1 Feature Description

1.1 Introduction

This document is an extension and enhancement of the previously issued MK5.0 Release Notes and covers the modifications and enhancements to the MK system from MK5.0 through MK6.3, MK7.1, MK7.2, and MK7.3. The MK7.3 release comprises results from a number of activities and projects at the The Open Group Research Institute (RI).[1] These are:

a. The general evolution of the microkernel interface.

b. The MK6 performance improvement project, which has contributed the server collocation component of the release.

c. The MK6 device driver asynchronous I/O model, which allows task notification directly from interrupt context.

d. The NMK18 NoRMA project, which has contributed enhancements to the multicomputer support component of the release.

---

1. The Research Institute was previously known as the Open Software Foundation Research Institute, prior to the merge of the Open Software Foundation and X/Open to form The Open Group.

e. The RT2 real-time project (MK6.1), which has contributed fundamental extensions to the kernel in support of real-time applications and environments.

f. The MicroKernel Performance Suite (MKPS).

g. The Event Trace Analysis Package (ETAP), which is designed to provide users with tools to study system behavior.

h. IP Multicast, which allows the transmission of an IP datagram to a “host group”, a set of zero or more hosts identified by a single IP multicast destination address.

i. Additional device drivers for the parallel port, DEC DE-450 Etherworks PCI card, the Myrinet PCI board, PCI SCSI adapters, DEC FDDI PCI board and the SoundBlaster ISA audio card.

Each of these sections, except for the general evolution of the interface, is described in greater detail in the following sections.

1.2 New and Enhanced Features in MK7.3

MK7.3 contains a number of specific new features, as well as enhancements of already existing features, including:

a. CORDS project enhancements, which, among other things, consolidate previously released capabilities.

b. Clock and timer extensions and improvements, increasing the stability of the NTP distributed clock, as well as providing faster access to a more accurate and precise system clock.

c. Debugger enhancements for multi-threaded programs.

d. Access to a wider variety of real-time services through POSIX specified interfaces[30], including a substantial portion of the POSIX thread and real-time features.

e. Initial implementation of the MK Scheduling Framework and a set of scheduling policies, including POSIX-defined policies and an experimental, Alpha OS-derived best-effort processor scheduling policy.

f. An experimental, microkernel-provided, trans-node RPC facility.

g. Myrinet is now supported, both as a local network interconnect and as an integrated part of the CORDS subsystem.

h. Restructuring and clean-up in several areas: thread control, preemption and locking, and the stack-per-shuttle optimization.

i. Inclusion of a number of test programs, including an interrupt latency benchmark.

These features are described in the following sections.

1.2.1 CORDS Project Enhancements

The MK7.3 release consolidates the CORDS technologies that were first offered with MK7.2. These technologies are GIPC (Group IPC) and its companion set of applications, which exercise group communication, fault isolation, and fault management properties.

GIPC's inner protocol constituents have been substantially hardened. For instance, protocols involved in maintaining node liveness have been enhanced with "skeptics"; skeptics recognize and mask status changes that occur too often within a given interval, due to failure modes (e.g., hardware flickers) or malicious behaviors. The membership protocol has also been improved to more quickly converge to a stable membership view, even when some of the nodes are not responsive and need to be evicted from a proposed membership view. The lower layers of GIPC protocol graphs have been adapted to support multiple interconnects (e.g., Ethernet and Myrinet); while two or more interconnects are still mutually exclusive, we anticipate taking full advantage of redundant interconnects for superior failure resilience. GIPC's API has not changed from MK7.2.

The two demonstration applications delivered with GIPC have also been revamped to improve their efficacy. The marker demonstration (see the man page for *demo_client*(8)) is now a complete outline of a distributed application whose constituents communicate using data sets that depend upon the agreed actual population of constituents; the demonstration proves the effectiveness of GIPC's virtual synchrony model, the viability of GIPC API and locking within a multi-threaded GIPC client. The ball-handler demonstration (see the man page for *balldemo*(8)), which requires a special mechanical apparatus, takes advantage of the experience achieved in offering the demonstration at several locations outside of the RI. For more information on the ball-handler demonstration, see:

http://www.opengroup.org/RI/ri/AdvisBrd/Oct96_Mtg/present/ft-demo/ftdemo.htm

The CORDS framework has been restructured to use either of two thread packages: *cthreads* or POSIX threads (*pthreads*). While *cthreads* is still the default choice of thread model, we anticipate shifting our focus to *pthreads*, due to its real-time viability. (Conversely, *cthreads* circumvents kernel scheduling decisions.) For further information on the use of *pthreads* in CORDS, please contact Franco Travostino (f.travostino@opengroup.org).

In-kernel CORDS configurations compiled with XK_ETAP can now be traced with ETAP. (There is a unique ETAP event for all CORDS activities). As part of the 'nosupport' deliverables, we introduce our experimental toolkit for off-line reconciliation of ETAP trace logs from different nodes (Distributed ETAP, or DETAP). The enabling technology for DETAP is CORDS' distributed clock synchronization facility (LTS).

1.2.2 Clock and Timer Extensions and Improvements

MK7.3 provides a number of improvements in the area of clocks and timers, including increased stability of the NTP[19] distributed clock, as well as faster access to a more accurate system clock. These improvements are documented in [20].

One objective of this work was to improve the clock OS support code to deal with the rather loose crystal accuracy characteristics evident in typical machines.

Another objective was to investigate and resolve problems that some users of the MK system had observed. These included: (1) reported time intervals that were impossibly small or negative; (2) inadequate NTP synchronization; and (3) excessive overhead to read the clock.

This last area, providing low-overhead, high-resolution clock reading, was particularly interesting given the fact that newer microprocessors, including the Pentium, have some form of "cycle counter" — i.e., a large register that counts at a high rate — that looks like a very good candidate for a timekeeping device. And, on the Pentium at least, this register can be made readable in any mode (including user) and so seems to offer the prospect of a clock that can be read with very low overhead.

MK7.3 offers the results of this work. Among the highlights:

a. The primary cause of anomalous time events (negative intervals) was a logic flaw in the 8254 Programmable Interval Timer implementation on one particular hardware platform. MK7.3 includes a software workaround.

b. The assumed frequency of the 8254 clock source is now a dynamically adjustable attribute.

c. A long-term feedback loop utilizing NTP tunes the 8254 frequency attribute to within 1 part-per-million.

d. The UNIX server now separates and modularizes UTC (Coordinated Universal Time) variables from those variables involved in local real-time timekeeping. Moreover, UTC-awareness has been removed from the kernel, and the execution time of *gettimeofday()* has been reduced by almost 50%. The execution time of the POSIX-compliant function *clock_gettime()* has been reduced by about 99%, and this is the recommended method for all applications to get the current time of day.

e. The Pentium Time Stamp Counter is now utilized as a very high precision, very low overhead, real-time clock device offering the same long-term accuracy. The result is a system clock with precision, speed, and accuracy heretofore only available with special purpose hardware.

As is described elsewhere in this chapter, all of this work complies to the current POSIX standard for clocks and timers.

1.2.3 Debugger Enhancements for Multi-Threaded Programs

Earlier MK7 releases provided separate versions of the ‘gdb’ debugger for use with single-threaded programs (based on *ptrace*) and for multi-threaded programs (where ‘gdb’ must be MK aware). For MK7.3, these functions have been merged so that there is now a single version of ‘gdb,’ which is both *ptrace* and MK-aware, and which can be found in:

- /usr/bin/gdb

It is based on the Free Software Foundation’s ‘gdb’ version 4.16 and works with executables compiled by ‘gcc’ version 2.7.1, which is the version of ‘gcc’ that has been used with MK7.2 and MK7.3.

Conceptually, this version of ‘gdb’ executes in two distinct modes. By default it executes as *ptrace* ‘gdb,’ and it can be used in the traditional manner (without requiring special privileges or additional support) to debug single-threaded programs.

However, to use this version of ‘gdb’ to debug multi-threaded programs, the user does need additional privilege and support. In particular, the user must be ‘root’ and the ‘machid’ program must be running. Then, after starting ‘gdb,’ the user should issue the following command:

- (gdb) target mach

At this point, for executables linked with ‘libmach,’ the user can issue the ‘run’ command to start the program. For those executables that have not been linked with ‘libmach,’ the ‘run’ command should not be used. Instead, ‘attach’ should be used. (Multi-threaded executables will probably have been linked with ‘libmach.’)

You can use the standard ‘gdb’ commands. Additional commands specific to multi-threading include:

- “tl” — for thread list;
- “ts” — for thread select;
- “thread-tree” — to print threads as a stack tree;
- “help mthread” — to list all of the multi-thread subcommands.

1.2.4 POSIX-Compliant Access to Real-Time Services

The RI is working to advance the state-of-the-art of scalable, embedded real-time applications by developing an operating system for scalable embedded real-time systems that complies to the real-time POSIX standard[30] and that builds upon the proven MK Microkernel, which provides scalability, performance, real-time, and instrumentation capabilities.

The POSIX series of standards specifies application and user interfaces to open systems. They define the application interfaces to basic system services for input/output, file system access, and process management. They also include interfaces appropriate for real-time applications and interfaces appropriate for multi-threaded applications. (All of the POSIX standards cited here correspond to interfaces expressed in the C programming language.)

We have identified a functional subset of these POSIX interfaces that will provide a small, simple, and *fast* standards-subset OS environment specifically targeted at scalable, embedded real-time applications. The POSIX real-time services will include most of the thread, memory management, clocks and timers, and synchronization subsystems.

We are providing POSIX services in two phases. The first phase provides a POSIX threads run-time library, which is included in MK7.3 and is described below.

The second phase, which will be provided in future operating system releases, will produce a more full-featured POSIX facility, utilizing one or more servers operating on top of the MK Microkernel. The resulting system will support distributed, fault-tolerant, real-time applications, employing MK Microkernel facilities to provide POSIX-compliant services in the most advantageous manner.

POSIX standards define a number of conforming implementation options, which correspond to groups of POSIX routines — for example, _POSIX_PRIORITY_SCHEDULING or _POSIX_REALTIME_SIGNALS. These implementation options allow a given POSIX implementation to maintain POSIX conformance while providing only a subset of the POSIX-specified routines (as opposed to *all* POSIX-specified routines). Presumably, a subset is provided to satisfy the requirements of some intended audience.

In MK7.3, none of these POSIX conformance options have been set because we do not conform to the POSIX specification at this time. However, for those routines that we have implemented, the interfaces comply to the POSIX specification, as do the resultant behaviors — to the extent that is currently reasonable.

1.2.4.1 The POSIX Threads Run-Time Library

The POSIX standard[30] specifies support routines for multi-threaded applications in a real-time environment. We have implemented a considerable subset of these routines, and have chosen to implement them directly on top of the MK Microkernel for two reasons. First, the POSIX thread model is very similar in nature to the native thread model found in the MK system. Secondly, having this close relationship between POSIX threads and MK Microkernel threads allows us to exploit fully the technologies we are creating for real-time systems. In this way, the POSIX thread library can take advantage of such things as migrating threads and the real-time scheduling framework.

Currently supported are:

- threads
- thread attributes (with some limitations)
- mutex variables
- mutex variable attributes
- condition variables
- “once” variables
- thread-specific data

Some POSIX-specified features have been left out so that we can support POSIX-style thread programming in the absence of a complete POSIX (server) environment. Features that require server support will be implemented as the POSIX Server becomes available in future system releases.

For this first implementation of the POSIX Threads Run-Time Library, we do not support:

a. thread-level signals

   Since the MK real-time system philosophy is based on cooperating threads using RPC and shared memory for communication, signals are not needed; and, in fact, they conflict with the provision of end-to-end real-time performance and time management.

b. sharing of thread data structures between separate tasks

   The POSIX standard allows for mutex and condition variables to be shared between separate tasks. We don't support this in order to provide a more efficient implementation of these synchronizers.

c. "heavyweight" POSIX *fork()* support

   Currently we support a single task, multiple thread application model. We will add *fork()* support if a multiple task model requires it.

d. thread stack attributes

   The POSIX standard supports thread attributes which control the size and placement of the run-time stack for the thread. In this implementation, we provide for a single system wide control over the size (granularity) of thread stacks, but not their placement. This is primarily to allow for a more efficient implementation of thread-specific data structures used by the library itself.

e. making the POSIX run-time library "thread-safe" and "thread-friendly"

   We intend to provide a fully thread-safe and thread-friendly POSIX run-time library in the future. This library will continue to comprise a standards-compliant subset of the full POSIX specification [30].

1.2.4.2 Additional Notes Concerning the POSIX Threads Run-Time (*pthreads*) Library

a. There are actually two *pthreads* libraries available in MK7.3. The new *pthreads* library described above has header and library files in the "standard" places, specifically:

   - /usr/include/pthread.h
   - /usr/ccs/lib/libpthread.a

   In addition, there is an older *pthreads* library, corresponding to Draft 4 of the POSIX System Application Program Interface, which constitutes part of the OSF/1 API that was added to support DCE (Distributed Computing Environment). The older *pthreads* library is based on OSF/1 system calls rather than MK microkernel primitives. This has certain consequences, notably that it is available to unprivileged programs, where microkernel system calls are not.

The corresponding header and library files for the older version of the *pthreads* library are found in:

- /usr/include/posix_draft_4/exc_handling.h
- /usr/include/posix_draft_4/pthread.h
- /usr/include/posix_draft_4/pthread_exc.h
- /usr/ccs/lib/posix_draft_4/libpthreads.a

b. The new *pthread* routines comply to POSIX specifications, with these routines currently unimplemented:
pthread_atfork()
pthread_attr_getscope()
pthread_attr_setscope()
pthread_attr_getstackaddr()
pthread_attr_getstacksize()
pthread_attr_setstackaddr()
pthread_attr_setstacksize()
pthread_condattr_getpshared()
pthread_condaddr_setpshared()
pthread_kill()
pthread_sigmask()
pthread_mutexattr_getpshared()
pthread_mutexattr_setpshared()

c. The new *pthreads* library does not currently offer 32 priority levels for real-time system configurations. Rather, in MK7.3, 30 priority levels are available for user threads. This deviation from the POSIX specification will be addressed in future releases.

d. Access to the Best-Effort scheduling policy has not yet been integrated into the *pthreads* library routines. For the time being, applications should use the microkernel-level '*thread_set_sched()*' system call to access the Best-Effort scheduling policy. Best-Effort support will be integrated into the *pthreads* library in a future release.

e. Do not use '*mach_thread_self()*' to get the kernel port for a *pthread* because this will generate a thread reference that the *pthread* library does not know about and can result in resource leakage. Rather, call *_pthread_kernel_port(pthread_self())*.

f. If possible, MK users should use the *pthreads* library, rather than either *cthreads* or *rthreads*, since this is the library on which future work will focus. Previously, real-time users were advised to use *rthreads*. Many of those users should be able to transition to *pthreads* now, but some cannot due, for instance, to the fact that *rthreads* is currently more integrated with the migrating thread abstraction than is *pthreads*.

1.2.4.3 Clocks and Timers

MK7.3 features additional and improved clock and timer functionality relative to previous releases, providing users with higher clock resolution, lower overhead to access the clock, and more accurate time. (This work is described in more detail elsewhere in this chapter.)

As of this release, all implemented clock and timer functions comply with the POSIX specification. Not all of the specified functions in the clocks and timers chapter are implemented, so _POSIX_TIMERS is not set. In particular, the following clock and timer functions are currently unimplemented:

clock_getres()
nanosleep()

1.2.5 MK Scheduling Framework and Scheduling Policies

MK7.3 includes the initial implementation of the MK Scheduling Framework and a set of scheduling policies, which comprises not only POSIX-specified processor scheduling policies, but also a time-driven Best-Effort scheduling policy derived from work done on the Alpha Operating System, which performed research into the support of distributed, fault tolerant, real-time applications[14][9].

MK7.3, including the new Scheduling Framework, is as robust as its predecessors with respect to scheduling. That is, the introduction of the Scheduling Framework and its attendant system restructuring has not decreased system robustness based on the RI's testing to date.

Nonetheless, this work is ongoing, and it is anticipated that the Scheduling Framework interfaces will continue to evolve as more scheduling policies are implemented and as richer scheduling capabilities are pursued. Similarly, the scheduling policy implementations might evolve in the future, particularly those for new policies such as Best-Effort. Despite this evolution, though, the existing POSIX-defined scheduling policies are expected to remain stable and are not expected to display any user-visible changes.

The MK Scheduling Framework and its policies have not been tested on SMP platforms.

1.2.5.1 The Scheduling Framework

The MK scheduling subsystem, which provides the policies and mechanisms to schedule and control threads, has been restructured in MK7.3 to incorporate the MK Scheduling Framework to:

a. separate scheduling policies from scheduling mechanisms,

b. support multiple independent scheduling policies concurrently, and

c. enable new functionality for real-time support.

The Framework provides access to scheduling policy interfaces for the kernel and provides kernel mechanisms for use by scheduling policies. These mechanisms, which include, for example, context switching, run queue manipulation, and timer facilities, do not change when different scheduling policies are installed. These mechanisms can themselves be replaced but they are considered policy independent.

The Framework design is based on the philosophy that each policy should be treated as a black box. A policy exports its interfaces to the Scheduling Framework which, in turn, provides access to the policy for the rest of the kernel and the other policies.

To mediate interactions among policies, each policy associates an attribute that is universally understood within the framework with each thread it schedules. This attribute is a *metapriority*, an integer value from a bounded range that gives the kernel a low-overhead mechanism for evaluating the relative importance of two threads, even those scheduled by different policies.

The Framework always executes the ready thread with the best metapriority. Whenever a ready thread's metapriority is better than that of the currently running thread, the Framework preempts the current thread.

Policies can have overlapping metapriority ranges, although no metapriority can be shared by two or more policies. The purpose of a policy's metapriority range is to allow the relative importance of a policy's threads to be easily configurable. Allowing overlapping ranges precludes a policy from implicitly locking out the threads of another policy.

Each scheduling policy is represented by a *policy object*, which defines the interfaces (including a small amount of shared data) between the policy and the framework. The framework, in turn, provides access to these interfaces for the rest of the kernel.

Each policy object is characterized by the following policy-specific data:

- policy ID— A unique policy number.
- priority mask — An explicit indication of the metapriorities managed by the policy.
- scheduling attributes size — The size of the scheduling parameters data structure for this scheduling policy. These are the parameters that are actually communicated — and meaningful — outside the policy. (This is as opposed to the internal state, which has some overlap with the scheduling parameters.)
- scheduling information size — The size of the scheduling policy's internal state that describes the state of an individual thread.
- standard scheduling operations — The set of policy-specific operations that correspond to the general interfaces for policies exported by the MK Scheduling Framework.

The standard scheduling operations provided by a scheduling policy include:

- *sp_thread_attach()* — Associate a thread with a scheduling policy.
- *sp_thread_detach()* — Disassociate a thread from a scheduling policy.
- *sp_thread_begin()* — Notify the policy that a new thread has been selected to run
- *sp_thread_done()* — Notify the policy that the current thread is done
- *sp_thread_dispatch()* — Notify the policy that an old thread is ready to be requeued.
- *sp_thread_unblock()* — Notify the policy that a thread has become runnable
- *sp_thread_set()* — Set a thread's scheduling attributes.
- *sp_thread_get()* — Get a thread's scheduling attributes.
- *sp_alarm_expired()* — Handle an expired alarm condition.

1.2.5.2 Scheduling Policies

The Timesharing, FIFO, and Round Robin scheduling policies that were previously supported on MK systems continue to be supported in MK7.3 systems, and are referred to as the *MK Scheduling Policies*. Kernel and UNIX server threads utilize the MK Scheduling Policies.

The FIFO and Round Robin policies correspond to the 'SCHED_FIFO' and 'SCHED_RR' policies described in the POSIX standard[30].

In addition to the MK Scheduling Policies, MK7.3 contains a time-driven scheduling policy called the Best-Effort Scheduling Policy. This policy is based on previous implementations in the Alpha Operating System[16][21]. It is intended to be useful in a dynamic real-time system, where processing overloads can occur. It attempts to provide a mechanism for graceful degradation during periods of overload.

The Best-Effort Scheduling Policy bases its scheduling decisions on time constraints that are explicitly declared by an application. The application also supplies estimates of the expected execution time needed for each time-constrained computation. In the event that there are sufficient processing cycles to satisfy all specified time constraints, then all threads are scheduled in "deadline order." However, if a processor is overloaded, then threads are shed based on a heuristic that attempts to maximize the value accrued by the application (as specified by the time constraints in application-specific units).

The following items should be considered when using multiple policies in the framework:

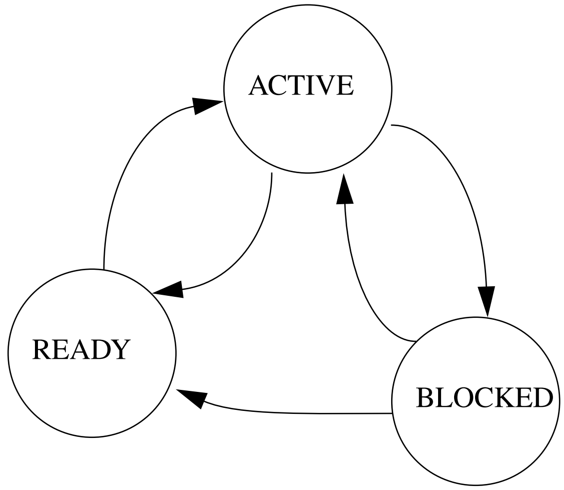
a. The MK Scheduling Policies, which are needed to run the UNIX Server, are always present.
b. Only one real-time policy — that is, a policy that bases scheduling decisions on explicitly specified application time constraints, such as Earliest Deadline First or Best-Effort — should be active. (Note that, by this definition, none of the MK Scheduling Policies are real-time policies.)

1.2.5.3 Scheduler State Machine

The original Mach 3 scheduling state diagram was large and complex. The separation of scheduling mechanisms from scheduling policies implemented in MK7.3 partitions some of the scheduling state machine into the scheduling mechanisms, which are policy independent. Consequently, policy designers need only concern themselves with that portion of the state machine that varies from policy to policy.

Conceptually, the Scheduling Framework provides a scheduling policy with a simplified scheduling state machine in which threads can be in one of only three states:

- ACTIVE — running on a processor. *thread_dispatch()* moves the thread into BLOCKED if WAIT is set; otherwise, it moves it into READY.
- READY — runnable. *thread_invoke()* moves the thread into ACTIVE.
- BLOCKED — not runnable. *thread_setrun()* moves the thread into ACTIVE if there is an idle processor; otherwise it moves it into READY.



Thread State Transitions

1.2.6 Experimental Trans-Node RPC

MK7.3 has introduced support for Remote Procedure Calls (RPCs) that cross node boundaries. This mechanism is based on the notion of “migrating threads,” whereby scheduling (and other) information associated with the client thread flows along with any data to the server[4].

The server thread, possibly on another node, will then assume the scheduling characteristics of the client thread for the duration of the RPC function.

The current version uses KKT (Kernel-to-Kernel Transport) as the transport between the two nodes. Also currently required is a “DIPC” configuration for support of multi-node systems and communication paths between them.

1.2.7 Myrinet Support

MK7.3 introduces support for Myrinet. Like Ethernet and FDDI, Myrinet has been integrated with CORDS and can be used as an interconnect for CORDS protocols. In addition, Myrinet can also be used without CORDS as an another computer interconnect.

Information about Myrinet is available from Myricom at:

http://www.myri.com

1.2.8 Restructuring and Clean-Up

Although these changes should be generally invisible to users, several areas — including thread control, preemption and locking, and the association of a single stack per shuttle — have been made in MK7.3.

The thread control operations have been redesigned to work more reliably, to simplify the scheduling state machine, and to better support RPC. In order for thread control operations to behave reliably, they only effect the user visible state of a thread. A thread must be in user mode or blocked in a "user visible" condition before thread control operations can be applied.

Changes in thread attributes (e.g. registers, scheduling attributes) will be processed immediately if the thread is in the correct state. Otherwise, the requester will block until the thread reaches an acceptable state. Changes in thread state (e.g., *thread_suspend()*, *thread_abort()* and alerts along an RPC chain) will also block until the thread is in the appropriate state.

To improve both space-efficiency and performance, there is now a single kernel stack associated with each shuttle — that is, the schedulable portion of a migrating thread. Previously, there had been a kernel stack associated with each activation comprising a migrating thread.

1.2.9 Test Programs

Several test programs have been included in MK7.3. Typically, these test basic system features, such as kernel synchronizer performance or the correctness and performance of the *pthreads* library functions. There is also a test that measures the interrupt latency for the system.

1.3 Collocated Servers

Most of the performance penalties associated with microkernel systems result from locating an OS server in user space and using a general-purpose IPC message mechanism to implement RPC, thus adding overhead to the system call path. As such, the principal strategy for improving MK performance has been to move the OS server task back into the kernel address space and to optimize the RPC mechanism for the case where the client and server share an address space.

Collocation means locating separately-compiled and linked programs in the same address space. In the current implementation of collocation, each program runs in a separate task. This corresponds to the original model of collocating several tasks in the same address space. However, the model is evolving in the direction of having several separately compiled and linked programs loaded into a single task.

When programs share the same address space, it becomes possible to replace IPC message-based RPC with a form of direct procedure call that is called *short-circuited RPC*. This is many times more efficient than message passing, and accounts for the largest single performance improvement. One of the requirements for this project was to maintain the modularity of the MK design. In the case of RPC, this means that collocated programs must not be aware that they are communicating via short-circuited rather than message-based RPC.

More information can be found in a Research Institute paper [10].

1.4 Asynchronous Driver Model

In earlier versions of MK, asynchronous I/O completion notifications were posted to a task using a reply message generated by the microkernel. Because messages cannot be sent from interrupt context, an `io_done` thread lives in the microkernel to perform the messaging function. This `io_done` thread introduces an additional thread context switch, with the accompanying performance penalty, on each asynchronous I/O completion. A new mechanism has been introduced in the microkernel by which any server can circumvent the interposition of the `io_done` thread on each asynchronous I/O completion.

Instead of waiting on a port set for I/O completion requests, a task waits on a new kernel object called an `io_done_queue`. Applications such as the UNIX single server create a single `io_done_queue` at server initialization time. The corresponding wait operation returns the I/O completion status. For each wait invocation, the status of one I/O request is returned. If there are no pending I/O completions the wait operation blocks in the kernel on the address of the completion queue. The microkernel, from interrupt context, enqueues (FIFO) completions on the completion queue and posts a wakeup on it for each I/O completion. Completion processing previously handled by the `io_done` thread is now done by the task thread when it awakens.

1.5 Real-Time Extensions

This release is a continuation of the effort by the RI to provide support for real-time applications and environments. The major areas of real-time functionality provided in MK7.3 are described in this section.

1.5.1 Preemptible Kernel

One of the projects undertaken by the RI as part of the kernel interface unification effort is the design and implementation of several changes and enhancements to MK to improve event response and context switch latency. These modifications include the following[1]:

- Implementation of high-performance blocking locks to replace most simple spinlocks.
- Elimination of all complex blocking locks, which can cause arbitrarily long scheduler invocations due to the "thundering herd" wakeup effect. (MK, like UNIX before it, has a feature wherein an arbitrary number of processes/threads can wait for a particular event to occur; when it does, they are all awakened simultaneously; hence, the thundering herd.)

1.5.2 Synchronizers

MK7.3 supports two types of synchronizers: semaphores and mutual-exclusion locks (see [5] for more information).

Semaphores are a simple and powerful synchronization primitive that can be used to implement a host of other synchronization protocols. The semaphore abstraction is particularly well suited for inter-task producer/consumer style communication. Tasks can dynamically create and destroy semaphores as the need arises; additional operations include signalling, and waiting on, a semaphore. Semaphores are named by port rights, enabling operations on semaphores to be network transparent.

Locks are first class microkernel entities, which can be created and destroyed, as well as locked and unlocked. Lock operations are performed against *lock sets*. A lock set represents either a single lock or a collection of associated locks. Lock sets are named by ports, to allow them to be referenced across task boundaries; while they are owned by tasks, the locks are owned by threads. The thread that acquires a lock becomes the owner of the lock. A lock can only be released by the thread that owns the lock. Lock ownership may be *handed-off* between thread.

1.5.3 Real-Time Threads (*rthreads* and *pthreads*)

The RT Threads Library is a user-level library and is not, therefore, documented in the Kernel API Changes chapter of the Release Notes. See [5] for a complete description of the library functionality.

---

1. Currently, kernel preemption only works on uniprocessor systems.

The POSIX threads (*pthreads*) library, not the *rthreads* library, is the focus of current and anticipated future development by the RI in this area. Consequently, the RI advises users to switch to *pthreads* where appropriate and discourages the use of *rthreads* when it can be avoided.

1.5.4 Extended Real-Time Thread Control Semantics

The thread_suspend function blocks the targeted thread to stabilize its user visible state. The successful return from thread_suspend guarantees the caller that the targeted thread is no longer executing user code.

If a suspended thread is involved in a system call that changes its user visible state, the state retrieved will always be consistent. Under these circumstances, if the suspended thread's state is changed by another thread, it is intended that the controlling thread's changes should supersede those of the suspended thread.

In the circumstances above, if a controlling thread suspends another thread, retrieves user visible state of the suspended thread, modifies the state, and resumes the thread, either of the following is possible:

- the suspended thread's changes to itself haven't taken effect:
  the retrieve returns the previous values; the modify updates the values; after the resume, the target will return from its system call indicating that the call was aborted

- the suspended thread's changes to itself have taken effect:
  the retrieve returns the values it set; the modify updates the values; after the resume, the target will return from its system call indicating success

In either case, the modifications specified by the controlling thread should be effective after the suspended thread is resumed.

1.5.5 RPC Kernel Interface

Recent work on existing systems indicates that message based systems that optimize the Remote Procedure Call (RPC) case experience significant performance improvements. This work leverages the collocation effort and the redefinition of a thread into two components: a thread and an activation.

Perhaps the most important difference between the RPC mechanism and Mach IPC based RPC is the treatment of message meta data. The RPC parameter data, and the IN and OUT arguments to the remote service, are treated separately from the meta data. Every effort will be made to use the parameter data, without additional copies, as it is provided by the standard ‘C’ compiler of each target platform. This contrasts with the **mach_msg** approach which requires the parameter data to be copied to the message body. It is possible to send either a message or an RPC to the same port. For a more detailed description of this functionality, see the included paper [4].

In order to better present the changes to the API, it is important to briefly define a number of terms:

- **Thread** – Normally an execution stack and processor state. In MK, and other systems, a thread is also a schedulable object that contains scheduling parameters and possibly scheduling policy information.
- **Activation** – An MK thread that is currently associated with a shuttle.
- **Unbound Activation** – An activation that is not currently associated with a shuttle.
- **Shuttle** – This contains the scheduling policy and parameters, as well as CPU usage and a pointer to the current thread. The shuttle is the part of the thread/shuttle system that can migrate, while the thread part remains with the task in which it was created.
- **RPC chain** – Active threads (i.e. activations) in different tasks linked by RPC calls.
- **Alert** – A software generated exception that affects the thread at which the shuttle is currently pointing.
- **Signature** – An array of descriptors that describe the kernel visible arguments (e.g. ports) pushed on the stack for an RPC call.
- **Subsystem** – A data structure registered by an RPC target task to allow the kernel to marshall and unmarshall arguments and directly invoke task work functions.

1.6 CORDS

CORDS (Communication Objects for Real-time Dependable Systems) is a configurable, object-oriented Communication Subsystem derived from the University of Arizona’s x-kernel. CORDS has a number of desirable real-time features: virtual channels, fine control over system resources, privileged messaging and queue-free code paths. CORDS is execution domain independent: a protocol stack, generalized to a protocol graph, can be extended across one or more protection domain boundaries (i.e., the kernel/user boundary) without changes to individual protocols. CORDS’ dependencies on the host OS are all resolved within a limited portion of the CORDS framework; thus, it is conceivable to rehost CORDS to different OSs.

Many protocols have been implemented and are available in CORDS, including the most popular Internet protocols (i.e., TCP, UDP, IP, BOOTP, etc.), non-standard protocols developed at University of Arizona (i.e., CHAN, BLAST, etc.), and protocols developed at the RI for distributed systems (i.e., KKT, reliable and ordered group communication, and time synchronization protocols).

In MK7.3, the above mentioned protocols and/or new, user-provided protocols can be combined to build arbitrary CORDS protocol graphs.A set of built-in MK7.3 kernel configurations — namely {DEBUG, FAST, PRODUCTION}+NORMA+net — already include a CORDS protocol graph which complements DIPC with the communication fabric required by Ethernet-based NORMA clusters.

Two additional protocols were added to the CORDS environment in MK7.2 to support the use of Digital Equipment Corporation's DEFPA FDDI module for the PCI bus:

- fddi

  This is a hardware independent protocol that provides an interface between the actual FDDI drivers and the rest of CORDS. There should be a separate instantiation of the FDDI protocol for each driver protocol.
- xkfta

  MK in-kernel driver for the DEFPA module with microkernel and CORDS interfaces. Each instantiation is associated with a single FDDI device. XKFTA requires no lower protocol. The last digit of the instance name must correspond to the unit device number for a particular device. For example:

  name=xkfta/fta0

  for device 0.

1.6.1 CORDS Evolution

CORDS has grown along both the framework and protocol dimensions. The key evidence of such a growth can be found in the successful demonstration of Real-Time, Fault Tolerant behaviors, as presented in public events. One such demonstration is documented in:

http://www.opengroup.org/RI/OS/CORDS/ftsorter.htm

Additional information about CORDS can be found in:

http://www.opengroup.org/RI/PubProjPgs/CORDS.htm

The framework has been enhanced with additional real-time features: new tunable parameters have been exposed, for the user to operate more "knobs" such as scheduling attributes of threads and event granularity. The Path[28] interface[18] has been extended as well. In kernel space, the framework can now operate safely in the preemptive environment provided with the real-time enabled kernel configurations. In user space, the framework can optionally be built to use *rthreads*[5] and synchronizers[6] rather than traditional *cthreads*. Moreover, the framework can also be built to use the *pthreads* library. Although the *pthreads*-based configuration is not officially supported in MK 7.3, it will become the preferred configuration in the future. Please note that the choice of thread paradigm has no impact on protocol code.

In the realm of protocols, the original protocols were significantly enhanced and consolidated for the MK7.2 distribution. These protocols implement clock synchronization, IP multicast, node-liveness, and ordered group communication. In addition to Ethernet, CORDS protocols can now access an FDDI and a Myrinet network interface, both from kernel space and user space.

Within the CORDS space, two protocol stacks were implemented as servers in MK7.2: the LTS and GIPC servers.

1.6.2 LTS

LTS[12] provides a distributed clock with a guaranteed upper bound on the difference between the master clock and any of its client clocks. If the service becomes unable to maintain this guarantee, the client clock explicitly fails. These properties differentiate LTS from more traditional approaches to clock synchronization[19]; they make LTS an important building block of real-time distributed systems.

The LTS service is highly configurable. The server consists of a user-space management protocol and in-kernel protocols which perform the synchronization and maintain an MK kernel clock object using this data. Applications access the distributed time maintained by LTS using the standard MK clock object interface.

The synchronization and clock control algorithms are based on the work of F. Cristian et. al.[8].

1.6.3 GIPC

Our quest for predictable behaviors in fault isolation, fault management, and group communication[29] produced the first implementation in MK7.2: the Group IPC (GIPC) Developer's Kit. The kit includes several GIPC server configurations, a client library and a set of clients: tests and visual demonstrations. These clients are already instrumented to validate real-time properties of group interaction.

The GIPC server implements a single type of membership (live but not accurate), and a single type of ordered group communication and delivery semantics (FIFO atomic broadcast with safe delivery). In addition, the GIPC server exports a minimal process group interface, with one process client per physical node and only one group supported.

With the proper configuration of protocols and tunable parameters, the MK7.3 GIPC server Developer's Kit detects node failures and establishes a new, stable membership view in less than 100 milliseconds.[1]

1.7 Microkernel Performance Suite

Most MK performance measurements are accomplished by running typical UNIX programs on top of the server. For system developers, the performance figures obtained with such methods are not sufficient to analyze the performance of the microkernel as an individual component.

The Microkernel Performance Suite (MKPS) is a framework to take multiple measurements of each system service and compute the average performance and its standard deviation. It calibrates the number of iterations to get a significant measure (it has to perform enough iterations to get a low variation in the measure, but not too many to make sure that the results are available after a reasonable time), and automatically invokes the tests selected by the user. It also provides some tools to detect resource leaks that might affect the results. The set of microkernel services measured include IPC, VM, exception handling, scheduling, port, task and thread manipulation, EMMI, and device services. Since there are no kernel API modifications for this feature, see [1] for a complete description.

1.8 Event Trace Analysis Package

The Event Trace Analysis Package (ETAP) is an MK tracing facility, designed to provide users with a robust set of kernel level facilities to study system behavior. ETAP supports both kernel internal and application level event tracing. Kernel event tracing provides system programmers with a powerful tool for monitoring kernel internal activity. Application level trace support provides application programmers with a consistent, native to the microkernel, model for application specific event tracing. Together, the two levels of tracing enable users to better understand the interaction between their applications and the MK operating system. (See [7].)

1. on a private, isolated 10Mbit/second Ethernet.

1.9 IP Multicast[1]

In the MK7.2 release, a new IP Multicast feature was added with full compliance with the existing standards. For a more complete description of this standard protocol see [11].

IP multicasting is the transmission of an IP datagram to a "host group", a set of zero or more hosts identified by a single IP destination address. A multicast datagram is delivered to all members of its destination host group with the same "best-efforts" reliability as regular unicast IP datagrams, i.e., the datagram is not guaranteed to arrive intact at all members of the destination group or in the same order relative to other datagrams.

The membership of a host group is dynamic; that is, hosts may join and leave groups at any time. There is no restriction on the location or number of members in a host group. A host may be a member of more than one group at a time. A host need not be a member of a group to send datagrams to it.

A host group may be permanent or transient. A permanent group has a well-known, administratively assigned IP address. It is the address, not the membership of the group, that is permanent; at any time a permanent group may have any number of members, even zero. Those IP multicast

1. Copyright © 8.2 (Berkeley) 3/21/94

addresses that are not reserved for permanent groups are available for dynamic assignment to transient groups which exist only as long as they have members.

In general, a host cannot assume that datagrams sent to any host group address will reach only the intended hosts, or that datagrams received as a member of a transient host group are intended for the recipient. Misdelivery must be detected at a level above IP, using higher-level identifiers or authentication tokens. Information transmitted to a host group address should be encrypted or governed by administrative routing controls if the sender is concerned about unwanted listeners.

1.10 New Kernel Device Drivers

MK7.3 includes support for the NCR/Symbios PCI adapter 53C810; the DEC DEFPA-AA PCI FDDI board; the DEC DE-450 Etherworks 10 PCI card; the Diamond Stealth PCI 64-bit graphics card; the Myrinet PCI network board, SoundBlaster compatible audio cards, and a data acquisition ADC/DAC interface board.

Support for the DDI 8900 series relay cards has been added to MK. Set status and get status routines exist for setting or checking the state of individual relays. The “cio” device must be in the configuration to make use of this driver.

A device driver for the Quatech DAQ-12 12-bit data acquisition card has been added to MK. Set status and get status routines are provided for controlling the A/D and D/A converters, along with other hardware on the card. WARNING: Portions of this driver have not been fully debugged, use at your own risk!

Note: The DDI 8900 and the Quatech DAQ-12 are both used to support the ball-sorter demonstration (see the man page for *balldemo*(8)).

1.11 Reference Papers

[1] Barbou des Places, François, Philippe Bernadat, “OSF Microkernel Benchmark Suite”, OSF Research Institute, Cambridge, MA, July, 1994.

[2] Black, David, Randy Smith, “Fast Lightweight IPC: Architecture and Design”, OSF Research Institute, Cambridge, MA, May 1995.

[3] Bryant, Bill, “Design of AD 2, a Distributed UNIX Operating System”, draft document, OSF Research Institute, Cambridge, MA, November 1995.

[4] Burke, Edward, Michael Condict, David Mitchell, Franklin Reynolds, Peter Watkins, William Willcox, “RPC Design for Real-Time Mach”, OSF Research Institute, Cambridge, MA, April, 1994.

[5] CaraDonna, Joseph, “OSF–RI MK6.1 Real-Time Threads Library: Rthreads Interface Guide”, OSF Research Institute, Cambridge, MA, *DRAFT*: September 8, 1994.

[6] CaraDonna, Joseph, Robert Haydt, Franklin Reynolds, “New Synchronization Services for OSF Mach: Synchronizers”, OS Collected Papers vol. 3, OSF Research Institute, Cambridge, MA, April 12, 1994.

[7] Caradonna, Joseph, “The Event Trace Analysis Package: ETAP Users Guide”, OSF Research Institute, Cambridge, MA, February, 1995.

[8] Christian, Flavio, “Probabilistic Clock Synchronization”, Distributed Computing 3:146-158, 1989.

[9] Clark, R.K., E.D. Jensen, and F.D. Reynolds, “An Architectural Overview of the Alpha Real-Time Distributed Kernel,” *Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pp. 127-146, Seattle, WA, April, 1992.

[10] Condict, Michael, Don Bolinger, Éamonn McManus, David Mitchell, Steve Lewontin, “Microkernel Modularity With Integrated Kernel Performance”, OSF Research Institute, Cambridge, MA, April, 1994.

[11] Deering, S. “Host extensions for IP multicasting”. 1989 August; Internet RFC 1112, 17 p. (Obsoletes RFC 988, RFC 1054)

[12] Feeney, Laura, “LTS: Algorithms and Implementation”, OSF Research Institute Technical Report, in preparation.

[13] Haydt, Robert, Joseph CaraDonna, Franklin Reynolds, “Mach Scheduling Framework”, OSF Research Institute, Cambridge, MA, April, 1994.

[14] Jensen, E.D. and J.D. Northcutt, "Alpha: An Open Operating System for Mission-Critical Real-Time Distributed Systems—An Overview," *Proceedings of the 1989 Workshop on Operating Systems for Mission-Critical Computing,* ACM Press, 1990.

[15] Langerman, Alan, David Black, Michelle Dominijanni, Steve Sears, Randall Dean, Dejan Milojicic, "NORMA IPC Version Two: Architecture and Design", OSF Research Institute, Cambridge, MA, April, 1994.

[16] Locke, C.D., "Best-Effort Decision Making for Real-Time Scheduling," Ph.D. Thesis, Carnegie Mellon University, Pittsburgh, PA, May, 1986.

[17] LoVerso, John, Paul Roy, "The Network Architecture of OSF/1 AD Version 2", OSF Research Institute, Cambridge, MA, February, 1994.

[18] Menze, Edwin, Franco Travostino, "The x-kernel-based Communication Subsystem for OSF Mach: Changes to the x-kernel 3.2 API", Version 1.0, May 1995, OSF Technical Report. (Also available as: *ftp://riftp.osf.org/pub/real_time_papers/xk_prog_man.ps*) .

[19] Mills, David, "Network Time Protocol (Version 3) Specification, Implementation and Analysis." Network Working Group RFC 1305. March, 1992.

[20] Murphy, Daniel, "Timekeeping Enhancements in OSF/MK", The Open Group Research Institute, Cambridge, MA, June, 1997.

[21] Northcutt, J.D., R.K. Clark, "The Alpha Operating System: Programming Model," Alpha Project Technical Report 88021, February 1988.

[22] Noveck, David, "Interposition Mechanism for the OSF/1 AD Version 2 File System", OSF Research Institute, Cambridge, MA, July, 1994.

[23] Noveck, David, "Token Module for OSF/1 AD Version 2", OSF Research Institute, Cambridge, MA, July, 1994.

[24] Patience, Simon, Faramarz Rabii, “The Design of the Process Management Component of OSF/1 AD Version 2”, OSF Research Institute, April, 1994.

[25] Rabii, Faramarz, Simon Patience, “The Process Management Architecture of AD Version 2”, OSF Research Institute, Cambridge, MA, October, 1993.

[26] Roy, Paul, David Noveck, Durriya Netterwala, “The File System Architecture of OSF/1 AD Version 2”, OSF Research Institute, Cambridge, MA, August, 1993.

[27] Sears, Steve, David Black, Randall Dean, Michelle Dominijanni, Alan Langerman, “Kernel to Kernel Transport Interface for the Mach Kernel”, OSF Research Institute, Cambridge, MA, April, 1994.

[28] Travostino, Franco, Ed Menze and Franklin Reynolds, ‘‘Paths: Programming with System Resources in Support of Real-Time Distributed Applications”, Proceedings of the 1996 IEEE Workshop on Object-Oriented Real-Time Dependable Systems, Laguna Beach, CA, February 1996.

[29] Travostino, Franco, “Elementary Groupware For Fault Management”, OS Collected Papers vol. 4, OSF Research Institute, Cambridge, MA, October 1995.

[30] —, “ISO/IEC 9945-1: Second Edition 1996-07-12 (ANSI/IEEE Std 1003.1), Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language],” Institute of Electrical and Electronics Engineers, New York, NY, 1996.