Enhancing MINIX 3.X Input/Output Performance

Pablo Pessolani - Gustavo Weisz – Marisa Bardus – César Hein
Departamento de Sistemas de Información
Facultad Regional Santa Fe - Universidad Tecnológica Nacional – Argentina
{ppessolani,gweisz,mbarduz,chein}@frsf.utn.edu.ar

Abstract

MINIX 3.X 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 driver runs 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.X is based on the message transfer paradigm, device drivers must request those operations to the System Task (a special kernel representative process) sending request messages and waiting for reply messages increasing the system overhead.

This article proposes a direct call mechanism that keeps system reliability running device drivers in user-mode but avoiding the message transfer, queuing, dequeuing 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:

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

MINIX 3.X 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 on resource-limited and embedded computers and for applications requiring high reliability.

MINIX 3.X kernel is very small (4000 lines of executable code) and it is the only code that runs under kernel privilege levels. User processes, system servers including device drivers are insulated 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.
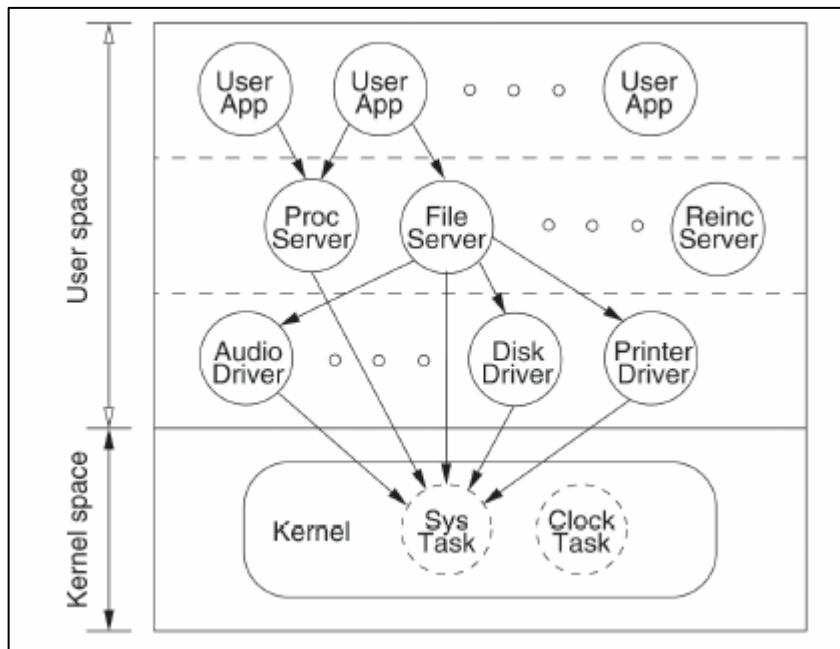

Figure 1:The Internal Structure of MINIX 3.X [From [4]]

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 special kernel representative server process) sending request messages and waiting for reply messages. As sending/receiving messages with rendevous to another process implies several process switches (including system scheduler invocations), that approach impose a significative overhead to the system performance.

This article propose a new Input/Output (I/O) model for MINIX 3.X that keeps system reliability running device drivers in user-mode but avoiding the message transfer overhead.

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 are detailed in Section 5. Finally, Section 6 presents conclusions and future works.

2   OVERVIEW OF INPUT/OUTPUT IN MINIX 2.X

For each class of I/O device present in a MINIX system, a separate I/O task (device driver) is present [6]. These drivers are full-fledged processes, each 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.X they share kernel memory address space as it is shown in Figure 2.
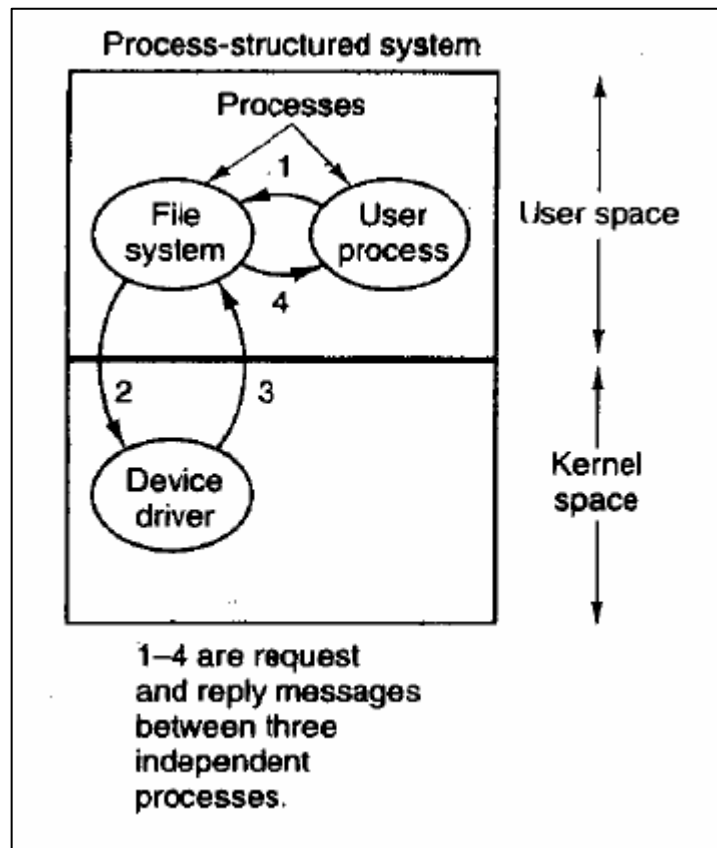

Figure 2: MINIX 2.X User-System Communication (From [6])

As device drivers share kernel address space and run in privileged mode, 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), the can install their own interrupt handler, share code with other device drivers, and they can execute privileged I/O CPU instructions.

All processes in the system can communicate using the following primitives:

- *send*: to send a message to a process.
- *receive*: to receive a message from a process.
- *sendrec*: to send a request message to a process and then receive a reply message from it.

Those primitives are implemented as CPU traps that change the processor from user-mode to kernel-mode.

Hardware interrupts are masked converting them into a message transfer. When a hardware interrupt occurs, the kernel notifies the corresponding task simulating to send it a message.

Figure 2 shows the message tranfers that a simple System Call as *ioctl* needs.

1. The User process sends a message to the File System server for an *ioctl* operation on a device.

2. The File System Server sends a message to the device task for an *ioctl* operation.

3. The device driver task replies the File System Server.

4. The File System replies the User-process.

Although they are endless debates on performance penalties of microkernel based OSs against monolithic ones, the performance impact of message transfers is real, but it must be considered against the benefits of process isolation, clean interfaces, unprivileged server processes, extensibility, etc. of the microkernel approach.

3   OVERVIEW OF INPUT/OUTPUT IN MINIX 3.X

A drawback of MINIX 2.X 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.X is reliability [2], but greater reliability will also improve security. The design of MINIX 3.X 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.X, the driver must 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.X 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.X kernel has a bit map 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 a interrupt occurs, it is converted at a low level to a notification sent to the appropriate driver.
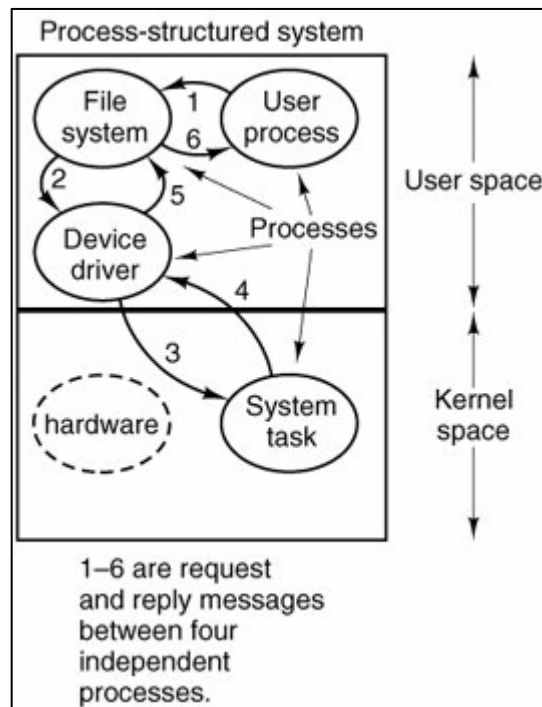

Figure 3: MINIX 3.X User-System Communication (From [1])

All processes in the system can communicate using the following IPC primitives:

- *send*: to send a message to a process but the sender is blocked until the destination process receives it.

- *receive*: to receive a message from a process but the receiver is blocked until the sender process send it.

- *sendrec*: to send a request message to a process and then receive a reply message from it.

- *notify*: to send a message to a process but the sender is not blocked if the destination is not waiting for it.

Those primitives are implemented as CPU traps that change the processor from user-mode to kernel-mode.

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* through *Kernel Calls*. It 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 System Task 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 can execute privileged CPU instructions.

The services provided by the System Task are described in [1]. The following is a reduced list of services requested by device drivers that are interesting for this article:

- *sys_devio*: Read from or write to an I/O port.

- *sys_sdevio*: Read or write string from/to I/O port.

- *sys_vdevio*: Carry out a vector of I/O requests.

- *sys_umap*: Convert virtual address to physical address.

- *sys_vircopy*: Copy using pure virtual addressing.

- *sys_physcopy*: Copy using physical addressing.

- s*ys_virvcopy*: Vector of virtual copy requests.

- *sys_physvcopy*: Vector of physical copy requests.

Some requests in MINIX 3.X need two additional messages (3 and 4) than on MINIX 2.X as it is shown in Figure 3,. Those messages are used to request the System Task for I/O operations that Device Drivers need to execute because they have not privileges for instructions like IN/OUT. The System Task, 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% [7] against MINIX 2.X.

The following are all kinds of system services provided by MINIX 3.X in different levels (Figure 4) 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 System Task.

- *IPC Primitives:* They are used for interprocess communication such as *send*, *receive*, and *notify* to implement System/Task/Kernel Calls.
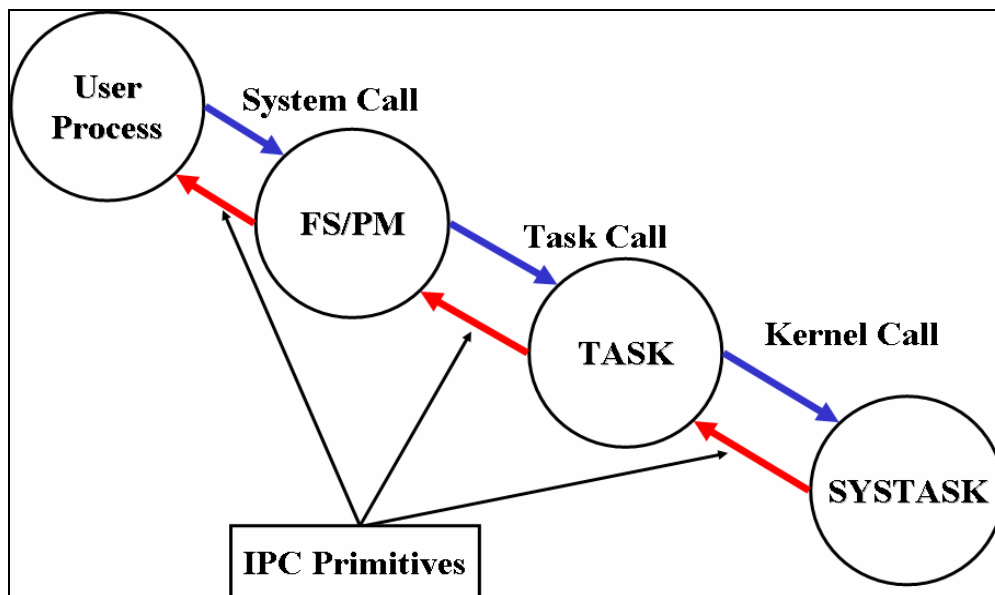

Figure 4: MINIX 3.X User-System Communication

4   ENHANCING INPUT/OUTPUT PERFORMANCE

A hardware abstraction layer (HAL) is a software layer between the physical hardware of a computer and the operating system that runs on that computer. Its function is to hide differences in hardware from most of the operating system kernel, so that most of the kernel-mode code does not need to be changed to run on systems with different hardware.

A HAL allows operations from user level Device Drivers to communicate with lower level components, such as directly with hardware. HALs are as application programming interfaces (API) that interact directly with hardware instead of a system kernel, therefore HALs require less processing time than APIs.

Some MINIX 3.X Kernel Calls can be seen as a HAL 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 discomposed in the following operations to detail the overhead:

1. A Device Driver Task makes a CPU Trap for the request 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 System Task message buffer.

5. The kernel puts the requesting Device Driver Task in UNREADY state, and removes it from the READY queue.

6. The kernel puts the System Task in READY state, and inserts it into the READY queue.

7. The kernel calls the scheduler.

8. The kernel restores the System Task context.

9. The kernel dispatches the System Task.

10. The System Task checks the privileges of the requested operation.

11. The System Task executes the requested operation.

12. The System Task makes a trap to the CPU to request a SEND operation for the reply.

13. The kernel saves the System Task context.

14. The kernel checks the message destination against the permited destinations.

15. The kernel copies the message buffer from the the System Task to the requesting Device Driver Task message buffer.

16. The kernel puts the requesting Device Driver Task in READY state, and inserts it into the READY queue.

17. The System Task traps the CPU making a *receive()*, waiting for a new request.

18. The kernel puts the System Task in UNREADY state, and removes it from the READY queue.

19. The kernel calls the scheduler.

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 [7] on executing a *getpid* System Call test.

The proposed approach to enhance MINIX 3.X I/O performance is based on replacing I/O request and reply messages to the System Task (SYSTASK) by an I/O HAL based on CPU traps extending the set of kernel primitives.

Using HAL based I/O Kernel Calls avoids calling the scheduler in the same way as if the I/O operations are done in kernel mode.

Each HAL based I/O requested operation can be discomposed in the following operations to detail how the overhead is reduced:

1. A Device Driver Task makes a CPU Trap for the request using I/O operation (HALINB or HALOUTB) for the request.

2. The kernel saves the context of the requesting Device Driver Task.

3. The kernel checks the privileges of the requested operation.

4. The kernel executes the requested operation.

5. The kernel restores the Requesting Task context.

6. The system returns 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 [8].

4.1 MINIX3 HAL Implementation

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

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

  o *int sys_outb(port t port, u8 t byte):* Write a value *byte* into *port*.

Two HAL Calls were added to the kernel in equivalence of those Kernel Calls:

  o *int hsys_inb(port t port):* Return a value read from *port*.

  o *int hsys_outb(port t port, u8 t byte):* Write a value *byte* into *port*.

Obviously, the Device Drivers source code must be changed to use HAL 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.

HAL Calls use a TRAP different from MINIX IPC primitives to avoid that the new code would affect the standard code and to permit that Device Drivers that use Kernel Calls still using them without affecting the normal device operation.

All code changes and additions are preceded by ***#ifdef HAL*** and finished by ***#endif*** to avoid affecting the compilation of standard MINIX. The ***HAL*** macro controls the compilation of the HAL code and it is defined *in /usr/include/minix/config.h.*

A diagram of the relations among changed/added files and functions is shown in Figure 5.

/usr/src/drivers/tty/rs232.c
hsys_in()
hsys_out()

usr/src/include/minix/syslib.h
#define hsys_outb(p,v)   hsys_out(HALOUTB, (p), (v))
#define hsys_inb(p)   hsys_in((p))

usr/src/lib/syslib/hsys_out.c
int hsys_out(function, port, value)
    → _halcall(HALOUTB, port, value, unused)

usr/src/lib/syslib/hsys_in.c
unsigned char hsys_in(port)
    → _halcall(HALINB, port, valor, unused)

usr/src/lib/syslib/halcall.c
_halcall(function, port, value, unused)
    → _ haloutb(port, value)
    → _halinb(port)

usr/src/lib/i386/rts/_ipc.s
    → __haloutb
    → __halinb

usr/src/include/ibm/interrupt.h
#define HAL_VECTOR   35

usr/src/kernel/protect.c
    → { h_call, HAL_VECTOR, USER_PRIVILEGE}

User Mode
Kernel Mode

usr/src/kernel/mpx386.s
_h_call
    → _hal_call

usr/src/kernel/proc.c
int hal_call (function, port, value, unused)
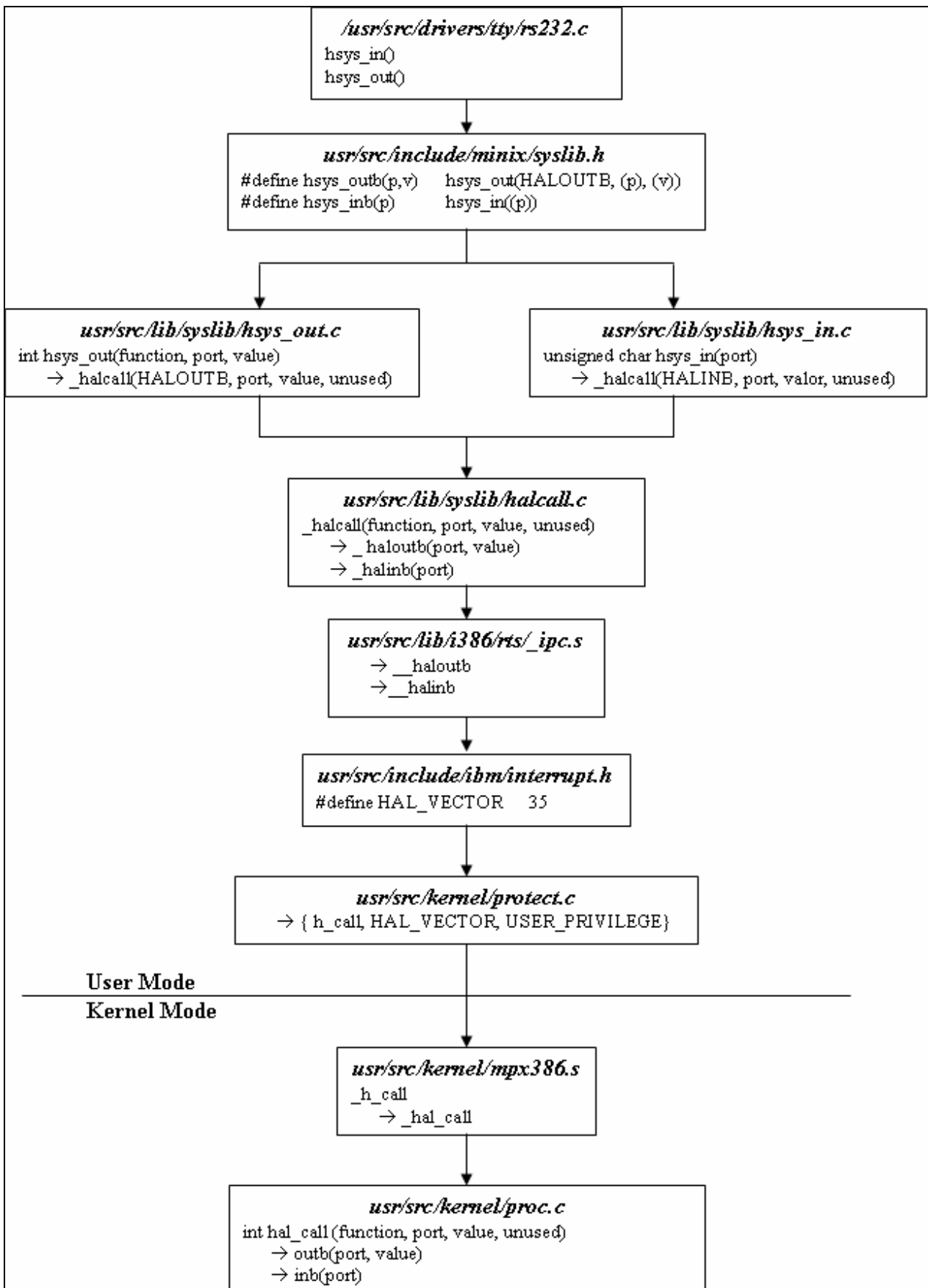   → outb(port, value)
    → inb(port)

Figure 5: HAL changed/added files and functions

The kernel function *hall_call()* was added in the */usr/src/kernel/proc.c* file.

```
#ifdef HAL
/*===========================================================================
 *                                hal_call
 *===========================================================================*/
PUBLIC int hal_call (function, port, value, unused)
int function;                        /* IO function = HALINB or HALOUTB */
int port;                            /* port to read or write */
int value;                           /* value to write on a port */
long unused;                         /* unused  */
{
/* HAL calls are done by trapping to the kernel with an INT instruction.
 * The trap is caught and hal_call() is called to OUTB or INB a value to/from a port */

int result;                                  /* the system call's result */

/* See if the process have IO priviledges and try to handle it. */
if ( !(priv(proc_ptr)->s_call_mask & (1 << SYS_DEVIO))) {
        kprintf("hall_call: request %d from %d denied \n",function,proc_ptr->p_nr);
        return(ECALLDENIED);
}

/* Now check if the call is known and try to perform the request. The  HAL calls that exist in MINIX-HAL are:
 *               -HALOUTB: output a byte to a port
 *               -HALINB: input a byte from a port
 */
switch(function) {
                case HALOUTB:
                        outb(port, value);
                        result = OK;
                        break;
                case HALINB:
                        result = inb(port);
                        break;
                default:
                        kprintf("hall_call: bad_call %d from %d \n",function,proc_ptr->p_nr);
                        result = EBADCALL;  /* illegal HAL call */
                        break;
                }
return(result);
}
#endif /* HAL */
```

4.2 I/O Performance Measuraments

A set of mechanisms were introduced to evaluate I/O performance among the standard MINIX 3.X and the modified MINIX 3.X with I/O HAL layer. They are based on the Pentium CPU Time-Stamp Counter (TSC). MINIX standard code has the kernel function to read TSC but it has not any Kernel Call that can be used by a Device Driver Task.

All code changes and additions are preceded by **#ifdef HAL_TEST** and finished by **#endif** to avoid affecting the compilation of standard MINIX. The **HAL_TEST** macro controls the compilation of the performance test code and it is defined *in /usr/include/minix/config.h.*

The Kernel Call added to get the TSC value has the following prototype:

   *int sys_get_tsc (int function, unsigned long int *h32TSC, unsigned long int *l32TSC));*

where:

function: The value GETTSC.

h32TSC: A pointer to the high order 32 bits of the TSC value read.

l32TSC: A pointer to the low order 32 bits of the TSC value read.

The kernel function *get_tsc_k()* was added in the */usr/src/kernel/proc.c* file.

```
/*===========================================================================*
 *                            get_tsc_k                                      *
 *===========================================================================*/
#ifdef HAL_TEST
PUBLIC int get_tsc_k(function, h32TSC, l32TSC, unused)
int function;    /* function = GETTSC   */
unsigned long int *h32TSC; /* high 32 bits of TSC */
unsigned long int *l32TSC; /* low 32 bits of TSC  */
int *unused;
{
  unsigned long int kHigh, kLow;
  phys_bytes src_phys, dst_phys;

  read_tsc(&kHigh, &kLow);
  src_phys = vir2phys(&kHigh);
  dst_phys = numap_local(proc_ptr->p_nr, (vir_bytes) h32TSC, sizeof(unsigned long int));

  if ((src_phys != 0) && (dst_phys != 0))
    phys_copy(src_phys, dst_phys, sizeof(unsigned long int));

  src_phys = vir2phys(&kLow);
  dst_phys = numap_local(proc_ptr->p_nr, (vir_bytes) l32TSC, sizeof(unsigned long int));

  if ((src_phys != 0) && (dst_phys != 0))
    phys_copy(src_phys, dst_phys, sizeof(unsigned long int));

  return (OK);
}

#endif
```

5 PERFORMANCE EVALUATIONS

This section describes the tests performed on MINIX Standard and MINIX with the I/O HAL modifications.

The tests were performed sending and receiving files through the RS-232 serial port at 19200 and 38400 Kbps. The I/O performance test results are presented in Table 1. The time units are CPU Hz reported by the TSC Register.

Table 1: I/O Performance Tests

	MINIX STANDARD		MINIX-HAL	
	IN	OUT	IN	OUT
Average	3268	3346	1435	1454
Std Deviation	1053	785	754	687

The avarage time to perform I/O operations with the I/O HAL is 43% of the avarage time used by MINIX Standard.

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].

6 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 mode 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.X uses a new level of message transfer from Device Driver Tasks to the SYSTASK to execute privileged I/O instructions that the formers can not execute in user mode. This new level of message transfer cause an additional overhead, but it can be avoided breaking the paradigm with an I/O HAL based on CPU Traps limited to basic I/O operations.

Comparative performance results presented in this article prove noticeable reduction of the I/O overhead using the proposed approach without sacrificing robustness and simplicity.

Planed future works will allow User Mode Device Drivers to execute privileged I/O instructions that will be traped by privilege violations and they will be executed in Kernel Mode as is done by some Virtual Machine approaches. New calls will be included to extend the HAL to support word I/O operations and to copy regions of memory, using either virtual or physical addresses to demonstrate that it is possible to build systems which employ user-level device drivers, without significant performance degradation [9].

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] Tanenbaum Andrew S., Woodhull Albert S., "*Operating Systems Design and Implementation, 2nd Edición*", Prentice-Hall , 1999.

[7] 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

[8] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum, "*Construction of a Highly Dependable Operating System*", Vrije Universiteit,2006.

[9] 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.