Modifying the Behaviour of Minix System Calls through the Redirection of Messages

Pablo Andrés Pessolani
Departamento de Sistemas de Información - Facultad Regional Santa Fe
Universidad Tecnológica Nacional - Santa Fe – Argentina
ppessolani@frsf.utn.edu.ar

Abstract. Minix 3 is an open-source operating system designed to be highly reliable, flexible, and secure. The kernel is small and user processes, specialized servers and device drivers runs as user-mode isolated processes. Minix is a client/server operating system that uses message transfers as communication primitives between processes. Minix system calls send messages to request for services to the Process Manager Server (PM) or the File System Server (FS), and then waiting for the results. The request messages refer to destination processes with fixed endpoint numbers for each server. This article proposes changes to the Minix kernel that allow the redirection of messages to different servers other than the standard FS or PM, without changes in the source code or binary code of programs.

Keywords: Operating System, microkernel, IPC, message transfer.

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 clean 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.

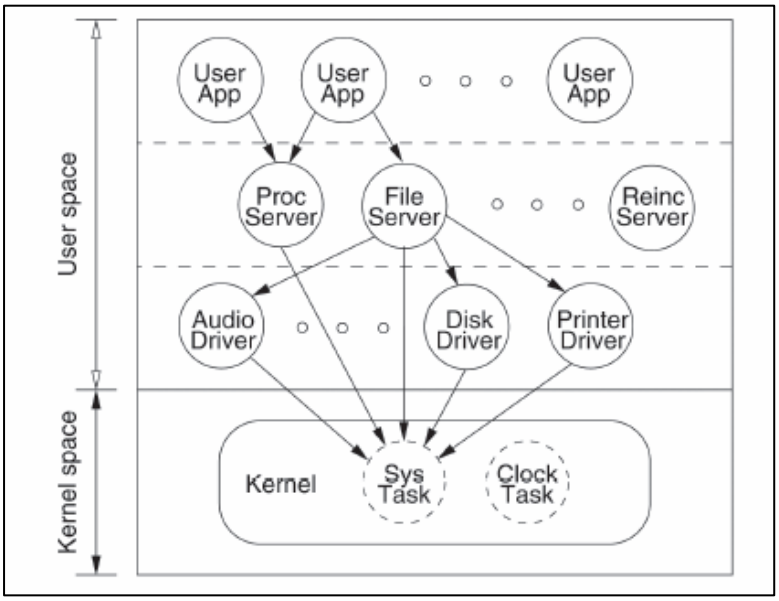

Figure 1: The Internal Structure of Minix 3 [From [4]]

A process makes system calls to request OS services, and the system calls may deliver the user requests to other functions of the OS which process the requests and return the result to the caller. Minix implements system calls using message transfers, packaging function arguments and the results in the same way as RPC does. The following source code shows how it is done:

```
PUBLIC int _syscall(who, syscallnr, msgptr)
int who;        /* destination server i.e. PM or FS */
int syscallnr;  /* System Call number */
register message *msgptr; /* pointer to the message */
{
  int status;
  msgptr->m_type = syscallnr; /* System Call number */
/* Send the Request and wait for the Reply */
  status = _sendrec(who, msgptr);

  if (status != 0) {msgptr->m_type = status; }
    if (msgptr->m_type < 0) {errno = -msgptr->m_type;
         return(-1);  }
       return(msgptr->m_type);
}
```

The POSIX system call *getpid()* is implemented sending a GETPID request to the PM and waiting for the reply from the server:

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

PM is a constant define as:

```
#define PM                    PM_PROC_NR
#define PM_PROC_NR 0  /* process manager */
```

The destination process is hard coded into the system call as a constant restricting that system calls can only be served by PM or FS Servers.

The development of some useful features as remote process execution, multiple filesystems support, multiple processing environments or personalities, proxy and gateway servers, security reference monitors, system call profiling, etc. would be simplified if a system call message transfer would be served up by other servers without the need of changing the program. The problem and the solution were described by the Minix`s author, prof. Andrew Tanenbaum:

"Currently FS_PROC_NR is defined as a hard constant (1). Instead, it could be a per-process entry in the process table, so when a process sent a message to 1, this would tell the kernel to look up the real number in the process table. This would mean every process could have its "own" file server. Same for PM_PROC_NR. For a specific process, the number of the "File Server" could be a user-level gateway process that had a permanent TCP connection to a remote server. The command that the user sent would then be forwarded to gateway locally and from there it would be forwarded to the remote machine and executed there. That would allow using remote file systems. On the remote machine would be another gateway process that did the work and marshalled and returned the answer…."

Redirection of Messages can satisfy those needs where *redirection* basically refers to send a message to an entity but really the message is deliver to another.

This article examines several approaches and pieces of modified or added source code to the Minix 3.1.2a kernel as a proof of concept of Redirection of Messages. It must be clear that it is not a definite and refined version of Minix 3.

The rest of this article is organized as follows. Section 2 is an overview of Minix 3 system calls implementation, Section 3 describes the Redirection of Messages mechanism. Section 4 refers to the proposed *relay()* IPC primitive. Finally, Section 5 presents conclusions and future works.

2. Overview Minix 3 System Calls Implementation

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

- *send():* to send a message to a process.
- *receive()*: to receive a message from a specified process or from any process.
- *sendrec():* to send a request message and to receive the reply from a process.
- *notify():* a non blocking send of a special message type.

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

As it was mentioned in the previous section, Minix uses message transfers to implement system calls. Usually, the destinations of request messages are the PM server and FS server.

Minix does not have a single process table, it is scattered among servers and the kernel. The kernel process table keeps attributes, status and statistical information of each process. The FS and PM have their own process tables with fields with specific information that they need.

The kernel process table has (NR_TASKS+NR_PROCS) entries (See Figure 2) where NR_TASKS counts the following special tasks:

- *The Idle task*: It runs when no other runnable process can be scheduled.
- *The Clock task*: It accepts only messages from the timer interrupt handler. It keeps the *realtime* variable that counts timer ticks.
- *The System task*: It represents the kernel and shares its address space (it is like a kernel thread). It accepts requests for special kernel services (called *kernel calls*) from drivers and servers and carry them out.
- *A bogus Kernel task*: Really this task does not exist but its process number is used by interrupt handlers as the source process when they send *notify()* messages to device driver tasks.

NR_PROCS entries are available for servers, device drivers tasks and user processes. It can be specified in a configuration file but the operating system must be compiled completely.

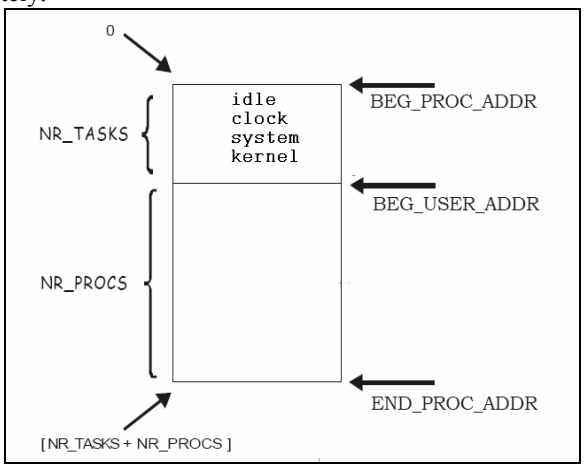

Figure 2: Minix 3 Kernel Process Table

The kernel *proc* data structure that describes a process has two fields related to message tranfers:

```
proc_nr_t   p_nr; /* index of this entry in the table  */
int         p_endpoint;/* endpoint number */
```

The *p_nr* field is the index of the entry in the kernel process table minus NR_TASKS. Therefore, the first process in the table (*proc[0]*) has *p_nr = (-NR_TASKS)*. The reader should not confuse *p_nr* 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 identify Minix processes to be used as a parameter in system calls for processes management, such as adjusting the process's priority.

The *endpoint* field uniquely identifies a single processes with respect to IPC and its associate the *p_nr* field with the *generation* number of the slot. Each slot has a *generation* number that counts how many processes has occupied that slot. Each time a new process occupies the slot, the *generation* count is increased. This action prevents that a message addressed to a dead process that has previously used the slot will be delivered to the new one.

The kernel implements IPC primitives using a function with the confusing name *sys_call()* defined as follow:

```
int sys_call(call_nr, src_dst_e, m_ptr, bit_map)
```

The parameter *call_nr* really is the code of an low-level IPC primitive (SEND, RECEIVE, SENDREC, NOTIFY, explained in Section 4).

The *src_dst_e* parameter is the source/destination endpoint number.

The *m_ptr* parameter is a pointer to the request message, and the *bit_map* parameter is a bitmap of flags that change the behavior of the call.

3. Redirection of Messages

Redirection of Messages refers on resolving the process number of destinations process (*p_nr*) through a table instead of setting it as a constant. It requires adding new kernel data structures and making some modifications of the *sys_call()* function to send/receive messages to/from endpoints that are referred indirectly.

3.1. Data Structures

The following approaches were analyzed for Redirection of Messages:

- *Two new fields into the process data structure*: Adding one field for the PM process number, and other field for the FS process number. I.e. when a user processes makes a system call to the PM or FS, the kernel gets the destination's endpoints from those fields. This approach limits the Redirection of Messages only to user process and POSIX system calls.
- *A per process Servers Table*: Each process has its own servers table where the index is the *p_nr* of the server used to get the server endpoint.
- *A fixed number of system wide Servers Tables*: As possibly not all processes need Redirection of Messages, and surely some of them could use the same table, only a fixed number of servers tables are needed. Each table represents an execution environment that can be set to each process similar to the *priv* table that Minix uses for privileges management.

A convenient table size could be (NR_PROCS+NR_TASKS) to allow the Redirection of Messages not only to user-space processes but to system processes and tasks too. The experimental version has NR_SVRTABS number of tables named *svrtab[]* with (NR_TASKS + NR_PROCS) entries each.

```
proc_nr_t  svrtab[NR_SVRTABS][NR_PROCS+NR_TASKS];
```

A new field in the kernel process descriptor was added to store the servers table that the process will use in system calls message transfers. This field is named *p_svrtab,* and represents the execution environment for the process. The servers table establishes the set of servers that will reply for system calls requests for the process environment.

3.2. Data Structures Initialization

The initialization code sets the numbers of default servers used by Minix for all tables. System programmers can change the servers numbers of a table to redirect some system calls messages to new servers leaving the other servers numbers unchanged for a standard behaviour.

The kernel *svrtab[]* is initialized in *main()* function of the kernel as it is shown in the following source code:

```
void init_svrtab(void)
{
  int i, j;
  for( j = 0; j < (NR_PROCS+NR_TASKS); j++)
       for (i = 0; i < NR_SVRTABS; i++)
             svrtab[i][j] =  (j-NR_TASKS);
}
```

All entries in *svrtab[]* are initialized with the corresponding *p_nr*, therefore the j-th entry of each table is initialize with the value (j-NR_TASKS) as it can be seen in Table 1.

Table 1: Kernel Servers Table – *svrtab[]*

Server	svrtab[0]	svrtab[1]	svrtab[2]	svrtab[3]	….
-4	-4	-4	-4	-4	….
-3	-3	-3	-3	-3	….
-2	-2	-2	-2	-2	….
….	….	….	….	….	….

The *p_svrtab* field of a process is initialized with the value 0, therefore it use *svrtab[0]* as its default table. This means that if *svrtab[0]* table has not been changed, the servers numbers will be the same as in the official Minix version, therefore the system calls will have the standard behaviour.

```
  #ifdef MSGIND /* rp points to the process descriptor */
  rp->p_svrtab = 0; /* Servers Table = 0  */
  #endif
```

The *p_svrtab* is a process attribute that will be inherited by its children when the process forks. The internal function of the SYSTEM task copies this field from parent to child process descriptor.

```
#ifdef MSGIND /* rpc points to child's  descriptor */
              /* rpp pointes to parent's descriptor */
    rpc->p_svrtab = rpp->p_svrtab;
#endif /* MSGIND */
```

3.3. Changes in IPC Primitives

As system calls use the *sendrec()* IPC primitive, the kernel function *sys_call()* was modified to apply Redirection of Messages as it is shown in the following source code.

```
#ifdef MSGIND
  if (function == SENDREC) {
    old_sd_e = src_dst_e; /* save original endpoint */
    src_dst = _ENDPOINT_P(src_dst_e);
    old_sd  = src_dst; /* save original process number */
    new_sd =  /* get the new process number from table */
      svrtab[caller_ptr->[p_svrtab][src_dst+NR_TASKS];
    new_sd_ptr = proc_addr(new_sd); /* get the pointer */
    /* change the original endpoint */
    src_dst_e = new_sd_ptr->p_endpoint;
  }
#else
   …. original source code of Minix ……
#endif
```

The user process will be deceived that it sends the request to the server specified in *src_dst_e* parameter, but really the request it will be sent to the server obtained from the process' *p_svrtab* servers table.

In Minix, user processes can't send any message to any other process. They can only send messages if they have the correct permissions for the destinations. Therefore, the process privileges to execute a system call are checked against the permissions to send to the standard server (***old_sd_e***) instead of the permissions of the new server to keep compatibility.

3.4. Auxiliary System Calls and Functions

Two basic auxiliary system calls were added to manage servers tables:

- *int setsti(int tabnbr, int index, int value):* The *Sets Servers Table Index* system call sets the *index*-th item of table *tabnbr* with the specified *value*.
- *int getsti(int tabnbr, int index):* The *Gets Servers Table Index* system call returns the value of the *index*-th item of table *tabnbr*.

A modified version of the *fork()* system call named *tfork()* was added to set the servers table number of the child process. The *tfork()* system call has the following C declaration:

```
pid_t tfork(int ptabnbr);
```

The parameter *ptabnbr* is the servers table number to be set for the child process.

The *tfork()* calls two auxiliary functions:

1. *sys_fork()*: This is the standard Minix function to create a new process (the child) and returns its PID.
2. *sys_setpsvrtab():* It sets the process' *p_svrtab* field to the value specified in the *ptabnbr* parameter.

The function that shows the kernel process table on system console was changed to print the ***svrtab*** field of each process. As it is shown in the following screen output, the process named **xtest** has *p_svrtab=1,* and the process **inet** has *p_svrtab=0.*

```
 -nr---svrtab--endpoint--name--- -prior-quant- -user---sys----size-rts
  48        1       71126  xtest     07/07 08/08       0      1     52K -pm
  51        0       35590  inet      03/03 04/04       5      0    900K ANY
```

An auxiliary function named *svrtab_dmp()* was added to the Information Server (IS) to dump on console screen the servers tables when the **Shift-F9** keys are pressed on the console keyboard.

The following console output shows that the servers table 1 has the value 30 for the 28th entry. Those processes that have *p_svrtab=1* that make system calls to the server with process number 28, really they will make the system calls to the server with process number 30. Those processes that have *p_svrtab≠1* will make the system calls to the server with process number 28.

```
# <PRESS Shift-F9>
index  [  0]  [  1]  [  2]  [  3]  [  4]  [  5]  [  6]  [  7]  [  8]
[ 26]    26     26     26     26     26     26     26     26     26
[ 27]    27     27     27     27     27     27     27     27     27
[ 28]    28     30     28     28     28     28     28     28     28
[ 29]    29     29     29     29     29     29     29     29     29
[ 30]    30     30     30     30     30     30     30     30     30
[ 31]    31     31     31     31     31     31     31     31     31
```

4. The *relay()* IPC Primitive

A new IPC primitive named *relay()* was added to help system programmers to implement proxy services, gateways, security reference monitor, system call interception, intrusion detection system or confinement software [5].

When a user program makes a system call to a server, the request would be redirected to an alternative server (may be a proxy or gateway) using Redirection of Messages that process the requests and return the result to the caller, or it could forward the request message (perhaps previously modified) to the original destination server using *relay()* (Figure 3). A similar technique called *trampoline function* is used by other OS, but as function relay (not message relay) that *bounce* a call to other function (hence the term *trampoline*).

Minix does not have IPC primitives that allow sending a message from a source process to a destination process through a third process (the caller).

To use *relay()* the following actors must be distinguished:

- *Source*: The process (i.e. a user process) that makes a system call to a server process (i.e. PM or FS) using the *sendrec()* primitive.
- *Destination*: The process the deals with system calls (i.e. PM or FS).
- *Caller*: The process that receives the request message from the source process through Redirection of Messages and will forward it to the destination process.

The *relay()* function has the following C declaration:

```
int relay (int src_e, int dst_e)
```

where *src_e* and *dst_e* are the source and destination endpoints.

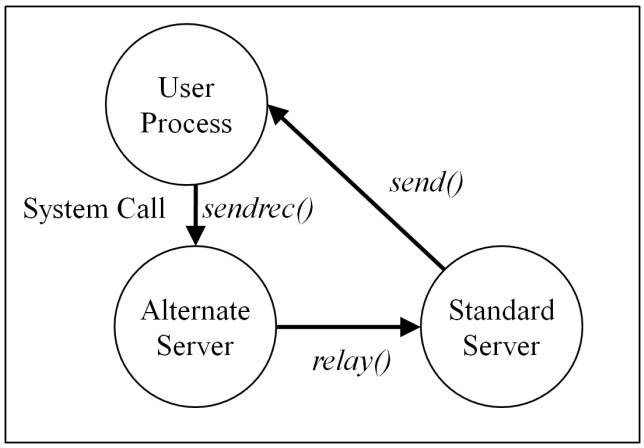

Figure 3: Sample use of *relay()*

The message itself it is not needed as an argument because it will be copied from the source message buffer to the destination message buffer.

The use of *relay()* assumes that the caller has received a request from the source process (using *sendrec()*), therefore the source process is waiting for the reply from the caller.

The kernel checks that the source process has the RECEIVING bit in the *p_rts_flags* field set to indicates that it is waiting for the reply message (line 0008), and the *p_getfrom_e* equals to the endpoint of the caller process to indicates that is waiting for the reply from it (line 0010).

```
0001 switch(function) {   /* function is the IPC code */
0002 #ifdef MSGRLY
0003 case RELAY:           /* for RELAY IPC            */
0004   src_p = _ENDPOINT_P(src_e); /* source process  */
0005   src_ptr = proc_addr(src_p); /* source endpoint */
0006   dst_p = _ENDPOINT_P(dst_e); /* dest. Process   */
0007   dst_ptr = proc_addr(dst_p); /* dest. Endpoint  */
0008   if(src_ptr->p_rts_flags != RECEIVING)
0009         return(EBADSRCDST);
0010   if(src_ptr-> p_getfrom_e !=
0011      caller_ptr->p_endpoint)
0012         return(EBADSRCDST);
0013   result = mini_relay(src_ptr, dst_ptr);
0014   break;
0015 #endif /* MSGRLY */
```

The *mini_relay()* kernel function is like *mini_send()* function that send the message to the destination process, but the caller process never blocks.

5. Conclusions and Future Works

Minix has proved to be a feasible testbench 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.

A drawback of Minix implementation is the fact that system calls are server by FS and PM. If new system calls need to be added, some kernel source code constants must be modified and the system must be recompiled.

The proposed Redirection of Messages mechanism allows that multiple servers and drivers could execute concurrently and be interpreted as different environments for processes. A user process could use the standard filesystem server, but other process could use other servers that support EXT2/3/4, FAT16/32, VFAT, NTFS, etc., or remote filesystems through a file system proxy or gateway server.

This article describes the use of Redirection of Messages only applied to user level processes and system calls, but the same approach would be applied to servers and drivers processes. New IPC primitives, like *relay()* are needed to take advantage of those facilities.

The reliability and robustness of Minix 3 would be improved with Redirection of Messages and the *relay()* IPC primitive. The primary/backup approach [6] for servers or drivers could be implemented easily. A server or driver would receive a request from a user or server process and could replicate the request to a primary server or driver and to a backup server or driver that could be local or remote.

The proposed extensions can be used to develop a variety of security related functions such as custom auditing and logging, fine grained access control, intrusion detection or confinement.

The author is working on his PhD. thesis about a *Distributed Microkernel based Operating System as a Middleware* where servers and drivers register their services and versions on different machines, making use of Redirection of Messages and the *relay()* system call to provide new facilities in the field of modern operating systems.

References

1.Tanenbaum, Woodhull. "*Operating Systems Design and Implementation, Third Edition*". Prentice-Hall, 2006.

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

3.Herder, "*Towards A True Microkernel Operating System*", master degree thesis, 2005.

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

5.K. Jain, R. Sekar; "*User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement"*; Iowa State University.

6. Budhiraja, Marzullo, Schneider, Toueg. "*The Primary-Backup Approach*". Cornell University.