**README\_MISSING\_SYSCALL\_OR\_IOCTL**

        
Dealing with missing system call or ioctl wrappers in Valgrind  
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
You're probably reading this because Valgrind bombed out whilst  
running your program, and advised you to read this file.  The good  
news is that, in general, it's easy to write the missing syscall or  
ioctl wrappers you need, so that you can continue your debugging.  If  
you send the resulting patches to me, then you'll be doing a favour to  
all future Valgrind users too.  
  
Note that an "ioctl" is just a special kind of system call, really; so  
there's not a lot of need to distinguish them (at least conceptually)  
in the discussion that follows.  
  
All this machinery is in coregrind/m\_syswrap.  
  
  
What are syscall/ioctl wrappers?  What do they do?  
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
Valgrind does what it does, in part, by keeping track of everything your  
program does.  When a system call happens, for example a request to read  
part of a file, control passes to the Linux kernel, which fulfills the  
request, and returns control to your program.  The problem is that the  
kernel will often change the status of some part of your program's memory  
as a result, and tools (instrumentation plug-ins) may need to know about  
this.  
  
Syscall and ioctl wrappers have two jobs:   
  
1. Tell a tool what's about to happen, before the syscall takes place.  A  
   tool could perform checks beforehand, eg. if memory about to be written  
   is actually writeable.  This part is useful, but not strictly  
   essential.  
  
2. Tell a tool what just happened, after a syscall takes place.  This is  
   so it can update its view of the program's state, eg. that memory has  
   just been written to.  This step is essential.  
  
The "happenings" mostly involve reading/writing of memory.  
  
So, let's look at an example of a wrapper for a system call which  
should be familiar to many Unix programmers.  
  
  
The syscall wrapper for time()  
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
The wrapper for the time system call looks like this:  
  
  PRE(sys\_time)  
  {  
     /\* time\_t time(time\_t \*t); \*/  
     PRINT("sys\_time ( %p )",ARG1);  
     PRE\_REG\_READ1(long, "time", int \*, t);  
     if (ARG1 != 0) {  
        PRE\_MEM\_WRITE( "time(t)", ARG1, sizeof(vki\_time\_t) );  
     }  
  }  
  
  POST(sys\_time)  
  {    
     if (ARG1 != 0) {  
        POST\_MEM\_WRITE( ARG1, sizeof(vki\_time\_t) );  
     }  
  }  
  
The first thing we do happens before the syscall occurs, in the PRE() function.  
The PRE() function typically starts with invoking to the PRINT() macro. This  
PRINT() macro implements support for the --trace-syscalls command line option.  
Next, the tool is told the return type of the syscall, that the syscall has  
one argument, the type of the syscall argument and that the argument is being  
read from a register:  
  
     PRE\_REG\_READ1(long, "time", int \*, t);  
  
Next, if a non-NULL buffer is passed in as the argument, tell the tool that the  
buffer is about to be written to:  
  
     if (ARG1 != 0) {  
        PRE\_MEM\_WRITE( "time", ARG1, sizeof(vki\_time\_t) );  
     }  
  
Finally, the really important bit, after the syscall occurs, in the POST()  
function:  if, and only if, the system call was successful, tell the tool that  
the memory was written:  
  
     if (ARG1 != 0) {  
        POST\_MEM\_WRITE( ARG1, sizeof(vki\_time\_t) );  
     }  
  
The POST() function won't be called if the syscall failed, so you  
don't need to worry about checking that in the POST() function.  
(Note: this is sometimes a bug; some syscalls do return results when  
they "fail" - for example, nanosleep returns the amount of unslept  
time if interrupted. TODO: add another per-syscall flag for this  
case.)  
  
Note that we use the type 'vki\_time\_t'.  This is a copy of the kernel  
type, with 'vki\_' prefixed.  Our copies of such types are kept in the  
appropriate vki\*.h file(s).  We don't include kernel headers or glibc headers  
directly.  
  
  
Writing your own syscall wrappers (see below for ioctl wrappers)  
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
If Valgrind tells you that system call NNN is unimplemented, do the   
following:  
  
1.  Find out the name of the system call:  
  
       grep NNN /usr/include/asm/unistd\*.h  
  
    This should tell you something like  \_\_NR\_mysyscallname.  
    Copy this entry to include/vki/vki-scnums-$(VG\_PLATFORM).h.  
  
  
2.  Do 'man 2 mysyscallname' to get some idea of what the syscall  
    does.  Note that the actual kernel interface can differ from this,  
    so you might also want to check a version of the Linux kernel  
    source.  
  
    NOTE: any syscall which has something to do with signals or  
    threads is probably "special", and needs more careful handling.  
    Post something to valgrind-developers if you aren't sure.  
  
  
3.  Add a case to the already-huge collection of wrappers in   
    the coregrind/m\_syswrap/syswrap-\*.c files.   
    For each in-memory parameter which is read or written by  
    the syscall, do one of  
      
      PRE\_MEM\_READ( ... )  
      PRE\_MEM\_RASCIIZ( ... )   
      PRE\_MEM\_WRITE( ... )   
        
    for  that parameter.  Then do the syscall.  Then, if the syscall  
    succeeds, issue suitable POST\_MEM\_WRITE( ... ) calls.  
    (There's no need for POST\_MEM\_READ calls.)  
  
    Also, add it to the syscall\_table[] array; use one of GENX\_, GENXY  
    LINX\_, LINXY, PLAX\_, PLAXY.  
    GEN\* for generic syscalls (in syswrap-generic.c), LIN\* for linux  
    specific ones (in syswrap-linux.c) and PLA\* for the platform  
    dependent ones (in syswrap-$(PLATFORM)-linux.c).  
    The \*XY variant if it requires a PRE() and POST() function, and  
    the \*X\_ variant if it only requires a PRE()  
    function.    
      
    If you find this difficult, read the wrappers for other syscalls  
    for ideas.  A good tip is to look for the wrapper for a syscall  
    which has a similar behaviour to yours, and use it as a   
    starting point.  
  
    If you need structure definitions and/or constants for your syscall,  
    copy them from the kernel headers into include/vki.h and co., with  
    the appropriate vki\_\*/VKI\_\* name mangling.  Don't #include any  
    kernel headers.  And certainly don't #include any glibc headers.  
  
    Test it.  
  
    Note that a common error is to call POST\_MEM\_WRITE( ... )  
    with 0 (NULL) as the first (address) argument.  This usually means  
    your logic is slightly inadequate.  It's a sufficiently common bug  
    that there's a built-in check for it, and you'll get a "probably  
    sanity check failure" for the syscall wrapper you just made, if this  
    is the case.  
  
  
4.  Once happy, send us the patch.  Pretty please.  
  
  
  
  
Writing your own ioctl wrappers  
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
  
Is pretty much the same as writing syscall wrappers, except that all  
the action happens within PRE(ioctl) and POST(ioctl).  
  
There's a default case, sometimes it isn't correct and you have to write a  
more specific case to get the right behaviour.  
  
As above, please create a bug report and attach the patch as described  
on http://www.valgrind.org.  
  
  
Writing your own door call wrappers (Solaris only)  
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
  
Unlike syscalls or ioctls, door calls transfer data between two userspace  
programs, albeit through a kernel interface. Programs may use completely  
proprietary semantics in the data buffers passed between them.  
Therefore it may not be possible to capture these semantics within  
a Valgrind door call or door return wrapper.  
  
Nevertheless, for system or well-known door services it would be beneficial  
to have a door call and a door return wrapper. Writing such wrapper is pretty  
much the same as writing ioctl wrappers. Please take a few moments to study  
the following picture depicting how a door client and a door server interact  
through the kernel interface in a typical scenario:  
  
  
door client thread          kernel       door server thread  
invokes door\_call()                     invokes door\_return()  
-------------------------------------------------------------------  
                               <----  PRE(sys\_door, DOOR\_RETURN)  
PRE(sys\_door, DOOR\_CALL)  --->  
                               ---->  POST(sys\_door, DOOR\_RETURN)  
                                           ----> server\_procedure()  
                                           <----  
                               <----  PRE(sys\_door, DOOR\_RETURN)  
POST(sys\_door, DOOR\_CALL) <---  
  
The first PRE(sys\_door, DOOR\_RETURN) is invoked with data\_ptr=NULL  
and data\_size=0. That's because it has not received any data from  
a door call, yet.  
  
Semantics are described by the following functions  
in coregring/m\_syswrap/syswrap-solaris.c module:  
o For a door call wrapper the following attributes of 'params' argument:  
  - data\_ptr (and associated data\_size) as input buffer (request);  
      described in door\_call\_pre\_mem\_params\_data()  
  - rbuf (and associated rsize) as output buffer (response);  
      described in door\_call\_post\_mem\_params\_rbuf()  
o For a door return wrapper the following parameters:  
  - data\_ptr (and associated data\_size) as input buffer (request);  
      described in door\_return\_post\_mem\_data()  
  - data\_ptr (and associated data\_size) as output buffer (response);  
      described in door\_return\_pre\_mem\_data()  
  
There's a default case which may not be correct and you have to write a  
more specific case to get the right behaviour. Unless Valgrind's option  
'--sim-hints=lax-doors' is specified, the default case also spits a warning.