Enhancing MINIX 3 Input/Output Performance Using a Virtual Machine Approach

Pablo Andrés Pessolani - César Daniel González
Departamento de Sistemas de Información - Facultad Regional Santa Fe
Universidad Tecnológica Nacional - Santa Fe – Argentina
{ppessolani,danignzz}@hotmail.com

Abstract. MINIX 3 is an open-source operating system designed to be highly reliable, flexible, and secure. The kernel is extremely small and user processes, specialized servers and device drivers run as user-mode insulated processes. These features, the tiny amount of kernel code, and other aspects greatly enhance system reliability. The drawbacks of running device drivers in user-mode are the performance penalties on input/output ports access, kernel data structures access, interrupt indirect management, memory copy operations, etc. As MINIX 3 is based on the message transfer paradigm, device drivers must request those operations to the System Task (a kernel representative process) sending request messages and waiting for reply messages increasing the system overhead. This article proposes a direct call mechanism using a Virtual Machine (VM) approach that keeps system reliability running device drivers in user-mode but avoiding the message transfer, queuing, de-queuing and scheduling overhead.

Keywords: Operating System, microkernel, Input/Output, Device Drivers.

1. Introduction

MINIX [1] is a complete, time-sharing, multitasking Operating System (OS) developed from scratch by Andrew S. Tanenbaum. It is a general-purpose OS 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.

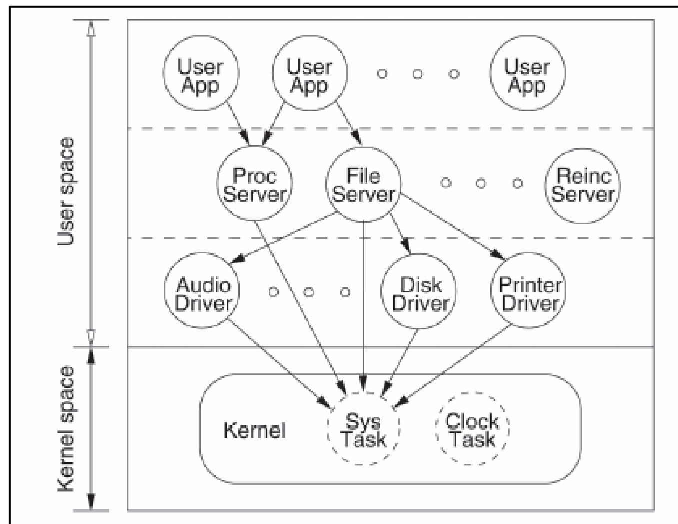

Figure 1: The Internal Structure of MINIX 3 [From [4]]

The drawbacks of running device drivers in user-mode are the performance penalties [5] on I/O ports operations, the access to kernel data structures, the indirect interrupt handling mechanism, the operations of copy memory blocks among different address spaces, etc. As MINIX 3.X is based on the message transfer paradigm, device drivers must request those operations to the System Task (a kernel representative server process) sending request messages and waiting for reply messages. As sending/receiving messages with rendezvous to another process results in several process switches (including system scheduler invocations), that approach impose a considerable overhead to the system performance.

This article is about the following part of the work presented in [6], but using a Virtual Machine (VM) approach. It does not only have impact on performance but in the device driver programming paradigm.

The rest of this article is organized as follows. Section 2 and Section 3 are overviews of I/O management on MINIX 2.X and MINIX 3.X respectively. Section 4 describes the proposed I/O model. Performance evaluation is detailed in Section 5. Finally, Section 6 presents conclusions and future works.

2. MINIX 2 and MINIX 3 Input/Output Overview

For each class of I/O device present in a MINIX system, a separate I/O task (device driver) is present [7]. These drivers are full-fledged processes, each one with its own

state, registers, stack, and so on. Device drivers communicate with each other and with system server processes using message passing.

Although each device driver is an independent process, in MINIX 2 they share kernel memory address space and run in privileged mode, therefore:

- They can access to kernel data structures (as the process table) to get needed information of processes.
- They can use kernel routines (as copy memory blocks).
- They can install their own interrupt handlers.
- They can share code with other device drivers.
- They can execute privileged I/O CPU instructions.

A drawback of MINIX 2 structure is that device drivers run in privileged mode and they share the same address space with the microkernel. The overall system can be affected by a device driver with errors, as occurs in monolithic OSs.

One of the main goals of MINIX 3 is reliability [2], but greater reliability will also improve security.

The design of MINIX 3 is based on the following principles:

- *Small kernel size*: It is based on the following statement "less code, less errors".
- *Bugs isolation*: In monolithic operating systems, device drivers reside in the kernel. A single bad line of code in a driver can bring down the system. Drivers cannot execute privileged instructions, perform I/O, or write to absolute memory. They have to make Kernel Calls for these services and the kernel checks each call for authority.
- *Limit drivers' memory access*: In monolithic operating systems, a driver can write to any word of memory and thus accidentally trash user programs. In MINIX 3, the driver must request the kernel to write, making it impossible for it to write to addresses outside the buffer.
- *Survive bad pointers*: Dereferencing a bad pointer within a driver will crash the driver process, but will have no effect on the system as a whole. A Reincarnation Server (RS) will restart the crashed driver automatically.
- *Tame infinite loops*: If a driver gets into an infinite loop, the scheduler will gradually lower its priority until it becomes the idle process.
- *Limit damage from buffer overruns*: MINIX 3 uses fixed-length messages for internal communication, which eliminates certain buffer overruns and buffer management problems.
- *Restrict access to kernel functions*: Device drivers obtain kernel services (such as copying data to users' address spaces) by making Kernel Calls. The MINIX 3 kernel has a bitmap for each driver specifying which calls it is authorized to make.
- *Restrict access to I/O ports*: The kernel also maintains a table telling which I/O ports each driver may access. As a result, a driver can only touch its own I/O ports.
- *Restrict communication with OS components*: Not every driver and server needs to communicate with every other driver and server. Accordingly, a per-process bit map determines which destinations each process may send to.
- *Reincarnate dead or sick drivers*: A special process, called the Reincarnation Server (RS), periodically pings each device driver. If the driver dies or fails to respond correctly to pings, the RS automatically replaces it by a fresh copy.

- *Integrate interrupts and messages*: When an interrupt occurs, it is converted at a low level to a notification sent to the appropriate driver.

A consequence of making major system components independent processes outside the kernel is that they are forbidden from doing actual I/O [1], manipulating kernel tables and doing other things operating system functions normally do. These special services are handled by the System Task (SYSTASK) through Kernel Calls.

SYSTASK offers services to Device Drivers and Servers processes to do I/O operations, access kernel tables, and do other things they need to, all without being inside the kernel. The SYSTASK and the CLOCK Task are the only processes that run with kernel privileged levels sharing the kernel memory address space allowing them to access kernel tables and execute privileged CPU instructions.

MINIX 3 requests need two additional messages than on MINIX 2. Those messages are used to request the SYSTASK for I/O operations that device drivers need to execute because they have not privileges for instructions like IN/OUT. The SYSTASK, that has the required privileges, executes I/O operations and memory copy functions on behalf of Device Drivers Tasks.

Performance tests report that the average system overhead introduced by this approach is limited to 5-10% [8] against MINIX 2.

Figure 2 shows the different classes of system services provided by MINIX 3 at different levels.

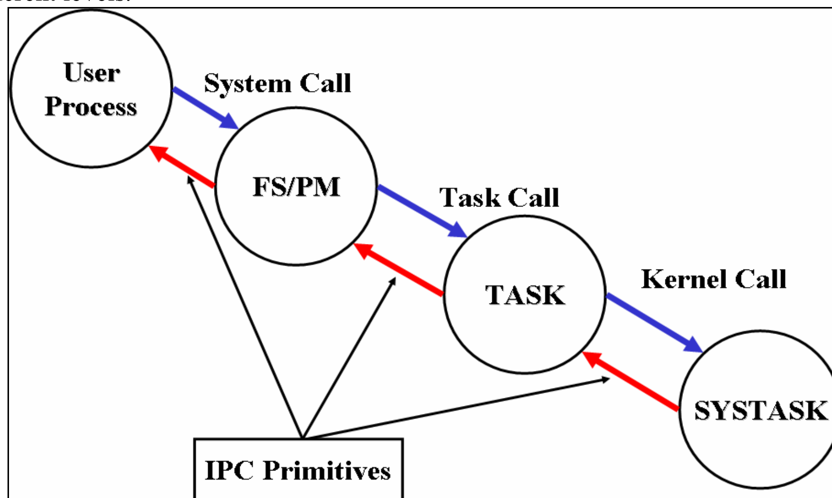

Figure 2: MINIX 3 User-System Communication

The following classification helps to clarify the terminology used in this article:

- *System Calls*: They are required by the POSIX standard and are used by User processes. System Calls are transformed into messages to Server processes.
- *Task Calls*: They are requests from Server processes to Tasks.
- *Kernel Calls*: They are requests from Device Drivers or Servers processes to the SYSTASK.
- *IPC Primitives*: They are used for interprocess communication such as *send*, *receive*, and *notify* used to implement System/Task/Kernel Calls.

3. Enhancing Input/Output Performance

MINIX 3 Kernel Calls are based on a Client/Server model but the message transfers used introduce an additional overhead to the system, particularly on I/O and memory copy operations that are frequently used by device drivers.

Each I/O requested operation can be divided in the following steps to detail the overhead imposed by the message transfer approach:

1. A Device Driver Task makes a CPU Trap to request SYSTASK for a Task Call using the *sendrec()* primitive.
2. The kernel saves the context of the requesting Device Driver Task.
3. The kernel checks the message destination against the permitted destinations.
4. The kernel copies the message buffer from the requesting Device Driver Task address space to the SYSTASK message buffer.
5. The kernel sets the requesting Device Driver Task in UNREADY state, and removes it from the READY queue.
6. The kernel sets the SYSTASK in READY state, and inserts it into the READY queue.
7. The kernel calls the scheduler and it selects SYSTASK as the next process to run.
8. The kernel restores the SYSTASK context.
9. The kernel dispatches the SYSTASK.
10. The SYSTASK checks the privileges of the requested operation.
11. The SYSTASK executes the requested operation.
12. The SYSTASK makes a trap to the CPU to request a SEND operation for the reply.
13. The kernel saves the SYSTASK context.
14. The kernel checks the message destination against the permitted destinations.
15. The kernel copies the message buffer from the the SYSTASK to the requesting Device Driver Task message buffer.
16. The kernel sets the requesting Device Driver Task in READY state, and inserts it into the READY queue.
17. The SYSTASK traps the CPU making a *receive(),* waiting for a new request.
18. The kernel sets the SYSTASK in UNREADY state, and removes it from the READY queue.
19. The kernel calls the scheduler and it selects the Requesting Task as the next process to run.
20. The kernel restores the Requesting Task context.
21. The system returns to User mode.

The performance penalty of the IPC model with destination checks is 22% as it is reported in [8] on executing a *getpid()* System Call test.

A Hardware Abstraction Layer (HAL) based I/O approach is proposed in [7] that shows a 43% of performance improvement over standard MINIX 3.

Modern CPUs have a protection mechanism based on exceptions to avoid that processes with lower privileges can execute privileged instructions. Virtual Machine Monitors (VMM) take advantage of this mechanism to trap sensitive instructions to invoke the hypervisor [9].

A related approach to enhance MINIX 3 I/O performance is based on replacing I/O request and reply messages to SYSTASK by the use of exceptions caused by the

execution of privileged I/O instructions in User Mode. The Device Driver Task running in user-mode executes I/O instructions as it is running in kernel-mode, but the CPU raise a general protection exception and the kernel checks for the offending instruction. If the process has the suitable privileges for that instruction and for the data, the kernel runs the instruction on behalf of the device driver.

Using exception based I/O avoid calling the scheduler as if the device driver has executed the I/O operations in kernel-mode.

Each exception based I/O operation can be divided into the following steps:

1. A Device Driver Task executes a privileged I/O instruction.
2. The CPU raises a General Protection Exception (GPE) and calls the kernel exception routine.
3. The kernel saves the context of the requesting Device Driver Task.
4. The kernel checks the privileges of the process for the instruction.
5. The kernel executes the requested instruction on behalf of the Device Driver Task.
6. The kernel restores the context of the Device Driver Task.
7. The system returns from exception to user-mode.

The *sys_privctl* Kernel Call can be used to set the privileges of each process in the system. This call can only be used by a privileged user-mode Server, and is used, for example, to restrict the I/O ports that can be used by individual drivers [10].

3.1. Changes to MINIX 3

The following Kernel Calls (shown as library functions) were considered as an example:

- *int sys_inb(port t port, u8 t *byte):* Read a value into byte from port.
- *int sys_outb(port t port, u8 t byte):* Write a value byte into port.

Two new System Calls were added in equivalence of those Kernel Calls:

- *int i386inb(port t port):* Return a value read from port.
- *int i386outb(port t port, u8 t byte):* Write a value byte into port.

Those device drivers that want to use the VM approach must change their source code to use the new I/O System Calls instead of Kernel Calls. As a proof of concept, the tests were carried out on the RS-232 code of the tty device driver. A new library named ***libio386*** must be linked with the device drivers object programs that use the VM approach.

The following code shows the *i386inb* and the *i386outb* functions that only use the INB and OUTB instructions. Therefore, a device driver running in user-mode that use these functions will generate exceptions trying to execute INB/OUTB privileged instructions.

```
! Read an (unsigned) byte from the i/o port
! PUBLIC unsigned i386inb(port_t port);
        .align 16
_i386inb:
         mov edx, 4(esp)! port
         sub eax, eax
         inb dx    ! read 1 byte
         ret
! Write  value  (cast to a byte)  to the I/O port
```

```
! PUBLIC void i386outb(port_t port, u8_t value);
         .align 16
_i386outb:
         mov  edx, 4(esp)! port
         movb al, 4+4(esp)! value
         outb dx    ! output 1 byte
         ret
```

All code changes and additions in the original MINIX source code to support the proposed VM approach are preceded by `#ifdef HALVM` and finished by `#endif` to avoid affecting the compilation of standard MINIX. The HALVM macro controls the compilation of the VM approach code. It is defined in ***/usr/include/minix/config.h***.

The kernel function *exception()* was changed to implement these limited VM Monitor code. This code performs the following operations:

- Gets the address of the offending CPU instruction.
- Decodes the instruction.
- If the instruction is INB or OUTB, it checks for the privileges of the process
- If the process has suitable privileges, it executes the INB or OUTB instructions
- Returns from exception

```
#if HALVM
phys_opcode = umap_local(saved_proc, T, old_eip, 4);
if (phys_opcode!=0 && vec_nr==0x0D && trap_errno==0)
 {
 phys_copy(phys_opcode,vir2phys(&opcode)
        ,phys_bytes) 4);
 if((priv(saved_proc)->s_call_mask & (1 <<SYS_DEVIO)))
   {
   switch(opcode[0])
      {
      case 0xEC:  /* IN AL,DX */
            saved_proc->p_reg.retreg =
                inb(saved_proc->p_reg.dx);
            saved_proc->p_reg.pc++;
            return;
      case 0xEE:  /* OUT DX,AL  */
            outb(saved_proc->p_reg.dx,
                saved_proc->p_reg.retreg);
            saved_proc->p_reg.pc++;
            return;
      case 0x66:
            switch(opcode[1])
               {
               case 0xED: /* IN AX,DX  */
                saved_proc->p_reg.pc+=2;
                return;
               case 0xEF: /* OUT DX,AX  */
                saved_proc->p_reg.pc+=2;
```

```
                    return;
                  }
                break;
              }
            }
          else
          kprintf("exception():I/O
           request from %d denied\n", saved_proc->p_nr);
          }
#endif /* HALVM */
```

As it is shown in the exception source code, the process' I/O privileges are checked before the kernel executes the I/O operations on behalf of the process.

4. Performance Evaluation

This section describes the tests performed on MINIX Standard, MINIX with the I/O HAL [6] and MINIX with the I/O VM approach.

A set of System Calls and kernel functions were added to supply the user with a set of performance measurement tools. They are based on the Pentium CPU Time-Stamp Counter (TSC) [6]. MINIX standard code has the kernel function to read TSC, but it has not any Kernel Call that use it.

The tests were performed sending and receiving files through the RS-232 serial port at 38400 Kbps. The I/O performance test results are presented in Figure 3 and the numeric results in Table 1. The time units are CPU Hz reported by the TSC Register.

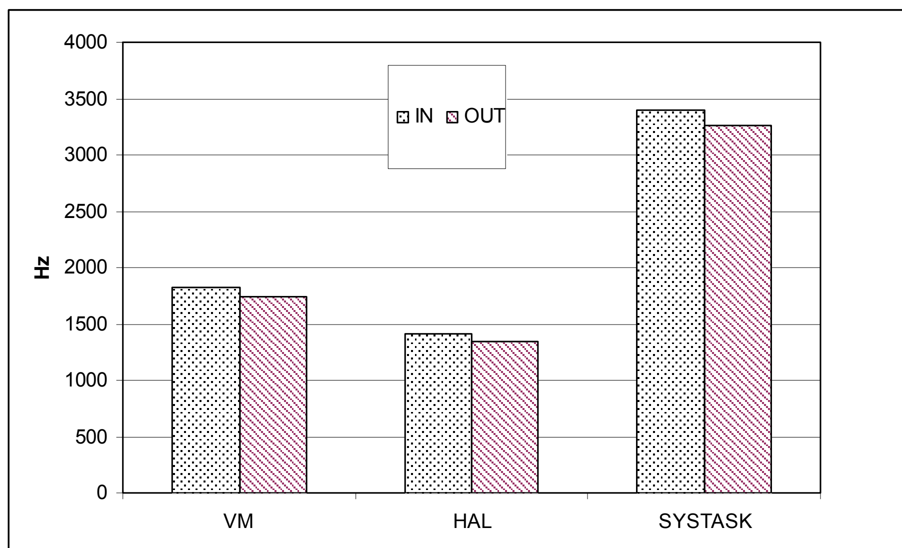

Figure 3: I/O Performance Tests Results

The average time to perform I/O operations with the I/O VM approach is about 53% of the avarage time used by MINIX Standard SYSTASK, but 29% greater than the I/O HAL approach.

Table 1: I/O Performance Tests Results

	I/O VM		**I/O HAL**		**SYSTASK**	
	IN	**OUT**	**IN**	**OUT**	**IN**	**OUT**
Average	1818,0	1737,5	1415,5	1341,3	3392,7	3264,9
Std Deviation	406,4	129,0	343,9	99,6	537,9	128,8
Relative Performance	1,00	1,00	1,28	1,30	0,54	0,53

The equipment used for the tests was an Intel Pentium MMX 233.9 MHz with a L1 Code Cache of 16 KB., L1 Data Cache of 16 KB, RAM size of 96 MB, SDRAM Acces Time 12 [ns], EDO Dram Acces Time 60 [ns].

5. Conclusions and Future Works

MINIX has proved to be a feasible testbed for OS development and extensions that could be easily added to it. Its modern architecture based on a microkernel and device drivers in user-level make it a reliable Operating System.

The message transfer is the paradigm used by MINIX to implement System Calls, Task Calls and Kernel Calls. MINIX 3 uses a new level of message transfer from Device Driver Tasks to the SYSTASK to execute privileged I/O instructions that the formers cannot execute in user mode. This new level of message transfer cause an additional overhead, but it can be avoided using a Virtual Machine Monitor paradigm based on CPU exceptions but limited to basic I/O operations.

Comparative performance results presented in this article show noticeable reduction of the I/O overhead using the proposed approach without sacrificing security, robustness and simplicity. This fact proves that it is possible to build systems which employ user-level device drivers, without significant performance degradation [11].

The VM approach allows that device drivers could be written as if they will be run with I/O privileges. Future works address the development of device drivers using privileged I/O operations on virtualezed devices into the kernel controlling remote real devices as if they would be real local ones.

References

1.Tanenbaum Andrew S., Woodhull Albert S., “Operating Systems Design and Implementation, Third Edition”, Prentice-Hall, 2006.

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

3.Jorrit N. Herder, "*Towards A True Microkernel Operating System*", master degree thesis, 2005.
4.Jorrit N. Herder, Herbert Bos, Ben Gras, Philiph Omburg ,Andrew S. Tanenbaum,"*Modular system programming in MINIX 3*", ;Login: April 2006.
5.Rogier Meur, "*Building Performance Measurement Tools for the MINIX 3 Operating System*", 2006.
6.Pessolani Pablo et al., "*Enhancing MINIX 3.X Input/Output Performance*", Aug. 2008
7.Tanenbaum Andrew S., Woodhull Albert S., "*Operating Systems Design and Implementation, 2nd Editión*", Prentice-Hall , 1999.
8.J. N. Herder, H. Bos, and A. S. Tanenbaum. "*A Lightweight Method for Building Reliable Operating Systems Despite Unreliable Device Drivers*". In Technical Report IR-CS-018 [www.cs.vu.nl/~jnherder/ir-cs-018.pdf], Vrije Universiteit, Jan. 2006
9.James Smith, Ravi Nair, "*Virtual Machines*", Morgan Kaurfmann, 2005.
10.Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum, "*Construction of a Highly Dependable Operating System*", Vrije Universiteit, 2006.
11.B. Leslie, P. Chubb, N. Fitzroy-Dale, S. G otz, C. Gray, L. Macpherson, D. Potts, Y. Shen, K. Elphinstone, and G. Heiser. "*User-level device drivers: Achieved performance*". J. Comput. Sci. & Technol., 20(5):654--664, Sep 2005.