MINIX 3: Local Bindery Service

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 uses message transfers paradigm to communicate user processes, servers, and device drivers. System Calls use Interprocess Communication (IPC) primitives to send messages requesting services from the Process Manager Server (PM) or the File System Server (FS), and to wait for reply messages. The request messages refer to destination processes with fixed endpoint numbers for each server. MINIX 3 kernel could use a Local Bindery Service to get the endpoint number of a desired service, therefore System Calls could be provided by processes other than FS or MM, without changes in the user's program source code. This article describes that problem, presents a solution approach, and sketches kernel source code changes as a proof of concept.

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 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.

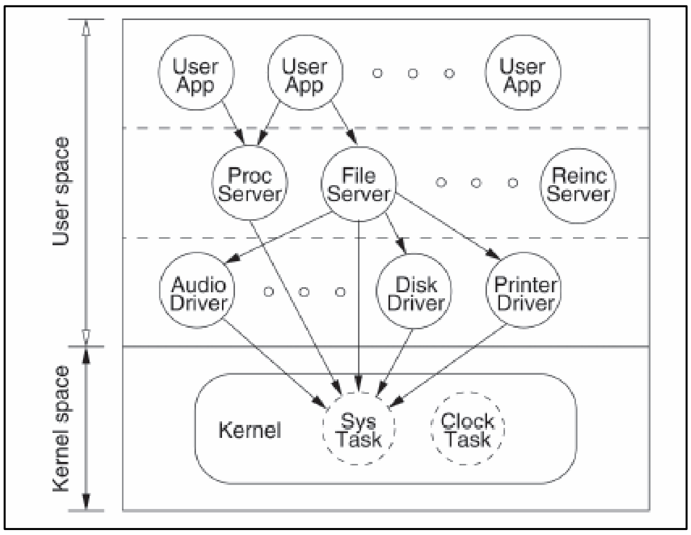

Figure 1: The Internal Structure of MINIX 3 [From [4]]

System Calls are implemented using message transfers (***sendrec***) packaging arguments and results in the same way as RPC does as it is shown in the next source code:

```
PUBLIC int _syscall(who, syscallnr, msgptr)
int who;      /* destination server i.e. PM or FS */
int syscallnr;
register message *msgptr;
{
  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);
}
```

MM (from the deprecated name Memory Manager) is defined as a constant:

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

A POSIX System Call like *getpid()* is implemented sending a GETPID request to the MM (an alias of PM) and waiting for the reply from the server as it is shown in the next source code:

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

The destination process is hard coded into the System Call as a constant restricting that POSIX and other MINIX System Calls only can be served by PM or FS Servers.

The problem was described by 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…."

Another limiting issue came up when a system programmer needs to add a new System Call. He/she must change several source files and must recompile the kernel, FS and PM servers, and the system must be restarted for the changes to work out.

The development and testing of some interesting features as, remote process execution, multiple file systems support, multiple processing environments, transparent proxy and gateway servers, a security reference monitor, System Call profiling, etc. would be simplified if a System Call message transfer would be served up by a assigned server in a dynamic way.

This article is about the use of a Bindery Server to resolve the endpoints of processes serving System Calls requests. This approach provides:

– *Flexibility*: System programmers would add new System Calls easily or replace existing ones.
– *Reliability*: A failed server would be replacing by other operational server.
– *Performance*: Multiple servers can concurrently provide the same service.
– *Dynamism*: A server can be (re)started at any time and client processes can use its services later without reboot.

The proposed approach of local System Calls binding can be extended to other operating systems offering similar properties.

The rest of this article is organized as follows. Section 2 is an overview of MINIX 3 System Calls implementation. Section 3 describes the operation of the proposed Local Bindery Service for System Calls. Section 4 refers to the auxiliary *bfork()* System Call; Section 5 sketchs a description of the Bindery Server services. Finally, Section 6 presents conclusions. As the source code is presented as a proof of concept, it must be clear that it is not a definite and refined version of MINIX 3.

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 a reply from a process.
– *notify():* a non blocking send of 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. The destinations of the request messages are the PM server or the FS server. MINIX kernel has a Process Table to keep the attributes of each process. The FS and PM have their own versions of the Process Table with fields for attributes that they need. The kernel *proc* data structure has two fields to identify a process:

```
proc_nr_t  p_nr;       /* number of this process  */
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 *endpoint* field is a combination of the *p_nr* field and the *generation* of the Process Table slot. Each slot has a *generation number* that counts the use of it. Each time a process uses the slot, the generation count is increased. This trick avoids that the process that is using the same slot of other terminated process can receive messages addressed to the dead process.

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

```
PUBLIC int sys_call(call_nr, src_dst_e, m_ptr, bit_map)
int call_nr;    /* system call number and flags */
int src_dst_e; /* src to rcve from or dst to sendto */
message *m_ptr;  /* ptr to msg in the caller's space */
long bit_map;    /* notification event set or flags */
```

The parameter *call_nr* really is the code number of an IPC primitive (the defined constants SEND, RECEIVE, SENDREC, NOTIFY, ECHO) not the code number of upper level System Calls (as OPEN, FORK, KILL, etc). The parameter *src_dst_e* is the endpoint number of the source/destination process. The other parameters are commented in the source code.

3. Local Bindery Service For System Calls

The Local Binder Service refers to resolve System Calls server endpoint numbers through a Bindery Server as RPC binder [5] does for network services.

The Local Binder Server (or BS) can be started after the system startup. At its initialization, it sets up its System Call database (named Global EndPoint Table or

GEPT) with the endpoints of PM and FS, then it waits for server registrations or client requests.

The following sequence describes the steps of System Calls dynamic bindery:

– Once a new server (i.e. GS: Generic Server) is ready to offer its services, it must register them with the BS (message 1 in Figure 2).
– If a user process invokes a System Call provided by the GS, it searches its Local EndPoint Table (LEPT) for the endpoint of that service number.
– The first time the process request that service it will not find it in its LEPT (the special endpoint number NONE is returned), therefore the process must request BS to resolve the endpoint number (message 2 in Figure 2).
– BS looks up the GEPT for the requested service. If the service and version match, BS responds to the process with GS's endpoint number (message 3 in Figure 2).
– The client process saves the endpoint of the GS into the requested service entry of its LEPT, and then invokes the System Call provided by GS (message 4 in Figure 2).

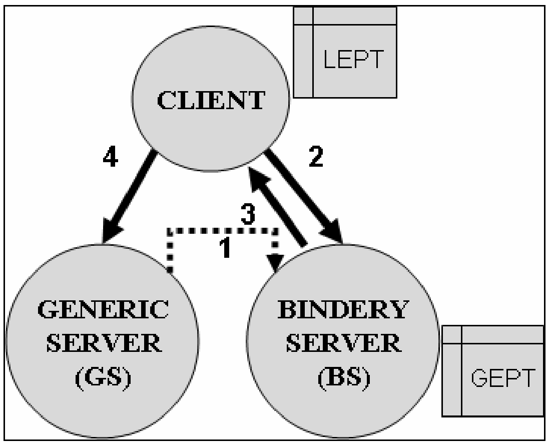

Figure 2: System Calls Dynamic Bindery

Message 2 and 3, occurs only first time of every System Call used by the process because the LEPT acts as a cache of endpoints.

3.1. Data Structures

Each process has a cache of service endpoints (LEPT). It has the following C data structure.

```
typedef int endpoint_t; /* endpoint number*/
```

There are SERVICE_VERSION numbers of possible versions for each service. The database of service endpoints in BS is the GETP. It has the following C data structure.

```
typedef int endpoint_t; /* endpoint number*/
/* The Global EndPoint Table (GEPT) entry structure*/
typedef struct {
    char   svc_name[M3_STRING];/ * service name */
    endpoint_t svr_ep[SERVICE_VERSION];/* Endpoint*/
} gepte_t;
```

The BS has an auxiliary table named *std_sct* (Standard System Call Table) with the following C data structure:

```
/* Definition of the System Call Table */
typedef struct {
     char         sc_name[M3_STRING];/* service */
     endpoint_t   sc_ep;/* Endpoint*/
     int          sc_nbr;/* syscall number */
} sctable_t;
```

The *std_sct* is statically filled with the names of the POSIX services, the endpoints of the servers (MM/PM or FS), and the System Call numbers.

```
#define POSIXCALLS 65
sctable_t std_sct[POSIXCALLS] = {
 {"_exit", MM, EXIT},
 {"access", FS, ACCESS},
……
 {"getgid" ,MM, GETGID},
….
```

A new field named *p_version* was added to the kernel's process descriptor data structure to store the System Call version number that each process will use.

```
struct proc {
…
  int p_endpoint;   /* endpoint number */
#ifdef LCLBIND
  int     p_version;/* system call version*/
#endif
…
};
```

3.2. Data Structures Initialization

The BS initializes the GEPT in two steps:

1. Resets all elements of the table with the special value NONE for service endpoint, pointing out that this entry is void.
2. Sets the entries corresponding to POSIX services with the endpoints of FS and PM in accordance with the standard MINIX System Calls extracted from the *std_sct* table. The version number is set to 0 for standard services.

```
PRIVATE void initgept()
{
  int i, j;
/* fill the GEPT with the special value NONE for all */
/* tables and versions                               */
  for (i = 0 ; i < GEPTSIZE; i++) {
    strncpy(gept[i].svc_name,"no_service",M3_STRING-1);
    for (j = 0 ; j < SERVICE_VERSION; j++)
            gept[i].svr_ep[j]  = NONE;
  }
/* fill the table for version 0 with the endpoints of*/
/* PM and FS servers for posix system calls          */
  for (i = 0 ; i < POSIXCALLS; i++) {
    j = std_sct[i].sc_nbr;
    gept[j].svr_ep[0]  = std_sct[i].sc_ep;
    strncpy(gept[j].svc_name,
               std_sct[i].sc_name,M3_STRING-1);
  }
}
```

The GEPT is initiated with the endpoints of PM and FS for version 0 and the special value NONE for the endpoints of other versions as shown in Table 1.

Table 1: Global EndPoint Table – GEPT

Service Name	Version 0	Version 1	Version 2	Version 3	….
_exit	PM endpoint	NONE	NONE	NONE	….
access	FS endpoint	NONE	NONE	NONE	….
…..	…..	…..	…..	…..	….

The Dynamic Bindery approach requires changes in System Calls Library functions therefore, all programs that will use the bindery services must be linked with the modified C library ***libc.a***.

The C run-time start-off routine *crtso* was modified to call an LEPT initialization function named *initlept()* before calling *main()* function.

```
push      eax    ! push argc
call _initlept   !INIT LOCAL ENDPOINT TABLE
 ......
call      _main  ! main(argc, argv, envp)
```

Every process linked with the modified ***libc.a*** library will has a LEPT to keep the endpoint numbers of each service and a global variable named *bind_ep* that it will store the endpoint number of BIND.

The *initlept()* function initialize the GEPT and bind_ep as follows:

1. Resets all elements of the table with the special value NONE for service endpoint, pointing out that this entry is void.
2. Request the PM for the endpoint of BS using the getprocnr() System Call and store it in bind_ep global variable.

```
endpoint_t  lept[LEPTSIZE];
int       bind_ep;
void initlept(void)
{
   int i, retcode;
   message m;
   char bindname[]="bind";
/* fill the LEPT with the special value NONE        */
   for (i = 0 ; i<LEPTSIZE; i++)
          lept[i]=NONE;
/* Request the PM for the endpoint of BS             */
   m.m1_p1 = bindname; /* name of the wanted process */
   m.m1_i1 = -1;       /* lookup by name             */
```

```
    m.m1_i2 = 5;   /* string length                     */
    if (_syscall(MM,GETPROCNR,&m) < 0)
          bind_ep=NONE;
    else
          bind_ep=m.m1_i1;
}
```

The *p_version* field is initialized on boot by the *main()* kernel function.

```
for (rp = BEG_PROC_ADDR, i = -NR_TASKS;
        rp < END_PROC_ADDR; ++rp, ++i) {
    rp->p_rts_flags = SLOT_FREE;/* init free slot */
    rp->p_nr = i;   /* proc number from ptr */
    rp->p_endpoint = _ENDPOINT(0, rp->p_nr);
    (pproc_addr + NR_TASKS)[i] = rp;
#ifdef LCLBIND
    rp->p_version = 0;
#endif
}
```

When a process forks, the *p_version* must be inherited by the child process.

```
#ifdef LCLBIND /* rpc points to child's proc struct */
           /* rpp pointes to parent's proc struct */
rpc->p_version=rpp->p_version;/*from parent to child */
#endif /* LCLBIND */
```

4. The *bfork()* System Call

An alternative version of the *fork()* System Call named *bfork()* was created to set the *p_version* field of a child process. The *bfork()* System Call has the following format:

```
              pid_t bfork(int version)
```

The parameter *version* is the value to store in the *p_version* field of the kernel *proc* data structure of the child process.

The *bfork()* System Call invokes:

1. *sys_fork():* to create the new (child) process and to get its PID.
2. *sys_setpver():* A new Kernel Call that sets the *p_version* field of the process to the value specified in the *version* parameter.

4.1. System Task Services

Two new Kernel Calls was added to System Task process (named SYSTASK that it is a process that share address space with the kernel):

- *Set Process System Call Version (SETPVER)*: Set the version number that the process will use on each System Call. The function *do_setpver()* of SYSTASK is used by *bfork()* to set the System Call version number of a process descriptor.
- *Get Process System Call Version (GETPVER)*: Get the version number that the process will use on each System Call. The function *do_getpver()* of SYSTASK is

used by the BS to get the System Call version number of a process descriptor, and then search the GEPT for that version.

4.2. System Calls Library Changes

To use Local Dynamic Binding, all programs must be linked with the modified ***libc.a*** library. The modified source code of the ***getuid()*** function is presented as an example:

1. The process checks that the BS is running. If it is not running, makes the System Call to PM.
2. The process checks for the desired service endpoint number in the LEPT. If it is not in the LEPT, makes a SVCBIND (Service Bind) request to the BS; then saves the replied endpoint into the LEPT for future uses.
3. The process sends a request message to the resolved endpoint as a destination for the GETPID service.

```
extern endpoint_t lept[LEPTSIZE];
extern int bind_ep;
PUBLIC pid_t getpid()
{
  message m;
#ifdef  LCLBIND
  int server_ep, retcode;
  if (bind_ep == NONE) /* is BS running?            */
        return( (pid_t)_syscall(MM, GETPID, &m));
  if ((server_ep = lept[GETPID])== NONE){
        m.m3_i1 = GETPID;
        retcode = _syscall(bind_ep,SVCBIND,&m);
        server_ep = m.m3_i1;
        if (retcode !=0 )
            return( (pid_t)_syscall(MM, GETPID, &m));
        lept[GETPID] = server_ep;
  }
  return( (pid_t)_syscall(server_ep,GETPID,&m));
#else
  return(_syscall(MM, GETPID, &m));
#endif
```

5. Bindery Server Services

The MINIX 3 DataStore (DS) server allows components to back up data and retrieve it after a restart. One of the 5 types of data that are supported by the DS is LABEL that it builds a one-to-one mapping between a name and a number. It could be used as a Bindery Server that maps the service name to the service endpoint number, but its functionality is very limited because it does not support multiple versions and other required services.

The proposed BS offers four services.

1. *Service Set:* It is used by GSs to register their services. They request BS to set specific versions of System Calls that will be served by them.
2. *Service Reset:* It is used by GSs to deregister their services. They request BS to reset specific versions of System Calls served by them.
3. *Service Bind:* It is used by programs linked with the modified ***libc.a*** library using Dynamic Binding. The process requests the endpoint number of a service to the BS that searches for the endpoint number and the version of the requester's process descriptor, returning the endpoint of the GS to the process.
4. *Get Global Endpoint Table*: It can be used by any process that needs a copy of the BS's GEPT. It is used by the modified Information Server (IS) to dump the GETP on the console screen after pressing Shift-F9 key combination.

6. Conclusions

MINIX 3 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, but 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 constants, and FS and PM System Call tables need to be changed and those servers must be recompiled and the system restarted.

The major advantage of Local Bindery Services is that new services can be implemented without kernel/servers recompilation and system restarting, speeding up the development process.

With the use of Local Bindery MINIX could have multiple memory management processes (other than the PM) that could manage memory areas with different policies, some of them using the Buddy algorithm, other using Best Fit, etc.

The Local Bindery Services are not limited to System Calls, they can be used for Task Calls or Kernel Calls offering a wide range of possibilities to developers and would be extended for networking and distributed services.

Similar approaches of the proposed Local Bindery Services can be included in other microkernel or monolitic operating systems as LINUX offering flexibility, reliability and performance enhancement.

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.RFC1833: *Binding Protocols for ONC RPC Version 2*. http://www.faqs.org/rfcs/rfc1833.html