Excercise 1 - Debugging syscalls

In this session, I created the Ubuntu container for my experiments. It is worth noting that I used a different command to the one in Lab1.pdf for this experiment; although the results were the same, the docker command in Lab1.pdf did not enter the container because the -rm parameter was present, whereas I entered the container and used the container terminal for the experiment.

Although not related to the system call command, I would like to explain about the docker command I used during my use, this was my first time using docker and I found it useful!

The first command for docker is docker build -t lab1 ex . This command essentially creates an image and names it lab1_ex1 . But why does it start? I noticed that there was a Dockerfile in the lab1/ex1 folder with the command FROM ubuntu, i.e. the image was from Ubuntu; this was followed by the RUN apt-get update && apt-get install -y strace build-essential iputils- ping command, this command is mainly meant to run the Linux command that follows RUN after creating the image, by updating the Ubuntu system and installing the strace command so that you can use the strace command directly after the container is created; while I first find the image I need by using the docker pull command, and then use the docker run The -it -name=xxx ubuntu command creates the container and enters it, then executes the command to update and install strace. docker run means to create a container, -it means to run in a pseudo terminal

and the container will stop when you exit it, and the -t parameter allows you to continue working in the background after you exit the container.

In the process of pulling the Ubuntu image I found that the size of the image was very much smaller than the average Ubuntu system size, only 72.8MB! Why does this happen? Because the image technology used by docker uses overlay technology, because ubuntu is a unix-like system and macOS is also a unix-like system in many ways can be reused without re-downloading, the ubuntu image only needs to download a part of the host system that cannot be reused, and similarly mysql has no system reuse so the file size is about the same.

After executing the docker run --rm -ti lab1_ex1 strace echo "Hello World" command a number of system calls are displayed, which are then explained one by one.

execve("/usr/bin/echo", ["echo", "hello world"], 0x7fffa059f378 /* 16 vars */) = 0

This command corresponds to the execve system call command, which is one of the most important commands on the system, it enables a process being called to become a new process, this command makes it possible for a program to become a process, the first argument to this command is the file directory, because I used the echo command, and in Linux the echo execution command file exists in /usr/bin The second argument to the function is the argument needed to pass into the new process created by the original file, at least one argument needs to be passed, in this case echo "hello world"

is passed as the first and second argument into the new process In this case the third parameter is the environment variable required by the new process, in this case it shows that 16 environment variables are passed, and in the environ() function we can see exactly what variables there are, by assigning a value to each parameter with ``name=value'', the corresponding hexadecimal number is the corresponding setting number. Deeper down, the essence of execve() is that the system call int 0x80 triggers a soft interrupt to enter the system kernel, and after entering the system kernel the do_execve() function is called to perform more operations, in this stage of do_execve the parameters are encapsulated and then the related functions will oh parse the ELF file and load the file into memory and modify the user Finally the process returns from execve to the user state, which already contains the parameters and the relevant environment, and is safe to execute the new program.

brk(NULL)=0x55f52d2c3000

This command modifies the heap size directly, where brk modifies NULL so the process's memory is not allocated by the brk function, but during the call to the brk function the brk function compares the target to be allocated with the brk process in the memory description and then makes the decision to expand and reduce the heap to fit the process's required memory capacity.

arch_prctl(0x3001 /* ARCH_??? */, 0x7ffc5076ff80) = -1 EINVAL (Invalid argument)

This system call function is mainly used to set the state of a specific process or thread

of the architecture, this function is able to set the FS and GS registers through the corresponding sub-functions ARCH_SET_FS, ARCH_GET_FS, ARCH_SET_GS, ARCH_GET_GS respectively. state was not set.

access ("/etc/ld.so.preload", R_OK) = -1 ENOENT

This function is to check the viability of the file, there are four sub-functions in the access function that correspond to the four modes of detecting the file, which are whether it is readable, writable, executable and exists, in my system call example it detects whether the file it wants to find exists, but it returns -1 which means it does not exist, and the file it is looking for is /etc/ld.so.preload, and This is an environment variable that is used to preload some shared libraries or target files, because of the high priority, so it is used to install some of the more important libraries, in my case because I am only using it to test that my libraries have not been installed yet, so loading the shared libraries fails, for the operating system the preload file it specifies will be loaded whether or not the program needs the libraries or files.

openat(AT_FDCWD,"/etc/ld.so.cache",O_RDONLY|O_CLOE EC) = 3

This system call accesses the current file path with the first argument being AT_FDCWD. The final effect is equivalent to the open() function, in which case openat opens the dynamic library file located in the /etc path, as above the system gives priority to fetching the shared or dynamic library information, but ld.so.preload has higher priority. In the case of my system call it reads the relevant dynamic library and

opens it in read-only mode.

Deeper down, a lot of things happen during the use of the opennat() system call. First, a system interrupt is made, after which the system enters the kernel state and then the corresponding system_open function is called with the appropriate arguments. In the kernel state, the do_sys_openh system function looks for an unused fd, then allocates a file structure to the file and stores the appropriate file information, then creates the appropriate dentry, and finally establishes the connection between the fd and the structure.

read(3,"\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0360q\2\0\0\0\0\0 "..., 832) = 832

Notice that the previous sentence uses a system call to open the library function to generate a return value, this value is called fildes, in linux systems, when using file operations, when the open function gets the file fields that in the system system will create the corresponding structure and assign a number, through fildes can successfully find the file. The second argument corresponds to the buffer, and you can notice that there are many numbers in this argument, which are all octal bits, indicating the contents of the file read by the read function, in this case it is the contents of the library function, which starts with \177ELF, and combined with the previous system call you can tell that this is a library file called ld.so.cache. library file and is stored in binary so that it can be handed over directly to the cache when loaded, for fast loading purposes.

For ELF files, it is Executable and Linkable Format, a target file format for binary files,

executable files, target code and shared libraries, etc. The ELF file consists of four main parts, namely the ELF header, Program header table, Section, In fact, only the ELF header is fixed and the rest are determined by the values. The most important thing in an ELF file is to locate the header and map it to memory by parsing the active LOAD type section and then returning it after loading.

Returning to the read function, it returns a value of 832, which means that the library file has 832 bytes.

mmap(NULL, 10916, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f87458e6000

The main purpose of this command is to allocate memory for a file, or to map a file or device into memory, in this case allocating the same size of memory space as the size of the file being read, and setting it to read-only mode and the fact that modifications can only be made privately.

In deeper terms, mmap is a method of memory mapping a file or other object into the incoming address space, enabling a one-to-one relationship between the file's address on disk and the virtual address in the process, and when the mapping relationship is complete, the process can use pointers to manipulate this section of memory. At the same time, in kernel space, changes to this area are also directly reflected in the user state to complete the file sharing between different processes.

Compare the difference between the mmap function, the read function and the write function. In conventional calls to the file system, the cache mechanism is used to improve the efficiency of reading and writing and to protect the disk, resulting in a

copy of the file page from the disk to the page cache when reading and writing the file.

close (3) = 0

Before closing the library file, the file has been allocated memory and the address is 0x7f31d69ad000, the close function returns 0 to indicate that the file was closed successfully.

openat(AT_FDCWD,"/lib/x86_64-linux-gnu/libc.so.6",
O_RDONLY|O_CLOEXEC) = 3

Here the operating system opens another library file named x86_64-linux-gnu/libc.so.6. It is worth noting that x86_64-linux-gnu is a very important folder in the Linux operating system and by default holds the system libraries for the Gnu C or C++ compiler, thus it is known that here the operating system This is because in the operating system when a file is closed, its corresponding fields are deleted and the operating system follows the principle of minimum allocation of fields, which results in new files being allocated without opening other files. This results in a new file being assigned the fields value 3 without opening another file.

pread64(3,"\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0
0"..., 784, 64) = 784
pread64(3,"\4\0\0\0\20\0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0
\0\0", 32, 848) = 32
pread64(3,"\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\t\233\222%\274\260\320\31

331\326\10\204\276X>\263"..., 68, 880) = 68

All three system call commands use the pread64 function, in Linux the underlying system call was renamed in kernel 2.6 and the pread() function was renamed to pread64(), but the function is the same, indicating that the file is read starting at a specific offset and ending at a specific byte, in this case the pread64 function reads the library file from 64 to 880 bytes.

The read function can be interpreted as a read request to the disk, but at a deeper level the request goes through the vfs layer, followed by the concrete file system layer, the cache layer, the generic block layer, the IO scheduling layer, the block device driver layer and finally the physical block device layer.

fstat(3, {st_mode=S_IFREG|0755, st_size=2029224, ...}) = 0

As with the other file above, here the OS goes to get the status of the newly read file and gets a file size of 2029224 bytes, then the OS starts preparing to map virtual space in the process for the file.

mmap(NULL,8192,PROT_READ|PROT_WRITE,MAP_PRIVATE|AP_ANONYMOUS, -1, 0) = 0x7f87458e4000

Notice the presence of MAP_ANONYMOUS here, where the meaning is virtual anonymous space for the process, and in your case private anonymous space, the main role is for memory allocation, which chases the OS to implement the malloc()h function in glibc, the traditional malloc() function implementation relies on the brk function and usually brk is used for So it is clear that the OS has allocated 8192 bytes

of anonymous private space here for subsequent memory allocation.

pread64(3,"\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0", 784, 64) = 784

pread64(3,"\4\0\0\0\20\0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0", 32, 848) = 32

pread64(3,"\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\t\233\222%\274\260\320\31\331\326\10\204\276X>\263"..., 68, 880) = 68

Again, as with the other file above, the operating system reads a specific segment of the file.

mmap(NULL,2036952,PROT_READ,MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f87456f2000

Here too, the operating system allocates memory space for the file and sets the parameter MAP_DENYWRITE to indicate that the file is denied write access, thus protecting the contents of the file, which returns a pointer to that address space.

mprotect(0x7f8745717000, 1847296, PROT_NONE) = 0

This instruction is about memory protection, specifically the control page protection, this system call will make a specific page have a protection mechanism, when an operation is used in the specific page that has the protection mechanism that violates the protection condition, the program will get the signal SIGBUS or SIGSEGV, in these parameters, the first parameter is the specific address of the memory that is protected,

then the second is the address and the third is the type of protection, in this case the area is protected from any permissions, such as read or write or execution. Here you can see that the interval is 1847296 bytes, while the size of a memory page in Linux is 4K i.e. 4096 bytes, and len must be an integer multiple of the page size, which gives us here an interval of 451 pages.

mmap(0x7f8745717000, 1540096, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,3,0x2500) = 0x7f8745717000

mmap(0x7f874588f000,303104,PROT_READ,MAP_PRIVATE|MAP_ FIXED|MAP_DENYWRITE,3,0x19d000)=0x7f87458 f000

mmap(0x7f87458da000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,3,0x1e70 0) = 0x7f87458da000

mmap(0x7f87458e0000, 13528, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f87458e0000

In all four lines of the system call, the mmap function is used. From the first line, the function executes and read-protects a section of the address space of the file, and a new parameter, MAP_FIXED, is used to ensure that the file is mapped in the process, so that any mapping previously generated in a specific area is removed and mapped to a

specific area of the file. All in all this is to ensure that the library function is correctly mapped and protected by the system. The next three commands are similar to the first one in that they protect certain address spaces of the file and overwrite the memory mapping.

arch_prctl(ARCH_SET_FS, 0x7f87458e5580) = 0

This command also appears above, where this system call function sets the architecture state of the thread or process, and as you can tell from the arguments, it sets the FS register-based architecture for that address. It is worth noting that this address is not far after the memory has been allocated as described above.

mprotect(0x7f87458da000, 12288, PROT_READ) = 0

mprotect(0x55f32e73d000, 4096, PROT_READ) = 0

mprotect(0x7f8745916000, 4096, PROT_READ) = 0

Similarly, read protection is provided in areas starting at these three addresses of 12288, 4096 and 4096 bytes in length, 3 pages, 1 page and 1 page of space respectively.

munmap(0x7f87458e6000, 10916)

This function is mainly used to contact the mapping relationship between the address in the process and the memory, returning 0 if successful and -1 if not, you can notice that the address corresponds to the file where the memory allocation was first made, which is the ld.so.chache library file.

brk(NULL) = 0x56419b3e0000

brk(0x56419b401000) = 0x56419b401000

The first line of the command appears at the very beginning of the system call, for brk(NULL) whose main function is to check the current interrupt, and in the second brk function will set the interrupt of the data segment of the process to address 0x56419b401000.

fstat(1,{st_mode=S_IFCHR|0620,st_rdev=makedev(0x88, 0x1), ...}) = 0

This system call is getting the status of the file, as you can see from the parameters, the first parameter indicates the standard output, because I didn't go ahead and run a file directly but typed the target of execution in the input, so here the function receives the first parameter as the standard output, while the second parameter is some specific status about that target, here you can see that its status is S_IFCHR, which indicates the character device, and after that this is its device number, and more information is omitted, and finally a successful read returns 0.

write(1,"Hello,World\n",12Hello,World) = 12

Here the write function is used to write, again, because the first argument is 1, which means that the write target is the standard output, which means that the character string in the second argument is displayed on the terminal, in this case this function achieves the effect of displaying "Hello,World" on the terminal. The third parameter is a count, which counts how many characters are in the string.

close(1) = 0

close(2) = 0

After the write function is called, the process ends, the standard input and standard output are closed, and the process is exited.

docker run --rm -ti lab1_ex1 strace -e trace=read echo "Hello"

This is the second command on the pdf file about the strace command trace, where the -e trace=read command appears to indicate that only the command about the read operation will be traced, so after the execution of this command the operating system returned me about the read command.

read(3,"\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\360q\2\0\0\0\0\0 "..., 832) = 832

Likewise, strace can track different and specific commands individually, such as -e strace=write etc. to view a particular type of system call individually.

docker run --rm -ti -v $(pwd)/:/root/lab lab1_ex1 /bin/bash -c "cd /root/lab; gcc -o hello hello.c; strace ./hello"

This command is used to trace the system calls generated by the execution of the hello file. After executing it, I found that the system call used was like the one used to execute echo "hello world", with the following differences.

execve("./hello", ["./hello"], 0x7fff708d8810 /* 8 vars */) = 0

For the echo command, you need to find the echo executable in the Linux system file and then execute the subsequent commands, while for the hello.c file you need to find the hello.c file and then execute it in the subsequent operations, while the other operations are very similar, only the memory space is divided differently.

In conclusion I would like to talk about the docker run -v command, this command is used to bind data volumes between the host and the container, this will cause the folders bound between the host and the container to be synchronized, for example, in the container to operate on the files in the folder bound in the container will also be operated on the host, this allows the container to interact with the data outside, but there is a hidden danger that If the container is destroyed, the data will be lost, so it is safe to create another folder as an intermediary between the host and the container so that the data can be passed between them.