Group 11 - Report 2

Introduction to Operating System (IOS)

01FB15ECS213 Praveen C Naik  
 01FB15ECS202 Parashara Ramesh

01FB15ECS209 Prajwal B

01FB15ECS224 Rahul Pillai

Minix Version : 3.2.1  
http://www.minix3.org

System Calls  
**1) Getsysinfo()**

Defined as a function prototype in:

|  |
| --- |
| line 8 : minix/include/minix/sysinfo.h |

Defined as a function in:

|  |
| --- |
| **line 8 : minix/lib/libsys/getsysinfo.c** |

**Reference links** :

<https://blog.heabuh.com/jekyll/update/2016/11/29/minix-getsysinfo.html>

**The code for getsysinfo:**

|  |
| --- |
| int getsysinfo(  endpoint\_t who, // from whom to request info  int what, // what information is requested  void \*where, // where to put it  size\_t size // how big it should be  )  {  message m;  m.SI\_WHAT = what;  m.SI\_WHERE = where;  m.SI\_SIZE = size;  if (\_syscall(who, COMMON\_GETSYSINFO, &m) < 0) return(-1);  return(0);  } |

**Code flow execution**

1.firstly we create a test file in the “servers” layer and call the function in this way

|  |
| --- |
| getsysinfo(PM\_PROC\_NR, SI\_PROC\_TAB, &mproc,sizeof(mproc)); |

Where, **“PM\_PROC\_NR”** is the “**endpoint\_t who”** that is from whom to request the info, as we want the PM to handle this.

**“SI\_PROC\_TAB”** is the **“what”** parameter ie this is the copy of the entire process table which is to be copied into the 3rd argument **“mproc”**.

Other values the “what” parameter could take include :

|  |
| --- |
| /\* What system info to retrieve with sysgetinfo(). \*/ #define SI\_PROC\_TAB 2 // copy of entire process table #define SI\_DMAP\_TAB 3 // get device <-> driver mappings #define SI\_DATA\_STORE 5 // get copy of data store mappings #define SI\_CALL\_STATS 9 // system call statistics #define SI\_PROCPUB\_TAB 11 // copy of public entries of process table #define SI\_VMNT\_TAB 12 // get vmnt table |

**2. But when we look at the getsysinfo code we see that it calls** \_syscall(who, COMMON\_GETSYSINFO, &m) < 0 (where #define COMMON\_GETSYSINFO (COMMON\_RQ\_BASE+2) is defined in com.h) .

Which sends a Common request/signal to several system servers for retrieving system information.

3.in this case this calls the do\_getsysinfo() in servers/pm/misc.c .since each server has its own do\_getsysinfo() we are only interested in the do\_getsysinfo() present in the pm server as we have passed PM\_PROC\_NR in the getsysinfo() function as the endpoint.

4. In the do\_getsysinfo() of servers/pm/misc.c there is a switch case for “SI\_PROC\_TAB” in which we copy the entire process table into the struct mproc using the sys\_datacopy() function

5.If an unauthorized end point is calling this function then the do\_getsysinfo() prints an error message.

**Observations/Insights**

1. There are two getsysinfo(), one for the kernel(it’s called sys\_getinfo) and the other one as only a system call to the server layer and doesn’t go to the kernel.

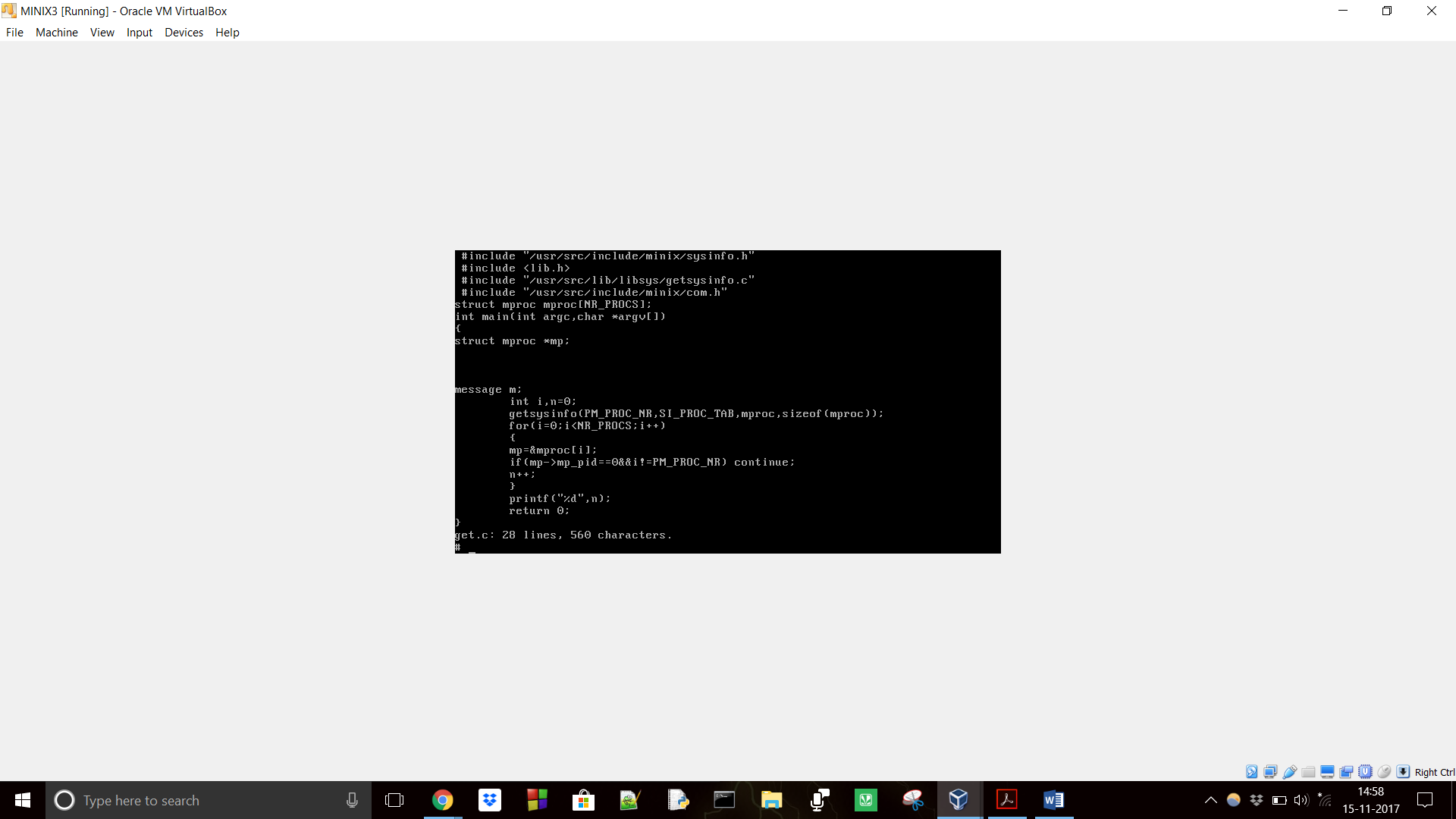
2.There is a do\_getsysinfo implemented in all the system servers,thus to identify which server it should go to , a common signal is send to all the servers,Based on the 1st argument in \_syscall it identifies which server has to handle that call.

3.There are only 6 types of information that can be asked from getsysinfo() .They are defined in minix/sysinfo.h .

4. Permissions are implemented for getsysinfo in do\_getsysinfo by checking the effuid of the process,and if denied prints unauthorized call.

5.sysinfo() although simple is a very important call in minix as many other system calls depend on it.

**Sample code and output:**



This program tries to print the no of processes in the process table .

**OUTPUT:**

39

**2) Alarm**

## **Code flow for alarm system call**

* alarm is called using alarm() system call: unsigned int alarm(unsigned int seconds)
* we ran a test program with an infinite loop , and an alarm of 3 seconds set up before the loop.The program was killed exactly after 3 seconds.
* man alarm says it is part of the unistd library, thus has its prototype in unistd.h The description says it cause a SIGALARM to the invoking process in a number of seconds given in the argument.This signal terminates the process by default unless caught or ignored.
* The return value is the amount of time previously remaining in the alarm clock.
* The library function is defined in /usr/src/lib/libc/gen/alarm.c
* Two struct variables of type itimerval is declared.itimerval structure has two members for seconds and microseconds.
* timerclear() is called to clear the previous time interval,The parameters are set and then passed to setitimer .ITIMER\_REAL means we are setting up a real timer based on real time.If this call results in an error -1 is returned.
* setitimer is defined in /usr/src/lib/libc/sys-minix/setitimer.c.This function sets the interval timer.In this a system call to do\_itimer() through the mapping of ITIMER(#define ITIMER 64) in callnr.h is performed with the message parameters set.do\_itimer has its prototype in proto.h.
* the do\_itimer call is defined in the server layer in pm server.There it performs some validity checks after the setval and getval are set.These variables are set based on the new\_val and old\_val parameters.following that is a switch case which has different cases for real,virtual and profile timers.Since we want a real timer the first case label is executed.If the old timer value is set then it calls get\_realtimer or if we wan to set a timer then set\_realtimer is called.
* get\_realtimer is called to know the remaining amount of time to expiry time(which is the current value),the interval of the timer is also stored in the "value" structure.
* set\_realtimer call set\_alarm with the number of tiicks for the alarm and the structure pointer "mproc" which gives information about the invoking process.
* set\_alarm in turn calls set\_timer which sets the timer with the arguments ticks,cause\_sigalarm and endpoint.The cause\_sigalrm is a watchdog function checks the number of ticks and sends the signal SIGALARM to the process when the ticks left are 0.
* set\_timer sets the previous time to 0 ,sets the timer argument and adds the timer to a list.The list has timers waiting to be started.It calls sys\_setalarm for the appropriate timer in the list which leads to a kernel call which sets the alarm.
* cancel\_timer clears the present timer , and then sets the next timer in the queue by calling the sys\_setalarm call.It cancels the alarm if the last timer was killed.
* cause\_sigalarm has a callback to keep checking if the timer ended so as to cause the SIGALARM signal by calling the check\_sig function defined in signal.c wgich in turn does a kernel call.
* sys\_setalarm is defined in libsys/sys\_setalarm.c .This asks the kernel to schedule a synchronous alarm for the caller, int sys\_setalarm(clock\_t exp\_time, int abs\_time, clock\_t \* time\_left,clock\_t \* uptime)This calls a kernel call with the handler SYS\_SETALARM.From the mapping in com.h , it executes the kernel function do\_setalarm in do\_setalarm.c in kernel/system.

**Observations/Insights**

1.In alarm we can set three possible types of timers . There is real-time , virtual and profiling timer which we can use based on the context.

2.There is a callback used in the watchdog function (cause\_sigalarm) which call set\_alarm will call set\_timer to set a new timer.

3.in set\_alarm when a previous alarm is still on and the timer is set , it cancels the previous alarm’s timer by calling cancel\_timer , thus setting a new alarm will remove the previous one.

4.The comments at the start of the file alarm.c says that large delays passed to alarm call can cause problems because of the type unsigned int in argument ,it can return uncalled for errors and there can be overflow when seconds are converted to ticks.Thus the alarm system call still has issues.

5. In set\_timer , we can see that there can be a list of timers and alarms can be rescheduled .

**Reference links : none(just minix code)**

**3) Read**

## **Working of read system call**

n = read(fd, buffer, nbytes);

* The system call is passed with three parameters.Fs receives a message containing these parameters and a code for read as the message type , after which it sends a reply.
* The message type is used as an index in a table to call the right procedure for reading.
* This procedure extracts the file descriptor from the message and uses it to locate the filp entry and then the i-node for the file to be read. The request is then broken up into pieces such that each piece fits within a block. For example, if the current file position is 600 and 1024 bytes have been requested, the request is split into two parts, for 600 to 1023, and for 1024 to 1623 (assuming 1-KB blocks).
* When the reply comes back to the user, the library function read extracts the reply code and returns it as the function value to the caller.

Character special files are not read through the block cache.

The real read mechanism starts for ordinary files atleast from the below line:

This req\_readwrite function breaks the request up into chunks, each of which fits in a single disk block. A chunk begins at the current position and extends until one of the following conditions is met:

1. All the bytes have been read  
 2. A block boundary is encountered  
 3. The end of file is hit

These rules mean that a chunk never requires two disk blocks to satisfy it.The actual calculation is done in the function cpf\_grant\_magic defined in libsys/safecopies.c

* The actual reading of the chunk is done by the req\_readwrite itself. When control returns, various counters and pointers are incremented, and the next iteration begins. When the loop terminates, the fileposition and other variables may be updated (e.g., pipe pointers).
* Read-ahead defined defined in mfs/main.c in the main loop brings the next block after it sends the reply for the 1st block to the user.The user can work on the data from the 1st block while the next block is being fetched.This improves perfomance.It knows where the next block begins from the global variables which has the next positions and the inode info.
* takes the inode and file position , converts them to a physical disk block number,requests the transfer of that block to user space.The mapping of file position to physical disk address is done in safecopies.c when we called cpf\_grant\_magic in req\_readwrite as it does a kernel call to grant that block so it can be used in the user space

**Code flow Execution**:

1.)After the system call is called, do\_read() common for both reading and writing is called from the file system(vfs) ,it calls the do\_read\_write() function with the read-write flag set to READING .

2.)do\_read\_write sets the members of the file descriptor using the scratch function whose scratchpad structure defined in scratchpad.h , does some validity checks(ex:-t checks whether the user process has the memory that it needs in order to perform the required transfer) on the file descriptor and passes it to the read\_write( ) function along with other parameters like size , buffer pointer and endpoint.

3) read\_write() extracts the file descriptor from the message and uses it to locate the filp entry and then the i-node for the file to be read.It checks the mode of the file using vp->vmode to identify the type of file constant(ex-: pipe,character special,block special).Character special files are not read through the block cache so it is separately read from the device i/o buffer.Block special files are also handled with different code which requires calling req\_breadwrite function defined in request.c.The real mechanism of read for regular files start from line 163.It issues a request to req\_readwrite() function with the present file position , buffer,size,etc,.

4)req\_readwrite is called from the request.c files which handles request,there the request is split into chunks such that each chunk fits into a block which can read(explained in the theory in the start of the report).This splitting of chunks happens via the kernel by calling cpf\_grant\_magic defined in libsys/safecopies.c ,here all required blocks are granted to user space after the kernel call.If the req\_readwrite happens successfully then in read\_write the current position is set to new position

5)The reply value is returned to the caller.The return value is the number of bytes actually read .A value of zero indicates end-of-file (except if the value of the size argument is also zero).

**Observations/Insights**

# read\_ahead is not called as a part of the read system call, it is called from the main loop in *main.c servers/fs/main.c*

# read\_ahead is performed after the reply is sent to the user so that user can run while the file system waits , so this improves perfomance.

1. Read performs the splitting of requests into chunks which makes reading efficient compared to other system calls ,there is not much significant overhead.
2. Read handles all possible types of files like character special,block special,pipe,etc making it versatile .
3. There is also a lot of validity checks in each stage of read which makes it very secure to use and gives proper error messages.

**Reference links :** [**www.minix3.org/theses/gerofi-minix-vfs.pdf**](http://www.minix3.org/theses/gerofi-minix-vfs.pdf)