Minix over Linux: A User-space Multiserver Operating System

Pablo Pessolani, Oscar Jara

Departamento de Ingeniería en Sistemas de Información - Facultad Regional Santa Fe
Universidad Tecnológica Nacional - Santa Fe - Argentina

{ppessolani,oajara}@frsf.utn.edu.ar

Abstract. *Minix is an open-source multiserver operating system designed to be highly reliable, flexible, and secure. The kernel is small and is the only piece of software that runs in privileged-mode, on the other hand user processes, specialized servers and device drivers run as isolated processes in user-mode. System Calls use Interprocess Communications primitives to send messages requesting services from the servers, and to wait for response messages. The aim of the project described in this article is a user-space multiserver operating system (a modified Minix version) running on top of a middleware-based virtual machine with simulated hardware constructed from services provided by a host operating system (Linux).*

1. Introduction

Minix [1] is a complete, general-purpose, time-sharing, multitasking, microkernel multiserver Operating System (OS) developed from scratch by Andrew S. Tanenbaum broadly used in Computer Science degree courses.

Though it is copyrighted, the source has become widely available for universities for studying and research. Its main features are:

- *Microkernel based*: Provides process management and scheduling, basic memory management, IPC, interrupt processing and low level Input/Output (I/O) support.
- *Multilayer system*: Allows modular and clear implementation of new features.
- *Client/Server model*: All system services and device drivers are implemented as server processes with their own execution environment.
- *Message Transfer Interprocess Communications (IPC):* Used for process synchronization and data sharing.
- *Interrupt hiding*: Interrupts are converted into message transfers.

Minix 3 is a new open-source operating system [2] designed to be highly reliable, flexible, and secure. It is loosely based somewhat on previous versions of Minix, but is fundamentally different in many key ways. Minix 1 and 2 were intended as teaching tools; Minix 3 adds the new goal of being usable as a serious system for applications requiring high reliability.

Minix 3 kernel is very small (about 5000 lines of source code) and it is the only code that runs under kernel privilege level. User processes, system servers including device drivers are isolated one from another running with lower privileges (Figure 1). These features and other aspects greatly enhance system reliability [3]. This model can be characterized as a multiserver operating system [4].

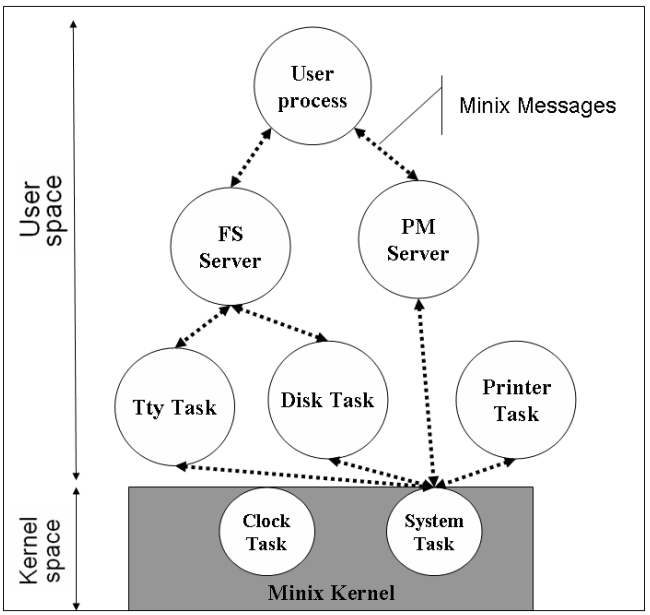

Figure 1. The Internal Structure of Minix 3

User applications make system calls using message transfers, packaging function arguments and the results in the same way as RPC does. Two servers handle requests from system calls, they are the Process Manager (PM) and the Filesystem Server (FS). FS and PM also make requests to device drivers processes using message transfers.

The Minix kernel is like a message broker where user processes, servers and tasks exchange messages through it. As all server processes and the majority of tasks run in user-mode, modifying them to run in user-mode of another OS sounds feasible.

Minix **o**ver **L**inux (MoL) could be considered as the set composed of system servers, device driver tasks, a virtual kernel and virtual devices; all of these running entirely in user-mode, achieving a Minix OS virtualization environment. MoL required the rewriting of the Minix kernel code, the IPC primitives, the Clock task, the System task, FS server, PM server, system libraries and the addition of a software layer to virtualize hardware.

On MoL, user processes run contained inside a self-contained environment. They have no access to any host resources other than those explicitly provided by MoL.

MoL lends itself to a variety of applications, such as:

- Operating system safe development and debugging.
- Simultaneously run different versions of servers and task.
- Application isolation such as sandboxing and jailing for intrusion detection and controlled malware analysis.

- Custom auditing and logging.
- Fine grained access control.
- System components and application profiling.
- IPC tracing and system calls interception and redirection.
- Virtualized devices that emulate hardware that is not physically present.
- Distributed processing running user processes, virtual kernel, servers and tasks in different hosts OS.

This article presents an overview (limited by space constraints) of the design and implementation of MoL. It also describes the outline of the work that have to be done to make MoL a realiable OS running as a message oriented middleware. The current is a germinal version of MoL that does not support serious Minix programs. This experimental version is a proof of concept of a Multiserver OS running in user-space using OS-based virtualization. It must be clear that the proposed approach is not restricted to Minix as the guest OS and Linux as the host OS.

The rest of this article is organized as follows. Section 2 refers to related works. Section 3 describes the MoL architecture and its components. Section 4 presents conclusions and plans of future works.

2.Related Works

Running a guest OS in user-mode on top of a host OS is not a new idea. It has several concepts that share with emulation and virtualization, furthermore it can be seen as a paravirtualization approach where the host OS is the Virtual Machine Monitor (VMM).

OS-level virtualization [5, 6] is a virtualization method where the kernel of an OS allows multiple isolated user-space instances, instead of just one (see Figure 2). Such instances (often called containers, VEs, VPSs or jails) may look and feel like a real host from the point of view of its processes.

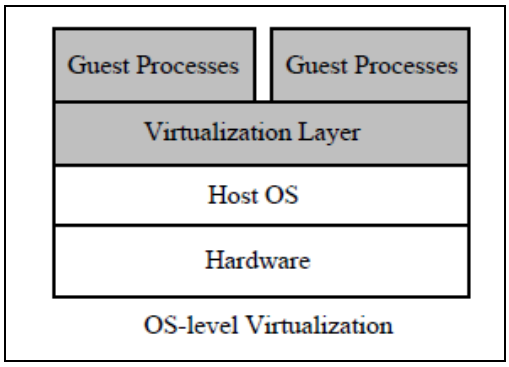

Figure 2: Virtual Operating System Approach (from [6]).

On Unix systems, this technology can be thought of as an advanced implementation of the standard ***chroot*** mechanism. In addition to isolation mechanisms, the kernel often provides resource management features to limit the impact of one container's activities on the other containers. Next sections present some related works about OS-level virtualization.

2.1. User-Mode Linux

A well-known project about running Linux in user-mode is User-Mode Linux (UML) [7] which does not involve a VMM or Hypervisor. User-mode Linux (UML) enables multiple virtual Linux systems (known as guests) to run as an application within a normal Linux system (known as the host). As each guest is just a normal application running as a process in user-space, this approach provides the user with a way of running multiple virtual Linux machines on a single piece of hardware.

2.2. Linux-Vserver

Linux-VServer [8] is a Virtual Private Server (VPS) implementation that was created by adding OS -level virtualization capabilities to the Linux kernel.

Linux-VServer is a jail mechanism in that it can be used to securely partition resources on a computer system in such a way that processes cannot mount a denial-of-service attack on anything outside their partition. Each partition is called a security context, and the virtualized system within it is the VPS.

2.3. OpenVZ (Open VirtualiZation)

OpenVZ [9] is an OS-level virtualization technology based on Linux. OpenVZ allows a physical server to run multiple isolated OS instances, known as containers, VPSs, or Virtual Environments (VEs). It is similar to FreeBSD Jails and Solaris Zones.

2.4. LXC (Linux Containers)

The LXC [10] package combines these Linux kernel mechanisms to provide a user-space container object, a lightweight virtual system with full resource isolation and resource control for an application or a system.

LXC builds up from ***chroot*** to implement complete virtual systems, adding resource management and isolation mechanisms to Linux's existing process management infrastructure.

2.5. FreeBSD Jails

The FreeBSD jail [11] mechanism is an implementation of OS-level virtualization that allows administrators to partition a FreeBSD-based computer system into several independent mini-systems called jails.

The need for the FreeBSD jails came from service providers' desire to establish a clean, clear-cut separation between their services and their customers, mainly for security and ease of administration. Instead of adding a new layer of fine-grained configuration options, the solution adopted was to compartmentalize the system, both its files and its resources, in such a way that only the right person(s) are allowed access to the right compartment(s).

2.6. Feather-weight Virtual Machine (FVM)

The key idea behind FVM [6] is access redirection and copy-on-write, which allow each VM to read from the base environment but write into the VM's private workspace. As a result, multiple VMs can physically share resources of the host environment but can

never change resources visible to other VMs. Windows IPC interfaces, and confine them in the scope of each individual VM. This allows the FVM layer to achieve strong isolation among different VMs.

3.MoL Architecture and Components

A main technical challenge with OS-level virtualization is how to achieve strong isolation among VMs that share a common base OS. This section presents the major MoL components and the implementation of the Minix OS on a Linux platform.

On several OSs (as Linux) a user process interacts with other entities through the following abstractions (Figure 3):

- *System Calls:* The process can operate on files, semaphores, queues, sockets, etc. requesting services to the OS through system calls (or through libraries).
- *Signals*: The operating system notifies the process that an event has occurred.
- *Shared Memory*: Shared memory allows two or more processes to share the same region of memory. Since a shared memory segment becomes part of a process's user-space memory, no kernel intervention is required for IPC.

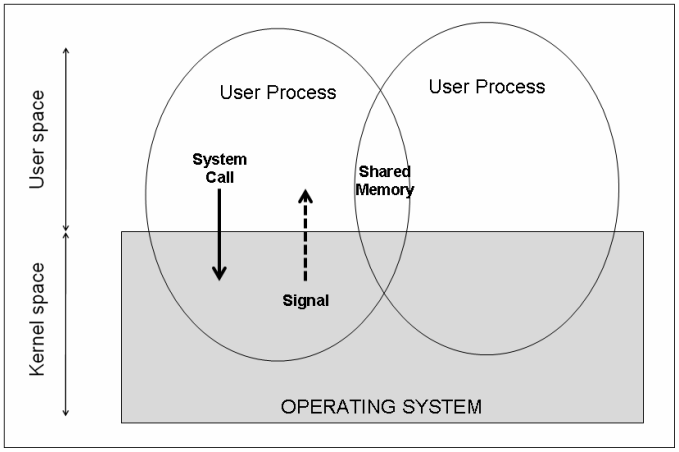

Figure 3: User Processes Interactions.

Until version 3.1.2 Minix did not support shared memory (at the time of this writing an experimental version is under development). Therefore, to emulate the execution environment for Minix processes is required the interception of system calls and to deal with signals. As this task requires modification of Minix libraries, Minix programs need to be linked with those libraries providing source code compatibility. Future versions will provide binary compatibility.

The MoL system is a set of Linux daemons that interchange Minix messages using some standard communication protocol (i.e. UDP) through a pseudo Minix kernel (***molkernel***) as it is shown in Figure 4 and explained in the following sections. The PM is emulated by a daemon named ***molpm***, the FS is emulated by another daemon named ***molfs***, and all Minix tasks have their MoL counterpart daemons.

To clarify the terminology used in this article, a Minix program running under MoL environment is called a "*Minix process*" (although, indeed, it is a Linux process).

A Minix process code runs natively on the processor without any sort of instruction emulation and has no access to host resources that were not explicitly provided by the MoL Virtual Machine.

Multiple instances of MoL (different instances of ***molkernel***, ***molfs***, ***molpm***, and MoL tasks) could be running on the same Linux Host OS, each of them as a virtual machine. On the other hand, it must be clear that MoL daemons do not need to run in the same Linux system, they could be scattered among different servers, acting as a primitive Distributed Operating System (DOS).

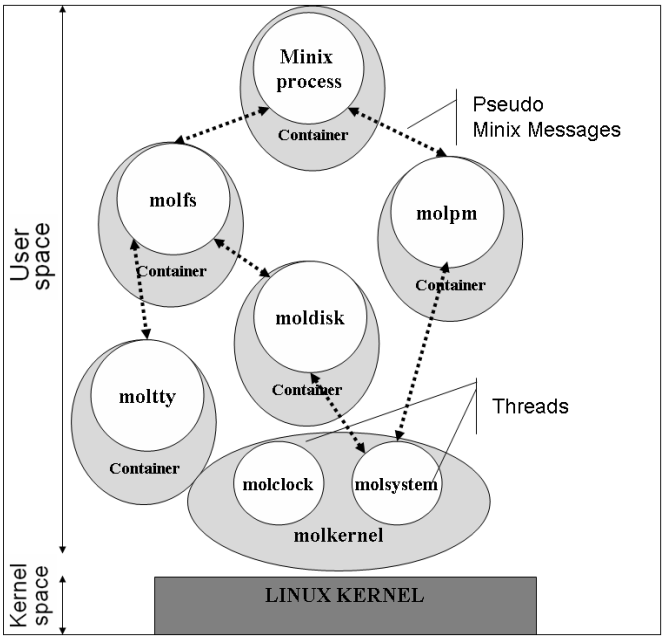

Figure 4: Minix Over Linux Architecture.

Two main design goals of MoL are:

- To be as independent of the host's architecture as possible to facilitate the migration to different platforms other than Linux.
- To use a minimum of the host's resources in order to facilitate live process migration on a cluster of machines.

3.1. Execution Environment of a Minix Process

To run a Minix program under Linux, MoL maps a Minix process to a Linux process and installs a container through a set of dynamic libraries that handle Minix system calls and traps Linux signals. The container intercepts system calls invoked by the Minix process and converts them into Minix messages encapsulated into MoL messages (explained in Section 3.2 and Section 3.3) using some of the communications protocols or IPC abstractions that Linux provides (explained in Section 3.4). The container also traps Linux signals addressed to the Minix process (explained in Section 3.5) and, depending on the kind of signal and the mask that the Minix process has set, calls the Minix handler, or ignores it, or informs the pseudo Minix kernel about the signal using MoL messages.

Process scheduling, memory management, exception handling, timer and interrupt management and alarm signaling are handle transparently by Linux through the process' container.

3.2. Minix System Calls Virtualization

Minix system calls are implemented using the *sendrec()* IPC primitive for message transfers (presented in Section 3.3). The arguments and results are packaged in the same way as Remote Procedure Call (RPC) does, as it is shown in the following source code:

```
int syscall(who, syscallnr, msgptr)
int who;                  /* destination server i.e. PM or FS */
int syscallnr;            /* System Call number */
message *msgptr;          /* the message has system call args */
{
          int status;
          msgptr->m_type = syscallnr; /* System Call number */
          status = sendrec(who, msgptr); /* Request&Reply */
          if (status != 0)
            {msgptr->m_type = status;}
          if (msgptr->m_type < 0)  {
           errno = -msgptr->m_type;
           return(-1);
          }
          return(msgptr->m_type);
}
```

In example, the POSIX system call *getpid()* is implemented sending a GETPID request to the PM and waiting for the reply from it, as it is shown in the following source code:

```
pid_t getpid()
{
          message m;
          return(syscall(PM, GETPID, &m));
}
```

Therefore, it is straightforward to convert the Minix message transfers IPC primitives (as *send*, *receive*, *sendrec* or *notify,* see Section 3.3) into another message transfer mechanism that can use pipes, sockets, message queues, RPC or MPI.

System calls invoked by Minix processes are, in fact, Linux system calls, but they are intercepted by the MoL container and converted into Minix IPC message transfers that are encapsulated in MoL messages as it is shown in Figure 5 and detailed in Section 3.4.

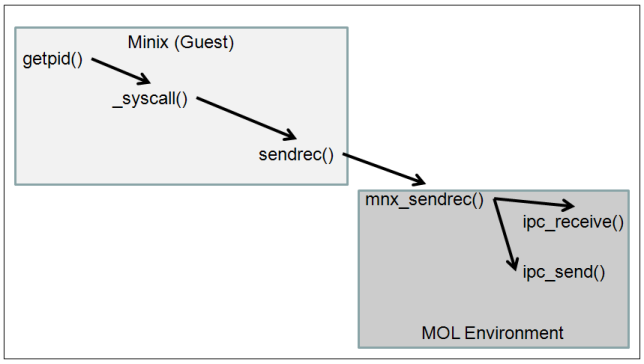

Figure 5: Path from a System Call to MoL IPC.

Minix system libraries need to be rebuilt to allow user programs to be compiled and linked with the support of the new mechanism of system calls.

The Linux system calls invoked by the Minix process are intercepted by a preloaded set of dynamic Linux libraries (using the Linux LD_PRELOAD environment variable) and replaced by their MoL counterparts.

In future versions that will support Minix binary compatibility, the system call interception will be implemented using Linux ***ptrace*** system call and a Linux module.

3.3. Minix IPC

A Minix message is a C language union of seven C language structures as it is shown in Figure 6. The field *m_source* is the sender's endpoint, *m_type* is the message type, and the other fields are used to carry different types of data (i.e. *m5_i1* means message type 5, integer field 1 and *m8_p2* means message type 8, pointer field 2).

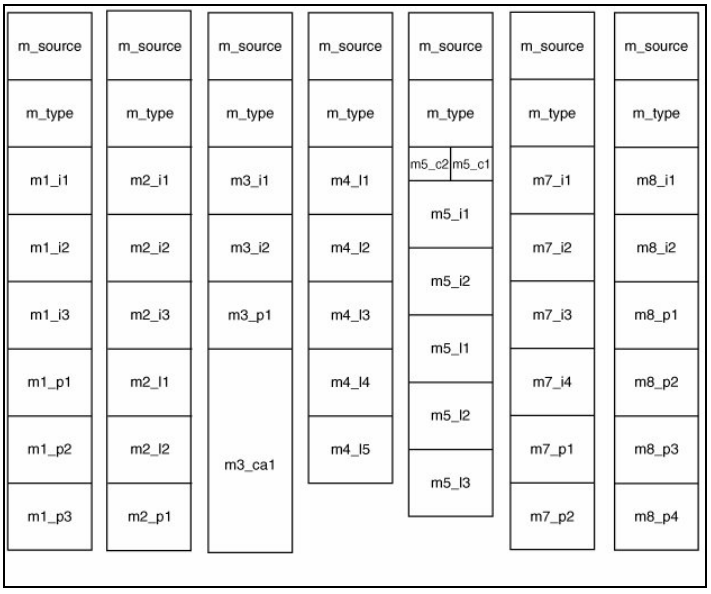

Figure 6: Minix Message Formats (from [1]).

A process *endpoint* uniquely identifies a single process with respect to Minix IPC. The reader should not confuse the *endpoint* with the PID of the process. The former is a number for system internal use related to the kernel process table slot, and the latter is a number that identifies Minix processes to be used as a parameter in system calls for processes management, such as adjusting the process's priority.

All processes in MINIX 3 can communicate using the following IPC primitives:

- *send*: to send a message to a process identified by its endpoint.
- *receive*: to receive a message from a process identified by its endpoint or from any process.
- *sendrec*: to send a request message and to receive a reply from a process identified by its endpoint.
- *notify*: a non blocking send of special message type.

In Minix those primitives are implemented into the kernel as CPU traps that change the processor from user-mode to kernel-mode. In MoL, they are message transfers from Minix processes to the MoL kernel.

3.4. MoL IPC

MoL IPC is a simple protocol that encapsulates Minix messages into MoL IPC messages. They are transferred using some internal communication abstraction as POSIX Message Queues, Named Pipes, Unix Sockets or a network protocol/interface as UDP/TCP sockets, RPC, MPI, etc. (the current version uses UDP sockets).

The MoL message format is shown in Figure 7. The header consists of the following fields:

- The operation/return code of a request/reply.
- The Sender's Linux PID.
- The Sender's message sequence number.
- The process number (explained in Section 3.6).
- The process endpoint.
- The destination endpoint of *send*/*sendrec*/*notify* request or the source endpoint of a *receive*/*sendrec* request.

Some of these fields are not really required (because they are fields of the Minix message), but they are included in the prototype version of MoL to facilitate debugging.

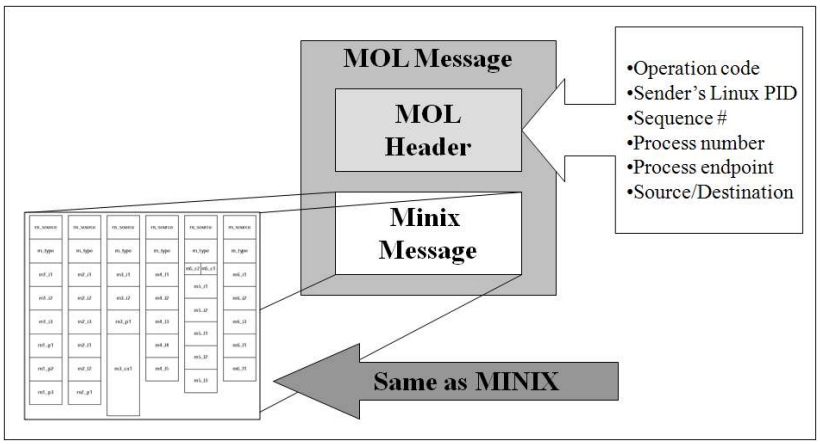

Figure 7: MoL Message Format.

The basic IPC operations that the MoL protocol supports for client processes are:

- *ipc_open*: Initialize the underlying protocol (UDP sockets, MPI, etc.).
- *ipc_send*: Sends a MoL message to ***molkernel*** (explained in Section 3.6)
- *ipc_rcv*: Waits for a MoL message from ***molkernel***.
- *ipc_close*: Frees the resources associated to the underlying protocol.

MoL messages not only transport Minix messages, but they are also used for communication between the ***molkernel*** and the containers of MINIX processes.

The current experimental version of MoL does not consider security issues nor performance optimizations. Therefore, it is likely that MoL header format, IPC operations numbers and arguments will change in future versions.

3.5. Signal Delivery

A signal is a notification to a process that an event has occurred. Signals are sometimes described as software interrupts.

The reader must distinguish signals delivered by the underlying Linux OS from those delivered by the emulated Minix OS. The MoL container installs Linux signal handlers to every Linux signal that could terminate the process by default (it is not possible to catch SIGKILL and SIGSTOP) and ignore those signals that are ignored by default. All Minix system calls related to signal handling (*sigaction*, *sigpending*, *sigprocmask*, *sigsuspend, longjmp, setjmp*) are intercepted by the MoL container.

When Linux delivers a signal to the Minix process (that would terminate it), the MoL container catches that signal and, if the Minix process has not installed a handler for it, the MoL handler for this signal terminates the MoL environment gracefully, notifying the ***molkernel*** of that event, closing communications and releasing any Linux resource that was acquired.

If the signal was produced by the Minix process itself and it had installed a handler for it (i.e. SIGFPE-Floating point exception), the MoL handler traps the Linux signal, and then executes the Minix installed handler. Once the installed handler was executed the MoL handler returns, and the execution flow continues under Linux control from the point where the signal was delivered.

3.6. The Pseudo Minix Kernel

The pseudo Minix kernel (***molkernel***) is a Linux daemon that manages all messages transfers between Minix processes checking for legal destinations and process privileges, copying messages from sender to receiver and changing Minix process virtual states.

The ***molkernel*** has its own process table that keeps information about the MoL protocol and the virtual state of each Minix processes. As Minix uses rendezvous for message transfers, MoL needs to keep track of the virtual state (Ready or Blocked) of Minix processes as the true Minix kernel does.

Unlike the Minix kernel, ***molkernel*** does not handle hardware interrupts, low level memory management nor process scheduling. Those tasks are performed by Linux (explained in Section 3.1).

3.7. The MoL System Task

In a conventional OS with a monolithic kernel, the term system call is used to refer to all calls for services provided by the kernel. However, in Minix terminology system calls are not requested directly to the kernel, they are requested to PM or FS. Server processes communicate with each other, with device drivers, and with the kernel through messages. The job of the Minix System task is to accept all the requests for special kernel services (called *kernel calls*) from the drivers and servers and carry them out [1].

Minix System task is a process that shares kernel address space (it is like a kernel thread).

MoL System task (***molsystem***) is a POSIX thread of ***molkernel*** that emulates Minix Sytem task. Some of the pseudo kernel calls that ***molsystem*** performs are about process management, time management, system information management and copying raw data among Minix processes (with the help of the containers), emulating a memory copy.

3.8. The MoL Clock Task

Timers are essential to the operation of any timesharing system, for example, they maintain the time of day. The Minix Clock task is driven by interrupts generated by a hardware device (i.e. the Programmable Interrupt Timer). The Minix Clock task is another process that shares kernel and System task address space.

MoL Clock task (***molclock***) it is not a process; it is a function integrated into ***molsystem*** that uses a Linux timer which delivers SIGALRM signals to it instead of real timer interrupts. Periodically, the variable *realtime* (which counts timer ticks since MoL start-up) is synchronized with the Linux system time to consider possible delays in the SIGALRM delivery. The frequency of SIGALRM signals can be less or equal of the Linux timer frequency.

3.9. MoL Virtual Devices

All devices accessible inside the virtual machine are themselves virtual. They are constructed from the appropriate abstractions provided by the host OS. MoL implements virtual device drivers tasks as Linux daemons (see ***moldisk*** and ***moltty*** on Figure 4) which use software abstractions of the host to virtualize hardware devices. For example, a MoL disk driver (***moldisk***) does not operate on a real hard disk device, it operates on a regular Linux file to virtualize a disk.

3.10. The MoL Process Manager

Minix servers provide services to Minix user processes. The PM carries out all the Minix system calls that involve starting or stopping process execution, such as *fork*, *exec*, and *exit,* memory management as *brk,* as well as system calls related to signals, such as *alarm* and *kill*, which can alter the execution state of a process. MoL implements a modified version of Minix PM as a Linux daemon named ***molpm*** that includes its own container.

3.10. The MoL Filesystem Server

The FS carries out all the file system calls, such as *read*, *mount*, and *chdir*. MoL implements a modified version of Minix FS as a Linux daemon named ***molfs*** that includes its own container. There are no significant difference between Minix FS server and ***molfs***.

4.Conclusions and Future Works

Minix has proved to be a feasible test-bed for OS development and extensions that could be easily added to it. Its modern architecture based on a microkernel and device drivers in user-mode makes it a reliable OS. The message transfer is the paradigm used

by Minix to implement system calls and kernel calls. Those features facilitate the construction of an OS-based Virtual Machine using any available protocol, interface or abstraction.

During the development of this project several alternatives arose which were disregarded in favor of a simplest programming and debugging instance. Those were not necessarily discarded and might be considered in future versions. Much work still ahead, such as the deployment of stable production-mode MoL version, the inclusion of security features, the virtualization of more devices, performance optimizations, binary compatibility, process and system status monitoring, etc.

MoL is able to behave as a primitive and trivial DOS running its servers, drivers and user processes on different hosts. This capability led the authors to think about MoL as the foundation for a Single System Image (SSI) DOS as a Multiserver OS running on top of other OSs. It will support transparently load sharing through live process migration, high reliability, fault tolerance and other facilities where user processes running on a cluster of machines will appear as they would be running on a single system. That DOS will be the subject of a forthcoming Ph.D. dissertation of one of the authors of this article (Pessolani).

The primary contribution of this work is to present an approach of a Multiserver OS running in user-space using OS-based virtualization. Although it can be used in a wide variety of applications as mentioned in Section 1, the authors' main target is about distributed processing and SSI-DOSs.

References

1.Tanenbaum, Woodhull. *Operating Systems Design and Implementation*, 3rd Edition, Prentice Hall, 2006.

2.MINIX3 Home Page, http://www.minix3.org/

3.Herder. *Towards A True Microkernel Operating System*. Vrije Universiteit Amsterdam. Master degree thesis 2005.

4.Herder, Gras, Tanenbaum. *Modular system programming in Minix 3*, Login:. April 2006.

5.Lowman. *Virtual Machines for the Intel x86 Architecture*. http://lowmanio.co.uk/. November 2007.

6.Yang Yu. *OS-level Virtualization and Its Applications*. Stony Brook University. PhD. degree thesis 2004.

7.Dike. *A user-mode port of the Linux kernel*. USENIX Association. Proceedings of the

4th Annual Linux Showcase & Conference, Atlanta Oct 10 –14, 2000.

8. Linux Vserver. http://linux-vserver.org/Paper

9. OpenVZ Wiki. http://wiki.openvz.org/Main_Page

10. lxc Linux Containers. http://lxc.sourceforge.net

11.Henning Kamp, Watson. *Jails: Confining the omnipotent root*. In Proc. 2nd Intl. SANE Conference. 2000.