Operating Systems
Project Report(Part Two)

Team Number: 19
Team Members:
Raghavendra G (1PI13CS113)
Sharath KP (1PI13CS141)
Shashwath.S.Bharadwaj (1PI13CS144)
Shreyas G (1PI13CS153)

System Call Implementation:

The system calls we selected for part 2 of this project were:

1. getsysinfo() and
2. reboot()

Both the calls are implemented at the kernel level in the directory usr/src/servers/pm in the file misc.c. We describe how both these calls can be made by the user(as a system call) in this document along with a brief explanation of how each of these calls actually work.

Note: These calls aren't intended for use by the ordinary user. Therefore these are implemented as Kernel calls i.e they cannot be directly accessed by the user. However, because of the layered architecture of minix, system calls are available, which can be made to execute these kernel calls.

1. getsysinfo():

A system call to getsysinfo() function can be made as shown below:

```
struct kinfo pinf;
int num_procs;
getsysinfo(PM_PROC_NR, SI_KINFO, &pinf);
num_procs = pinf.nr_procs;
```

The above system call retrieves the number of currently running processes in the system and stores it in the variable num_procs in the user level. The call getsysinfo() takes the following parameters, in order.(in accordance with its prototype defined in the header file include/unistd.h):

```
endpoint_t  who // from whom to request info
int  what // what information is requested
void  *where // where to put the information retrieved
size_t  size // how much data is to be retrieved
```

The prototype of this call is as shown:

_PROTOTYPE(int getsysinfo, (int who, int what, void *where));

Note: Passing the fourth argument, 'size' is optional.

- In the call made, the parameter 'PM_PROC_NR', a macro defined in the header file 'com.h', indicates that the request to perform the specified operation is to be made to the Process Manager(PM) server.

- The system call is transformed into a SYS_CALL request message that is handled in a function called do_call()

- Once the system is initialised the PM enters its main loop in which it waits for a message from the user, carries out the request contained in it and sends back a reply to the user.
- When a message comes into the PM, its main loop extracts the message type and puts the value in a global variable ‘call_nr’(defined in kernel/system.c), which is used to index the call_vec table declared in the file serves/pm/table.c. In this case the value of call_nr is 79(corresponding to getsysinfo) as defined in the header file include/minix/callnr.h. The value 79 is then used to index the call_vec table in table.c, which contains a pointer to the procedure ‘do_getsysinfo’, defined in the file servers/pm/misc.c. The prototype for this function can be found in the header file servers/pm/proto.h.
- The procedure do_getsysinfo is then executed at the kernel level. The value returned by the function, depends on the type of information requested and consequently the code for the function handles multiple cases for different types of requested data. In the example we’ve chosen, the data requested is the number of processes running currently in the system which is kernel information. It call be retrieved by further making a call to sys_getkinfo() function, by passing a variable of type struct kinfo as a parameter to it.
- The content returned by the kernel call(a kernel structure) is stored into a pointer to a structure of type struct kinfo(pinf). Using the attribute ‘nr_pro’ of the structure, the number of currently running process can be determined. In this case we store the value into a variable ‘num_procs’ for further use by the user.
- Thus, the required information is retrieved by performing a kernel call via the PM server.

2.reboot():

reboot() system call can be used to close down the system by terminating all the currently executing processes in a controlled manner. The prototype of reboot() is as shown :

```
_PROTOTYPE( int reboot, (int _how, ...) );
```

It takes one argument, which indicates 'how' the reboot is to be implemented(what set of operations is to be performed) in the kernel call. The parameter can take one of several values:

RBT_REBOOT
RBT_HALT
RBT_RESET
RBT_PANIC and so on.. each of which is defined in the file include/unistd.h and corresponds to a different mode or method of rebooting.

reboot() may only be executed by the super-user i.e a user who has all rights or permissions in all modes(typically root).

The code flow and execution of the reboot() system call is very similar to that of the getsysinfo() system call. Since the prototypes and definitions of these calls are located in the same file within the same directory, the process of making a system call to access these kernel calls remains the same. i.e the following steps remain common for both getsysinfo() and reboot():

- The system call is transformed into a SYS_CALL request message that is handled in a function called do_call()
- Once the system is initialised the PM enters its main loop in which it waits for a message from the user, carries out the request contained in it and sends back a reply to the user.
- When a message comes into the PM, its main loop extracts the message type and puts the value in a global variable 'call_nr'(defined in kernel/system.c), which is used to index the call_vec table declared in the file serves/pm/table.c. In this case the value of call_nr is 76(corresponding to reboot) as defined in the header file include/minix/callnr.h. The value 76 is then used to index the call_vec table in table.c, which contains a pointer to the procedure 'do_reboot', defined in the file servers/pm/misc.c. The prototype for this function can be found in the header file servers/pm/proto.h.

Note: The value of call_nr is 76 for reboot() as defined in the header file. This value was 79 for getsysinfo()

- The procedure do_reboot is then executed at the kernel level. The exact set of operations performed depends on the parameter 'how', which is passed to the procedure. Consequently, the code for do_reboot() handles the different cases corresponding to different modes of rebooting(RBT_PANIC,RBT_HALT,RBT_RESET etc) as specified by the user.
- A significant difference between the execution of reboot() and getsysinfo() system calls is that reboot() has to send signals to terminate all processes in a controlled manner as

it's first step i.e it has to terminate all running processes before proceeding with the reboot functionality.

- For this purpose, the do_reboot() procedure calls a function check_sig()(defined in servers/pm/signal.c) with the parameters -1 and SIGKILL passed to it as shown:

  check_sig(-1, SIGKILL);

  This call kills all currently executing processes except init.
- The procedure then signals the File System server to prepare for shutdown by calling the following function:

  tell_fs(REBOOT,0,0,0);
- Finally the procedure asks the kernel to abort by making a call to sys_abort function as shown:

  sys_abort(abort_flag, PM_PROC_NR, monitor_code, code_len);

  This causes all system services including the PM to get a notification to abort.
- If the call suceeds,it never returns.If an error occurs,the return value is -1.An error is indicated by errno.
- Thus, the system is rebooted by performing a system call via the PM server.