ManRiX, a Microkernel Based OS Design

Manish Regmi
BEIT 8th semester NCIT
regmi.manish@gmail.com

Rajesh Bikram R.C
BEIT 8th semester NCIT
rajesh.rc@gmail.com

ABSTRACT

The microkernel architecture abstracts lower-level OS facilities, implementing them in kernel space, and moves higher-level facilities to processes in user space. The kernel is tiny and contains only those services, which must reside on kernel. All other Services, which are not required in Kernel, reside on User Space. ManRiX is a microkernel based OS which aims for POSIX compliance. POSIX is an IEEE 1003 standard created to standardize Operating Systems. So, it is possible to run POSIX compliant applications without any modification. It is an open source project, which is written from scratch. ManRiX is a Multithreaded Kernel. A process can have many Threads and process is a passive entity. Kernel pre-emption feature reduces latency and makes it suitable for Real time Systems. Virtual Memory Management is based on VM manager of MACH microkernel. Interrupt handling is done in User space. Idle Hooking feature make use of idle CPU. SMP capability enables it to run on Multiprocessor Systems. IPC with Synchronous Message Passing with unique techniques make message passing faster. File Managers, Device Drivers, Console Managers reside on user space. A network manager deals with Network hardware and software. The High Availability manager makes the system able to remain up and running without interruption for extended periods of time and makes the system highly fault tolerant.
It contains its own Standard C Library and is aiming for ANSI, POSIX, and Single UNIX Specification compatibility. So, ManRiX is a next generation OS suitable for small embedded systems to Large Multiprocessor systems.

1. INTRODUCTION

ManRiX is a microkernel based UNIX like operating system. It has a tiny kernel with the few services in kernel space (i.e. Process and thread management, signals and timers, IPC, virtual memory etc) and the other services (i.e. device Driver, file system manager, console manager, bus manager etc) in user space. ManRiX supports POSIX interface. The IEEE has developed POSIX standard. This standard defines operating system interfaces and not the implementation. ManRiX works on the concept of client and server based architecture. The focus of this paper is to give overview how the ManRiX is designed.

This project was started with a view of creating Learning Operating System. Later we decided to make it as a full-fledged open source Operating System. The main aim of this project is to create a microkernel based operating system by the involvement of students and Professionals. This project has also the aim of POSIX compliance. Login to http://manrix.sf.net for further details.

2. ManRiX DESIGN

The following figure portrays ManRiX microkernel:

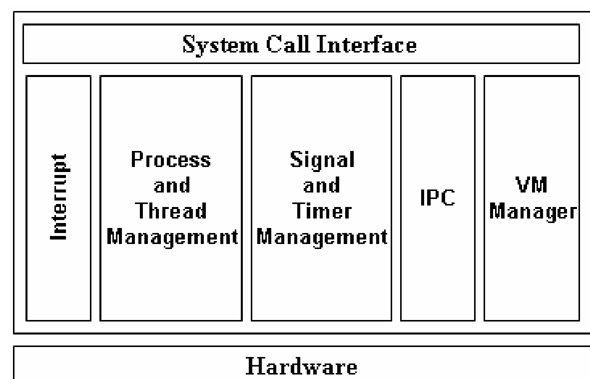

Fig 1: ManRiX Microkernel

Unlike other operating systems, there are very few sets of system calls, which is used to access microkernel services.

The some of the services provided in kernel mode can be listed as

- Process and Thread management
- Memory management
- Inter Process Management
- IRQ handling
- SMP and Kernel pre-emption
- Scheduler
- Idle hooking

The some of the services provided in user mode can be listed as:

- C- Library (POSIX, ANSI support)
- File system manager
- Console manager
- Device driver (i.e. Bus manager, ATA manager, floppy etc)
- Network manager
- High availability manager

The Design of ManRiX can be describes by the figure below

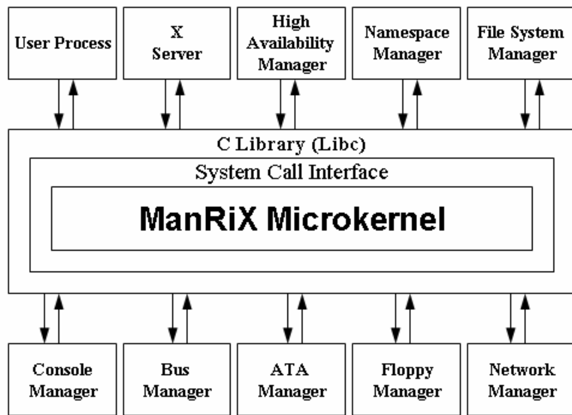

Fig 2: ManRiX Architecture

2.1 Process and thread management

Processes and Threads are the two basic concepts of program. In ManRiX microkernel, threads are the fundamental units of execution. Processes are inactive entity. Process just encapsulates the threads and address space. Threads are scheduled independently no matter to which process they belong. Each thread has two set of Stack one for running in Kernel Mode called Kernel Stack and for User stack for user mode. Each process contains a unique Process ID called pid and each thread contains unique thread ID called tid. The pid and tid are not unique to each other. The Users can Create Process by using POSIX fork and exec functions and Threads by using pthreads functions.

2.2 Memory Management

The ManRiX Memory Manager is based in Mach's VM design. The VM manager is divided into two parts, the machine dependent part called that *Pmap* and the machine independent part called the *vmmap.* This technique makes the portability easier to new architectures. MACH's original VM system was designed on an object oriented manner. Its basic abstractions are represented by objects that are accessed by a well defined interface.

The highest level object is vm_map. A vmmap describes the virtual address space of a process or the kernel. It contains a list of valid mappings in the virtual address space and those mapping's attributes. It holds a doubly linked list of vm_map_entries and a hint pointing to the last entry that resolved a page fault. Each vm_map_entry describes a contiguous region of virtual memory that has some protections.

A vm_object represents a memory object. The memory object can be a set of physical pages or a memory mapped files. A vm_object describes a file, a zero-fill memory area, or a device that can be mapped into a virtual address space. The vm object contains a list of vm page structures that contain data from that object. The vm_page describes a page of physical memory4. When the system is booted a vm page structure is allocated for each page of physical memory that can be used by the VM system.

The VM manager supports read/write sharing of memory and copy on write sharing of memory. The ManRiX VM manager uses a slab allocator (conceived by Solaris) to allocate kernel memory. The following figure depicts the ManRiX VM manager:

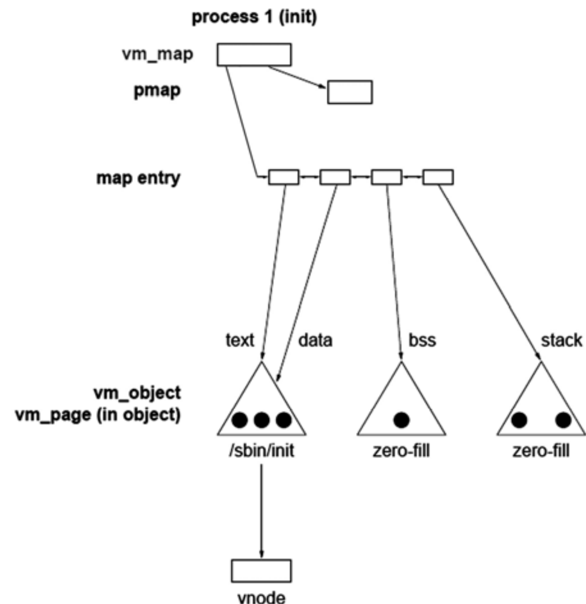

Fig 3: ManRiX VM Manager

2.3 Signals and Timers

Signals provide a mechanism for notifying processes of system events. They also function as a primitive mechanism for communication and synchronization between user processes. ManRiX supports POSIX signals. It has also a limited support of POSIX real-time signals. ManRiX supports 32 different signals. 26 of them are standard POSIX signals and 5 are Real-time Signals. In ManRiX signal handlers are per process objects. All threads in a process share same signals handlers but have their own signal mask.

Timer notifies a thread when the time as measured by a particular clock has reached or passed a specified value, or when a specified amount of time has passed ManRiX support limited POSIX timers.

2.4 Interprocess Communication

Inter Process Communication is Heart of any Microkernel Based System. In ManRiX Microkernel, Message passing is used as the IPC.
We are using Synchronous message passing. The message is sent directly sent to another thread using tid. There are basically three types of Messages in ManRiX. We have basically three types of Messages. They are SHORT, LONG and MAP.
The SHORT messages are passed Only in CPU Registers. This is used in IPC where messages to send

and receive are very short and fit in general purpose resisters. This type of message passing is extremely fast but can accommodate short messages (32 bytes). In LONG messages, the Messages are copied from one Address Space to another. This is not possible directly so we are first copying to Kernel and then to another address space. Usually, the Messages shorter than 256 Bytes are Long Transferred. It is quite slower than SHORT messages. And In the MAP messaging, the messages are NOT actually transferred but the page tables are mapped. Page Table mapping is not efficient for short messages.

2.5 Interrupt Request (IRQ) Handling

ManRiX does not handle devices in kernel space. They can be implemented in user space using a set of interfaces. The ManRiX microkernel sends appropriate signals to the corresponding threads on CPU exceptions.

The ManRiX microkernel doesn't handle interrupt request in kernel space. It only acts as an IRQ redirector. For IRQ handling the User thread can register a handler for an IRQ. On the arrival of IRQ event, the Kernel finds the handler(s) for that IRQ and runs the handler. The IRQ is handled exactly as Signal Handler.

2.6 Symmetric Multiprocessing (SMP)

ManRiX microkernel has a multiprocessor support. For now, it supports Intel Multi Processors (MP) systems only. The number of Processors supported till now is eight. In SMP systems, Kernel Synchronization is a major issue. We overcome this issue using spin locks. The specialty of ManRiX microkernel is that the Interrupts are always enabled except in few situations like Interrupt redirection code. So, Spin locks don't disable interrupts and thus the system is highly responsive. In ManRiX the Kernel code path is very short because the lengthy file system and device handler does not exist in microkernel. So, the mutex locks are not used.

2.7 Kernel Preemption

ManRiX microkernel is a fully preemptible. It means that it is able to preempt the thread even if it is running in kernel mode. So, a highest priority thread starts running as soon as it becomes eligible to run. The Kernel preemption feature highly increases the responsiveness of the system and decreases the latencies. So, due to this characteristic ManRiX can also be used in real time systems.

Preemption can be disabled at critical section. The locks automatically disable kernel preemption and re enables them when lock is released. When the preemption is disabled, the interrupts are still enabled but the timer interrupt does not switch threads.

2.8 Scheduler

ManRiX supports POSIX style scheduler. There are three scheduling policies, SCHED_FIFO (First in-first out (FIFO) scheduling policy), SCHED_RR (Round robin scheduling policy) and SCHED_OTHER (Another scheduling policy). SCHED_FIFO and SCHED_RR are used for real time threads and SCHED_OTHER is used for time sharing threads. The system threads use SCHED_OTHER but have a higher priority than normal threads. There are 128 priorities where numerically lower value means low priority and vice versa. The SCHED_FIFO and SCHED_RR policies have priorities ranging from 88 to 119. They have static priorities. The SCHED_OTHER priorities range from 4 to 87. The range from 0 to 3 is for idle threads. The system threads (i.e. file servers, consoles etc) use the policy SCHED_OTHER and have priorities from 46 to 87. The priorities for system threads are static. There are also special threads which handle interrupts called interrupt threads. They have the highest priority ranging from 120 to 127. The normal threads have priorities.

The ManRiX scheduler is capable of distinguishing between interactive and non interactive threads. The interactive threads are I/O bound i.e. they spend most of their time waiting for an I/O event. In the other hand, the non interactive threads are CPU bound i.e. they spend most of their time using CPU. So, the priorities of interactive threads should be higher than that of non interactive threads. The scheduler increases the priority of interactive threads when they go to sleep and decreases the priority of the non interactive threads.

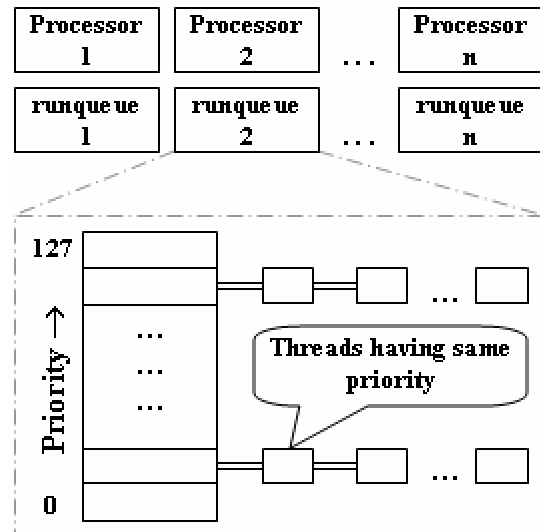

Fig 4: ManRiX Scheduler

Since ManRiX supports SMP systems, the scheduler should be able to schedule threads in all existing processors. In ManRiX, each processor has its own run queue. Since the run queue is a per processor object, there is no need to lock the run queue. One of the special features of ManRiX is automatic balancing

of CPU load. There is no need to migrate the thread from one processor to another. Each run queue contains a counter of total no of threads in running in that processor. There is a hint pointing to the run queue having least no of threads. Now, during enqueuing, a thread is enqueued to the run queue pointed by the hint. This hint is updated either by idle thread or the timer interrupt code on every 500ms on loaded systems.

2.9 Idle Hooking

The ManRiX microkernel is capable of making effective use of idle CPU. The user can hook a function to run when the processors are idle. Some examples are writing dirty disk buffers, power management etc.

3. USER SPACE SERVERS

ManRiX Microkernel contains just few basic components of an Operating System. Those components that are not required inside the microkernel are implemented as user process. ManRiX has its own set of Standard C library conforming to ANSI C and POSIX. The system services and drivers can easily use the standard C library but the Driver writers should be cautious to see the chicken or egg problem does not arise. The system services are based on Client Server architecture. All the applications use message passing to communicate with the servers. All servers are multithreaded. They implement a pool of threads called worker threads which do the actual work on request arrival. The listener thread listens for the request and passes it to the worker threads.

A special technique is used reduce the number of message passing for I/O managers. Using this technique, after an open call, the applications can directly send read, write, ioctl etc to I/O manager. For an example, if an application opens a file "/dev/ttyS0" (serial port), it can directly call the serial driver without going through File System. It is achieved by using a multiplexed file descriptor where first 16 bit is the file descriptor and the other 16 bit is the thread id of the thread managing that I/O device.
For Non I/O managers, there is a separate namespace server that resolve server name to its thread id.
Some of the important servers are:

3.1 File System Manager

ManRiX is conforming to POSIX so the file system is based on UNIX. File System manager uses VNODE/VFS architecture. It makes possible to support multiple file systems (like fat, ext2/3, ufs, ntfs etc.).

3.2 Console Manager

ManRiX has a POSIX terminal support. It can support more than one terminal. The Console manager manages the Keyboard Driver and the Text console driver. So, when an application do printf(), the message is sent to the console manager which displays the output to the terminal associated with that process.

3.3 Bus Manager

ManRiX manages almost all devices in User Space. The Bus manager manages the various buses like PCI, USB etc. So, if any other driver manager needs to read Configuration space of its device (connected to PCI BUS), it requests the BUS manager through IPC.

3.4 ATA Manager

The ATA manager manages ATA/ATAPI devices. They are popularly known as Integrated Device Electronics (IDE). The file manager requests the ATA manager if it needs to read/write a disk block.

3.5 Floppy Manager

The Floppy manager manages Floppy disks.

3.6 Miscellaneous I/O manager

The Miscellaneous manager manages devices that do not have their own server. For now only CPU driver is implemented in misciomanager.

3.7 High Availability Manager

The term *High Availability* (HA) System is a system, which remains up, and running without interruption for extended periods of time. The HA manager is a smart watchdog that can perform recovery whenever system services or processes fail. This subsystem has not been implemented yet.

3.8 Network Manager

The Network manager manages all the network related work. The subsystem contains BSD socket interface, Network Protocol stacks and Ethernet drivers. This subsystem has not been implemented yet.

4. HOW ManRiX BOOTS?

Initialization of ManRiX microkernel is highly architecture dependent. We give a brief overview how ManRiX boots in i386 systems. ManRiX can only be loaded by Grub (Boot Loader) or any multi boot complaint boot loaders. All executables are in UNIX ELF format. Since, the microkernel is not capable of loading whole Operating System; some extra modules need to be loaded by the boot loader. Boot loader typically loads Kernel, Console Manager, ATA and Floppy Manager, Bus Manager, File Manager and init. The boot loader then passes the control to the entry point of the kernel which is _start and it jumps to kernel_entry. The initializer part

- disables interrupt
- sets up the PAGE sized STACK (statically allocated) and switches to new stack
- calls multiboot_parse which checks if we are correctly loaded by a multiboot complaint boot loader

- calls the main C function ManRiX_main (architecture independent)
- call arch_setup; which does architecture specific setup. It
    - initializes paging
    - setup descriptors
    - initializes traps
- initialize VM Manager
- initialize Process and thread subsystem
- initialize Scheduler
- initialize IRQ
- initialize Timer
- initialize SMP
    - if not SMP system, do nothing and return
    - initialize Application Processors (AP)
    - create idle threads
- initialize Module
    - get the module information for each module
    - for each module,
        - check the validity of module, is it ELF? Is it compiled for target processor? Is it executable? Etc
        - create a new process using do_spawn
        - change its process ID and thread ID because some important modules/ servers have static process ID and thread ID
        - create the memory context
            - create map entries for image and stack
            - create vm_page_t structure for memory occupied by module image because it is skipped by memory manager initializer
- enable interrupts and become the idle thread for boot strap processor (BSP)

5. WHY MICROKERNEL?

Though there are some pitfalls of using Microkernel Design, there are some benefits too. First microkernel design makes ManRiX modular and scalability. Another important benefit is that the fault is one driver or subsystem does not bring the whole system down. The subsystems and drivers can be restarted at runtime. Also the code grows slower than Monolithic kernels.

6. WHY POSIX?

We are trying to give ManRiX POSIX interface. There are a lot of benefits of this and no drawbacks. First of all the end users and software developers does not need to learn ManRiX specific things. They can simply use the POSIX interface. This also cuts down Training Cost and time. Secondly, the UNIX programs will be source compatible with ManRiX. The program written for UNIX or LINUX will compile and run in ManRiX without any change and sometimes minor change. So, there is no need for us to create separate applications for ManRiX. We can easily port GNU application.

7. FUTURE PLAN

- porting open source applications
- make it usable by non technical users
- support X windows and different window managers
- port to different architectures like PPC, X86_64, Motorola, ARM etc
- make it real time capable
- create a robust network manager

8. CONCLUSION

ManRiX a microkernel based OS is effort of two students and is in nascent stage. As a designer, we will say it is not that easy but once started having arduous and great perseverance any thing becomes easy. Developing code in kernel doesn't require genius, magic, or a bushy Unix-hacker beard. The kernel, although having some interesting rules of its own, is not much different from any other big software project. There is much to learn as with any large project, but there is not too much more sacred or confusing about the kernel than any other software endeavor.

9. ACKNOWLEDGEMENT

We love to thank Ananta Bhadra Lamichhane and Rupesh Bhochhibhoya for their encouragement and constructive criticism.

10. REFERENCE

1. Unix Internals the new frontiers; *Uresh Vahalia*
2. The Desigm of Unix Operating System; *M.J.Bach*
3. The Intel Processors; *Barry B. Brey*
4. Modern Operating System; *Galvin Silverscatz*
5. Linux Kernel Development*, Robert Love*
6. Solaris Internals: Core Kernel Architecture, *Jim Mauro* and *Richard MC Dougall*
7. Advanced UNIX Programming, *Richard Stevens*
8. Single UNIX Specification Version 3
9. The Operating System Resources : http://www.nondot.org/~sabre/os/articles
10. Intel processors manuals : http://developer.intel.com : http://www.x86.org
11. Linux Kernel Sources : http://www.kernel.org
12. GNU Mach Sources : http://www.gnu.org/software/hurd/gnumach.html
13. mega-tokyo forum : http://www.mega-tokyo.com/forum
14. Usenet: alt.os.development