides a BSD-style Portable Operating System Interface (POSIX) functionality. Finally, the I/O Kit provides an object-oriented framework for simplified driver development. It consists of header files and libraries that provide the services required for kernel driver development, as well as header files and libraries that are used by user space code to locate a kernel driver and interact with it.[3]

The usual practice of monolithic OSes is to provide efficient access to objects through the method calls of a well-defined programming interface. This practice is not followed by xnu's Mach core. Instead, Mach organizes communication between objects with message passing. Messages are passed between end points, or ports. These are 32-bit integer identifiers, although they are used as opaque objects. Messages are sent from one port to another port. Multiple senders can send messages to the same port, and these messages are enqueued until they are received by the designated receiver for that port.

OSes usually use some kind of descriptor or handle to access objects.[5] Instead, Mach accesses primitive objects through corresponding ports. This kind of structuring is very modular and flexible, allowing ports and rights to be passed from one entity to another. For example, complex Mach messages can contain ports delivered from one task to another, a mechanism akin to mainstream Unix's domain sockets, which allow passing file descriptors between processes.

Unlike xnu, the FreeBSD kernel is organized as a traditional Unix monolithic kernel.[4] Logically, the kernel can be divided into a top half and a bottom half. The top half of the kernel provides services in response to system calls or traps. In essence, it is a library of routines shared by all processes. The bottom half of the kernel consists of routines that handle hardware interrupts.

The FreeBSD kernel consists of processes that execute in kernel mode and routines that execute periodically within the kernel. Mach's kernel is organized as a kernel task with multiple kernel threads.

[THE FREEBSD KERNEL CONSISTS OF PROCESSES THAT EXECUTE IN KERNEL MODE AND ROUTINES THAT EXECUTE PERIODICALLY WITHIN THE KERNEL.]

PROCESSES AND THREADS

FreeBSD implements a multithreaded process design.[4] Each FreeBSD process keeps a linked list of its threads. These threads are scheduled by the kernel, and they own their 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; in other words, the concept of variable-weight processes is implemented.[4] FreeBSD has the *rfork()* system call that behaves like Linux's *clone()* system call. Xnu also implements variable-weight processes within its BSD layer, but with a different mechanism—by implementing multi-branch functionality at the *fork()* system call.

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

FreeBSD implements an elaborate priority-inversion mechanism using turnstile queues.[4] It organizes its sleep and turnstile queues effectively by keeping them in a data structure hashed by an event identifier. The hashing provides efficient search for the threads that need to be awakened for an event. When a high-priority thread blocks a contested resource, its turnstile queue is used to detect the lock holder thread and to raise the priority of that thread to release the lock. This priority-inversion mechanism significantly facilitates the implementation of applications with real-time constraints.

FreeBSD frequently explores adaptive spinning,[6] where the thread spins if the lock holder is currently executed on some CPU (with the hope that it will release the lock soon). When the lock holder is not in execution, however, it is wasteful to spin, so the thread is sleeping—waiting for the lock to be released. Mutexes of FreeBSD can be acquired recursively, and therefore a second acquisition of the same lock does not deadlock the thread.

FreeBSD's timeshare scheduling avoids preempting threads running in kernel mode, thus the worst-case real-time response to events is defined by the longest path through the top half of the kernel. However, FreeBSD provides real-time threads and interrupt threads that can preempt the timeshare threads, even in kernel mode. Therefore, no upper bounds on the duration of a system call, when running with just the timeshare scheduler, are guaranteed. Thus, the timeshare-scheduling algorithm is definitely not a hard real-time system.

However, FreeBSD can service real-time workloads. In other words, it can implement hard real-time tasks by using real-time and interrupt threads. These threads preempt lower-priority threads. The longest path that preemption is disabled for real-time and interrupt threads is defined by the longest time a spinlock is held and the longest duration of code within a critical section. Real-time applications should (and usually can) impose worst-case bounds on these times.

FREEBSD ELEGANTLY DIVIDES THE KERNEL STATE OF A PROCESS IN TWO PRIMARY STRUCTURES: THE PROCESS STRUCTURE AND THE THREAD STRUCTURE.

Concerning the synchronization primitives, both Mach and FreeBSD kernels offer efficient implementations of mutexes, read-write lock objects, spinlocks, and semaphores. Mach also implements lock-set objects—arrays of locks that can be acquired by a given lock ID. An interesting aspect of lock-sets is that they allow the handoff of locks—the passing of a lock from one task to another. Mach implements the flexibility of the handoff in the context of scheduling decisions. Specifically, a thread can yield the processor voluntarily, but can also specify which thread to run in its stead.

Another advanced feature implemented by the Mach kernel is that of continuation. A continuation is an optional resumption function along with a parameter to it, which a thread might specify if it voluntarily requests a context switch. Continuations improve significantly on the context switch time, because the thread is reloaded from the point of continuation with a new stack and no previous state was saved.

Mach also provides an abstraction of the machine with a host object and a useful API that provides information about kernel modules, memory tables, and other aspects that xnu's POSIX-based BSD layer does not offer. Mach selectively implements scheduler algorithm indirection. If only one algorithm is enabled at compile time, a direct function call is used; otherwise, calls are dispatched to the selected scheduling algorithms through a function pointer table.

Mach avoids spurious timer interrupt processing by using a tickless kernel. Specifically, at every timer interrupt, the timer is reset to schedule the next interrupt only when the scheduler decides that it is necessary. The xnu interrupt handler performs a fast pass over the list of pending deadlines (which usually are sleep timeouts set by threads), acts on them if necessary, and schedules the next timer interrupt according to these pending deadlines.

A BSD-style thread in xnu is not implemented independently but instead is mapped to a Mach thread. Like FreeBSD threads, Mach threads also represent the atomic unit of execution—both xnu and FreeBSD kernels schedule threads and not processes. As with threads, xnu builds BSD-style processes on top of Mach. Specifically, Mach tasks provide the machinery for implementing processes. Mach represents the kernel itself as a task. In essence, Mach provides primitives such as threads, tasks, scheduling, and VM as low-level abstractions with a deliberately basic and incomplete API, on top of which xnu builds rather efficiently.

The Mach kernel is organized as a set of kernel threads that run within a single task—the kernel task. The kernel threads perform all the kernel operations, such as scheduling, memory allocation, paging, and exception handling. Although the FreeBSD kernel also performs a lot of operations with kernel threads, it is not structured by itself as a process.

TRAP AND INTERRUPT STRUCTURE

Modern interrupt controllers perform a kind of prioritization on the interrupt request lines (IRQ). Generally, interrupts on the lower-priority lines are disabled whenever the processing of an interrupt request on a higher-priority line is pending.

Both FreeBSD and xnu use a system table called the 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.

Xnu provides three ways to perform kernel requests: BSD traps, Mach traps, and Mach remote procedure calls (RPCs). The Mach kernel implements very few system call entry points. Most Mach operations are implemented as Mach RPCs, including operations for VM and for thread and task control. The IPC-based implementation of traps, although elegant, imposes performance overhead.

Similarly to most other OSes, a system call number argument that is checked at kernel entry identifies the system call. However, xnu differs—a positive argument identifies a BSD system call, whereas a negative one indexes the Mach system call table.

A common practice for processing an interrupt 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 of the interrupted thread. If we reschedule, the stack with the saved interrupt frames is lost, and it is problematic to restore those contexts after the interrupt handler finishes.

Both systems on return from trap-interrupt processing check for the existence of ASTs (Asynchronous System Traps). Mach represents a particular AST with a reason bit. These reason bits are set by the software to trigger the corresponding trap. When a processor is about to return from an interrupt context, including returns from system calls, it checks for these bits and takes a trap if it finds one. The pending of such traps is checked in many cases when a thread is to change its execution state; for example, being suspended from running. Also, the kernel's clock-interrupt handler periodically checks for ASTs.

Mach implements a unique exception-processing approach, layering it on top of its message-passing architecture. Actually, Mach does not handle the exception but leaves exception handling to the upper software layers (for example, the BSD layer for xnu).

Mach exceptions are handled via the primary facility of the kernel—its message passing. An exception is wrapped to a message, which is raised via the *msgsend()* and caught by a handler through *msg-recv()*. Unlike the traditional models, which run the exception handler in the context of the faulting thread, Mach runs the exception handler in a separate context by making the faulting thread send a message to a predesignated exception port and wait for a reply.

Mach exception handling using thread and task exception ports for every type of exception is versatile and elegant. The thread that causes an exception is called the *victim thread*, whereas the thread that runs the exception handler is called the *handler thread*. When a victim raises an exception, the kernel suspends it and sends a message to the appropriate exception port, which can be either a thread exception or a task exception port. Subsequently, the handler thread processes the exception.

FreeBSD creates a thread context for each interrupt handler, making it impossible to access the context of another interrupt handler. Also, each interrupt handler has its own stack on which it runs.

In FreeBSD, interrupt handlers are composed of a filter routine, an ithread routine, or both. The filter routine executes in primary interrupt context—it does not have its own context. Therefore, it cannot block or switch the context, and spin mutexes should be used instead of sleep mutexes. These constraints impose the utilization of filter routines only with devices that require a nonpreemptive interrupt handler.

A filter routine can either completely handle an interrupt or relocate the computationally expensive work to its associated ithread routine, if it has one. An ithread routine, unlike a filter routine, executes in its own thread context. The ithread routine can sleep or wait on a condition variable.

Mach ASTs are like Linux *softirqs*.[1,7,8] They are a powerful vehicle used mainly to perform any interrupt-related work not performed by the interrupt handlers. The AST mechanism allows one or more reason bits to be set for a processor or thread. Each bit represents a particular software trap. When a processor is about to return from an interrupt context, including returns from system calls, it checks for those bits and takes a trap if it finds one.

SCHEDULING

The mechanisms that Mac OS and FreeBSD use for scheduling have much in common.

FreeBSD provides restartable system calls (as Mach does) and separates wait channel priority from user mode priority. FreeBSD assigns a higher priority to the threads sleeping in the kernel because they typically hold shared kernel resources when they are awakened.

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. This dynamic priority-adjustment scheme favors interactive I/O-bound threads over compute-bound ones.

FreeBSD uses high-priority interrupt threads. 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. Because FreeBSD uses threads to implement bottom-half interrupt processing, interrupt handlers have their context and thus can sleep.

In terms of priority, the REALTIME class follows the ITHD scheduling class. Therefore, with the prerequisite of small and bounded overhead for the ITHD thread processing, engineers can design and implement real-time processing workloads with FreeBSD. The KERN class follows REALTIME in priority, and performs deferred interrupt processing. Time-consuming parts of interrupt service tasks should be implemented within the threads of the KERN class. TIMESHARE, the class that runs the “normal” user applications, follows KERN in priority. The kernel dynamically adapts the priority of the threads of this class to provide a 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 of the TIMESHARE class has two scheduling priorities: one for scheduling user-mode execution (top-half priority) and one for scheduling kernel-mode execution (bottom-half priority). Finally, the IDLE class consumes the CPU time when no useful task exists.

Xnu’s scheduling is implemented within the Mach subsystem, which schedules threads as FreeBSD does. Generally, the two systems follow similar algorithms and concepts. For time-sharing threads, Mach also uses adaptive adjustment of priorities according to the amount of computation they perform, favoring interactive tasks. Like FreeBSD, it organizes threads into priority queues (with a real-time priority class).

Xnu uses a tickless-style timer for interrupt processing. A deadline queue keeps the events that will trigger timer interrupts in ascending order. The scheduler produces the deadline queue. The tickless style avoids the overhead of processing many spurious timer interrupts, if the traditional periodic processing of timer interrupts (based on Hz frequency) is used.

VIRTUAL MEMORY

Xnu uses the Mach VM subsystem (with a few modifications). FreeBSD is also heavily based on Mach. Thus, VM implementation is similar in xnu and FreeBSD.

- Both systems isolate the implementation of the machine-dependent physical map to a *pmap* module. The machine-independent data structures that implement abstractions as virtual address space maps (*VM maps*), VM objects (*vm_object*), named entries, and resident pages are isolated to a *vmap* module.
- Because the entire virtual address space is not mapped at any given moment, the VM map is divided into several entries. Each entry (*vm_map_entry*) represents a virtually continuous block of mapped memory that shares common protection and inheritance attributes.
- The virtually continuous address space that a *vm_map_entry* represents can span multiple pages, but always has a single backing store (such as physical memory or a hard drive). The source of data is represented by a *vm_object*.
- Every *vm_map_entry* points to a chain of *vm_object* structures that describe sources of data (objects) that are mapped at the indicated address range. At the tail of the *vm_object* chain is the original mapped data object, usually representing a persistent data source, such as a file. Interposed between that *vm_object* and the map entry are zero or more transient shadow objects that represent modified copies of the original data. Both Mach and FreeBSD use submaps—for kernel address space only—to isolate and constrain address space allocation for kernel subsystems. Specifically, a kernel address space *vm_map_entry* can point recursively to a submap instead of a *vm_object*. This is useful for constraining memory allocations in the kernel.
- The lowest-level data structure that represents the physical memory being used by the VM system is described by a *vm_page* structure. Each *vm_page* is identified within the *vm_object* by its offset from the start of the object. A *vm_object* keeps a list of residents in physical memory pages, which are described by a *vm_page* structure.

In essence, a *vm_object* contains information about accessing memory from its source. Physical memory caches are the most frequently or recently accessed contents of the *vm_object*. Some or all of the VM objects' memory might not be resident in physical memory. Instead, they can reside in a backing store—for example, a regular file, a swap file, or a hardware device.

Mach's IPC features are unified with its VM subsystems, which leads to various optimizations and simplifications. Specifically, Mach's IPC implementation uses the VM subsystem to efficiently transfer large amounts of data using copy-on-write (COW) optimizations. Out-of-line (OOL) data transfer is an optimization for large transfers. The kernel allocates a memory region for the message in the receiver's virtual address space without making a physical copy of the message. The shared memory pages are marked COW.

SHARING RESOURCES IN MACH BECOMES A MATTER OF PROVIDING ACCESS TO THEIR CORRESPONDING PORTS.

Mach takes an object-oriented approach and uses a memory object for managing the backing store. The memory object is a Mach port to which messages can be sent by the kernel to retrieve the missing data.

Sharing resources in Mach becomes a matter of providing access to their corresponding ports. In Mach, tasks can send parts of their address spaces to one another in IPC messages.

The owner of a memory object is a memory manager, or a pager. Xnu implements the pager as a specialized task that supplies data to the kernel

TABLE 1. Unix *Byte* benchmarks (absolute counts of operations over 20 seconds).

Benchmark	Mac OS	FreeBSD iMac	FreeBSD HP PC	Linux HP PC
execl 20	7139	24110	33347	49469
syscall 20	12884110	10620877	14665300	28773938
spawn 20	35638	71768	120943	168546
context switching 20	1597858	2484023	3261091	2312248
hanoi 20	4603145	5025719	4575185	4364714
pipe throughput	25591689	44677193	39667978	45399029
mcopy 20	56937707	375264695	309153485	1906408815
threadsWithComputation no joining	38642	30279	28103	21126
threadsWithSmallComputation no joining	74551	48152	40782	106691
threadsPerformingAtomicOps	73579	47262	42472	107903
threadsComp 10 16 joining	21102	26025	27135	11194
threadsComp 10 32 joining	11098	14667	24244	5932
mallocMemCopy	1331726526	1243759142	802439723	572844647
fstime write 20	1686255	102547	187417	1211860
fstime read 20	1862444	1752057	512512	4140431
fstime copy 20	812984	96719	126403	862759

and saves modified pages selected for eviction to the backing store.

Departing from the Mach design that allows user-space implementation, Xnu realizes all pagers as kernel threads. Also, instead of messages, Xnu implements the pager interface with direct function calls. Therefore, FreeBSD and Mac OS theoretically can demonstrate similar performance concerning VM operations, as they utilize similar architecture and algorithms. Indeed, both Mac OS and FreeBSD displayed excellent performance with the *mallocMemCopy* benchmark (see Table 1). This benchmark exercises VM operations, as it allocates and accesses a very large area of dynamically heap-allocated memory.

In Mach, the pager communicates with the kernel and with the memory object port. The message-passing interface permits pagers to be completely implemented in user space. However, the involved overhead is significant, and two switches between user and kernel mode are required—as well as the housekeeping operations for the relevant message queues. FreeBSD also has pagers that perform similar chores, but they reside completely in kernel space and are accessible through direct routine calls (C functions).

Specifically, Mach handles page faults by having the kernel communicate with the memory manager by sending it a message requesting the missing data. The memory manager responds asynchronously by fetching the data from the backing store it is managing.

Although Apple has replaced much of its message passing with direct function calls, there remains strong port-based infrastructure. The rather poor performance of Mac OS at the *fork()* and *exec()* system calls can possibly be alleviated by replacing the rest of the port-based implementations from the VM subsystem with a FreeBSD-like direct procedure call implementation.

BENCHMARKS

We ran benchmarks on an iMac and an HP PC (both had equivalent hardware). We also evaluated the tests on a Linux iMac with OpenSuSE 42.3 Leap, Linux kernel version 4.4.76-1. The results we obtained were very close to the HP PC results (actually slightly better, but with no significant difference). This is expected from the similar hardware configuration of the two machines. Both the iMac and HP PC had a 3200 MHz Intel i5 processor with 8192 Mbytes of memory. It is interesting that Linux displayed the best overall performance, especially compared with other OSes such as Solaris and Windows.[9] FreeBSD was close to Linux in terms of performance, but Mac OS was notably slower at some benchmarks.

The first set of benchmarks we evaluated were from *Byte* magazine (github.com/kdlucas/byte-unixbench; see Table 1).

The *execl* column in Table 1 displays the average count of *execl* calls over 20 seconds, measuring the effectiveness of creating a process from an executable. Mac OS was about three times slower than FreeBSD at this benchmark. The process that executed with *exec* is the executable file of the *execl* benchmark program itself.

The *syscall* benchmark evaluates the overhead of performing system calls. Here, Mac OS was slightly faster than FreeBSD, but Linux was about two times faster than both. The *syscall* benchmark performs the following system calls sequentially: *close()*, *dup()*, *getpid()*, *getuid()*, and *umask()*. Xnu implements the Mach system calls in a conventional way without involving message passing.[1] Therefore, Mac OS obtained competent (and sometimes better) system call performance with OSes that have fast system calls, such as Linux and FreeBSD.

The *spawn* benchmark exercises the *fork()* system call and measures the efficiency of creating processes. Mac OS can be considered to have a slow *fork()* system call. Linux was the most efficient, followed by FreeBSD.

The *context switching* benchmark measures the number of times two processes can exchange an increasing integer through a pipe. The test program spawns a child process with which it communicates via a bidirectional pipe. FreeBSD had the best performance in terms of context switching times. Linux came close, but Mac OS was notably slower. This benchmark creates two pipes, p_1 and p_2. It then proceeds by forking a child process. The parent process closes the input descriptor of p_1 ($p_1[0]$) and the output descriptor of p_2 ($p_2[1]$). The child process does the reverse. Then, the two processes communicate using these pipes. The information from parent to child is transferred using the p_1 pipe, and the p_2 pipe transfers the information from the child to the parent.

The *hanoi* is a purely computational benchmark on which all OSes display similar performance (FreeBSD was slightly faster).

The *pipe* is the simplest form of communication between processes. Pipe throughput is the number of times (per second) a process can write 512 bytes to a pipe and read them back. FreeBSD performed more efficiently than Mac OS, and Linux displayed similar performance to FreeBSD.

The *mcopy* benchmark consists of code adapted from W.R. Stevens and S.A. Rago's *Advanced Programming in the UNIX Environment*.[10] It measures the performance of file-copying operations using shared memory. At this benchmark, Linux outperformed FreeBSD by about six times, and xnu was about six times slower than FreeBSD.

The *threadsWithComputation no joining* benchmark creates a large number of threads that perform some computation without joining their execution. The *threadsWithSmallComputation* benchmark is similar, but the amount of computation that each thread performs is significantly smaller. Mac OS outperformed FreeBSD, demonstrating excellent multithreading performance. It is interesting that Linux did not perform well with the computational threads—it does much better with threads involving light computations.

At the *threadsPerformingAtomic Ops* benchmark, the threads perform atomic (and thus possibly blocking) operations on counters. Mac OS outperformed FreeBSD in this threading performance test, and Linux was the overall winner.

The *threadsComp 10 16 joining* benchmark operates a 10-second loop that creates 16 threads and then joins with their execution before proceeding with the next iteration. The next benchmark with 32 threads is similar. At these two benchmarks, Mac OS and FreeBSD performed similarly and significantly better than Linux.

The *mallocMemCopy* benchmark allocates two memory buffers with the *malloc()* library function. It then copies a few bytes from one buffer to another simply to exercise the buffers. Finally, the buffers are freed with the *free()* library routine. At this benchmark, Mac OS outperformed the other systems, presenting the most efficient C-library heap allocation/deallocation performance.

The *fstime* benchmarks measure file system–related performance aspects. FreeBSD demonstrated rather poor performance at the *fstime* benchmarks for write operations, but this was due to the synchronous writes of directory operations to maintain consistency.

The next set of benchmarks we used was the *lmbench* set (see Table 2).[11] The simple *syscall* measures a nontrivial entry into the kernel by repeatedly writing one word to /dev/null, a pseudo-device driver that does nothing except discard the data. At this benchmark, all OSes were nearly equivalent.

The *simple read*, *simple write*, *simple open/close*, *simple stat*, and *simple fstat* benchmarks measure the efficiency of the corresponding operations. Here, Mac OS and FreeBSD performed similarly, but Linux was most efficient.

TABLE 2. Lmbench benchmarks (in microseconds).

Benchmark	Mac OS	FreeBSD iMac	FreeBSD HP PC	Linux HP PC	Linux iMac
simple syscall	0.1061	0.1015	0.11098	0.1022	0.071
simple read	0.3750	0.1670	0.19080	0.1663	0.145
simple write	0.3222	0.1374	0.16430	0.1318	0.090
simple stat	1.1455	1.2639	3.23270	0.5105	0.560
simple fstat	0.3858	0.2521	0.53070	0.1098	0.170
simple open/close	2.1956	1.6104	3.65570	1.0097	1.070
select on 10 fds	1.0893	0.3352	0.46590	0.2411	0.270
select on 100 fds	18.6181	2.8129	8.76470	0.8377	0.760
select on 10 tcp fds	0.9444	0.3772	0.41380	1.2411	1.280
select on 100 tcp fds	17.1393	5.0446	6.15760	2.2007	1.810
signal handler installation	0.2382	0.1904	0.19470	0.1269	0.155
signal handler overhead	1.6820	1.3483	1.47840	0.6557	0.950
page fault	4.4007	0.0300	0.01190	0.1785	0.330
pipe latency	5.7449	3.2599	3.02500	4.3439	3.540
process fork+exit	262.9545	95.1455	105.24000	81.5161	90.730
process fork+execve	1859.0000	335.5333	401.70650	264.9545	460.230
process fork+/bin/sh	3081.5000	757.2500	1348.88610	1398.0000	4426.400
AF_UNIX sock stream latency	7.3108	3.5611	3.79670	4.0548	3.390
UDP latency using localhost	18.1897	6.7410	7.66220	8.0423	4.200
TCP latency using localhost	18.6043	7.3267	9.05000	9.6713	6.260
pipe bandwidth	5301.4400	9951.7100	9252.23000	13251.0500	6942.800
TCP bandwidth	1421.8100	2182.8100	2276.84000	5291.8800	3769.900
AF_UNIX socket bandwidth	1421.8100	2182.8100	2276,84.00000	5291.8800	14107.050

Here, pipe bandwidth is measured by creating two processes: a writer and a reader, which transfer 50 Mbytes of data in 64-Kbyte transfers. TCP bandwidth is measured similarly but using 1 Mbyte page-aligned transfers.

The *process fork+exit* and *process fork+execve* tests measure the number of times a process can fork and reap a child that immediately exits. Process creation refers to creating process control blocks and memory allocations for new processes, so this applies directly to memory bandwidth. Typically, this benchmark would be used to compare various implementations of OS process-creation calls. The shells script (for example, *process fork+/bin/sh*) test measures the number of times per minute a process can start and reap a set of one, two, four, and eight concurrent copies of a shell script where it applies a series of transformations to a data file.

Unix pipes create a one-way byte stream on which one process has a write descriptor and the other has a read descriptor. TCP sockets differ in that they are bidirectional and can cross machine boundaries.[12]

The interprocess communication latency benchmarks (AF_UNIX socket stream latency, UDP latency using localhost, and TCP latency using localhost) measure the time it takes to pass a small message (perhaps a byte in size) back and forth between two processes. The microseconds needed to make one round trip are reported in Table 2. At this benchmark, FreeBSD was the most efficient, while Mac OS was rather slow.

The majority of OSes use a well-defined interface for each subsystem and direct function calls. However, Mach organizes its subsystems as objects and is based exclusively on message passing. Therefore, Mach's objects cannot directly invoke one another. This design is definitely more modular but, as our benchmark results illustrated, can impose significant overhead on some operations. Apple made a lot of modifications to the original Mach design to improve performance, and our results demonstrate that Apple has rather successfully achieved this goal. However, Mac OS is significantly slower on process creation (*fork*) and at the execution of a binary image (*execl*). Also, the system call overhead is higher compared to FreeBSD and Linux. However, xnu outperforms on multithreading performance and on heap-allocated memory operations. Because most modern applications are structured on threads instead of processes, the outstanding multithreading and dynamic memory allocation performance of Mac OS has great practical benefits. Mac OS's port-based message passing is the main operation that makes it significantly slower than FreeBSD. It is expected that Apple will replace these functionalities with implementations based on direct procedure calls in future versions of Mac OS. In this aspect, the xnu kernel will be as fast as the efficient FreeBSD and Linux monolithic kernels, retaining the high modularity of structuring the implementation on top of the Mach microkernel.

REFERENCES

1. J. Levin, *Mac OS and iOS Internals: To the Apple's Core*, Wrox, 2012.
2. A. Singh, *Mac OS Internals: A Systems Approach*, Addison-Wesley Professional, 2016.
3. O.H. Halvovsen and D. Clarke, *OSvX and iOS Kernel Programming*, Apress, 2011.
4. M.K. McKusick, G.V. Neville-Neil, and Robert N.M. Watson, *The Design and Implementation of the FreeBSD Operating System*, 2nd ed., Addison-Wesley Professional, 2014.
5. A.S. Tanenbaum and Herbert Bos, *Modern Operating Systems*, 4th ed., Pearson, 2014.
6. R. McDougall and J. Mauro, *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture*, 2nd ed., Prentice Hall, 2006.
7. D.P. Bovet and M. Cesati, *Understanding the Linux Kernel*, 3rd ed., O'Reilly Media, 2005.
8. R. Love, *Linux Kernel Development*, 3rd ed., Addison-Wesley Professional, 2010.
9. S. Papadimitriou and L. Moussiades, "A Comparative Evaluation of Core Kernel Features of the Recent Linux, FreeBSD, Solaris, and Windows Operating Systems," *Proc. World Congress on Eng.* (WCE 16), 2016; www.iaeng.org/publication/WCE2016/WCE2016_pp387-392.pdf.
10. W.R. Stevens and S.A. Rago, *Advanced Programming in the UNIX Environment*, 3rd ed., Addison-Wesley Professional, 2013.
11. L. McVoy and C. Staelin, "lmbench: Portable Tools for Performance Analysis," *Proc. 1996 USENIX Ann. Technical Conf.* (ATEC 96), 2006; www.usenix.org/legacy/publications/library/proceedings/sd96/full_papers/mcvoy.pdf.
12. W.R. Stevens, *UNIX Network Programming, Volume 1: The Sockets Networking API*, 3rd ed., Addison-Wesley Professional, 2003.

ABOUT THE AUTHORS

STERGIOS PAPADIMITRIOU is a professor of computer science in the Department of Informatics and Computer Engineering at the Technology Education Institute of Eastern Macedonia and Thrace. His research interests include OSes, programming languages, scientific computation, bioinformatics, and machine learning. Papadimitriou received a PhD in computer engineering and informatics from the University of Patras. Contact him at sterg@teiemt.gr.

LEFTERIS MOUSSIADES is an associate professor of computer science in the Department of Informatics and Computer Engineering at the Technology Education Institute of Eastern Macedonia and Thrace. His research interests include graph algorithms, programming languages, OSes, and web application development. Moussiades received a PhD in computer science from the Aristotle University of Thessaloniki. Contact him at lmous@teiemt.gr.