In lab1, I used docker to create an ubuntu image, and in the ubuntu container, I used Linux commands to complete the lab.

During my learning process I learned that docker is like a container, different containers usually do not affect each other, and can be deployed quickly, it can package the development environment to the maintenance staff, so it can directly solve some environment incompatibility problems that may be encountered when moving from the development environment to the production environment. During my learning process, I learned that the three basic concepts of docker are image, container, and repository. To build docker containers, we need to build docker images first, in essence, containers are running instances of images, containers and containers are independent of each other, and the repository is where the image files are stored. docker containers are very fast to start, which is much faster than traditional virtual machines, and docker's use of system resources is very high. The use of system resources is very high, and many docker containers can be run on one host at the same time, which can also improve the efficiency of our operations.

Next, I will briefly describe how I started using docker. First, I opened docker, opened my computer's terminal, and in my terminal, I typed the command "docker ps -a", which allows me to see the status of my current process. The ps command can be followed by -a, -G, -g, -p, -T, -t, -U, and -u, and the ps command can be followed by different options to select different processes. I use the command "ps -a" to display all the processes, including other user processes and my own process information, which includes the ID of the container in the process, the image, command, created time, the status of the process at this time is up or exited status and details of the time and the container name. Then I type "docker run -it -name=lab1 ubuntu bin/bash" to create a container named lab1 in the ubuntu image. The -i is the run parameter that keeps the standard input open, and -i is usually used in conjunction with -t. -t is creating a pseudo-terminal for docker that connects to the container's standard input excuse, allowing us to get into interactive mode in the container and start doing our operations. We can do any system manipulation on the pseudo terminal without damaging our own computer by mistake. The "bin/bash" at the end of the command is what keeps the process running, and when I follow this command with "bin/bash", it means that when I start the container, I start bash. I find that when I enter lab1 When I pressed exit, I exited the container and went back to the terminal on my own computer, and when I tried to use the command "docker exec -it lab1 bin/bash" directly, I failed, I couldn't get into the container directly. Later I found out that I should first execute the command "docker start lab1" to change the status of lab1 from exited to up. Finally, I could execute "docker exec -it lab1 bin/bash" to get into the container successfully. Next, I will analyze the system calls.

When I tried to get the strace command directly with the "apt-get install strace" command, the system displayed "E: Unable to locate package strace", When I used the "apt-get update" command, I successfully installed the strace command using the previous command. I find the "apt-get" command very useful. I can install any command I want by using this command, it's very fast and convenient.

When I run "strace echo "Hello World"", several system function calls appear, and I will explain each one of them and how they work. The following is the result of running the strace system function call.

**root@2e838cdba31e:/#** **strace echo "Hello World"**

**execve("/usr/bin/echo", ["echo", "Hello World"], 0xffffc4ef37c8 /\* 8 vars \*/) = 0**

**brk(NULL) = 0xaaaafdac2000**

**faccessat(AT\_FDCWD, "/etc/ld.so.preload", R\_OK) = -1 ENOENT (No such file or directory)**

**openat(AT\_FDCWD, "/etc/ld.so.cache", O\_RDONLY|O\_CLOEXEC) = 3**

**fstat(3, {st\_mode=S\_IFREG|0644, st\_size=6263, ...}) = 0**

**mmap(NULL, 6263, PROT\_READ, MAP\_PRIVATE, 3, 0) = 0xffffbd444000**

**close(3) = 0**

**openat(AT\_FDCWD, "/lib/aarch64-linux-gnu/libc.so.6", O\_RDONLY|O\_CLOEXEC) = 3**

**read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0\267\0\1\0\0\0\350A\2\0\0\0\0\0"..., 832) = 832**

**fstat(3, {st\_mode=S\_IFREG|0755, st\_size=1449992, ...}) = 0**

**mmap(NULL, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0xffffbd442000**

**mmap(NULL, 1518672, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_DENYWRITE, 3, 0) = 0xffffbd2a5000**

**mprotect(0xffffbd3ff000, 65536, PROT\_NONE) = 0**

**mmap(0xffffbd40f000, 24576, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x15a000) = 0xffffbd40f000**

**mmap(0xffffbd415000, 11344, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_ANONYMOUS, -1, 0) = 0xffffbd415000**

**close(3) = 0**

**mprotect(0xffffbd40f000, 12288, PROT\_READ) = 0**

**mprotect(0xaaaabf496000, 4096, PROT\_READ) = 0**

**mprotect(0xffffbd449000, 4096, PROT\_READ) = 0**

**munmap(0xffffbd444000, 6263) = 0**

**brk(NULL) = 0xaaaafdac2000**

**brk(0xaaaafdae3000) = 0xaaaafdae3000**

**fstat(1, {st\_mode=S\_IFCHR|0620, st\_rdev=makedev(0x88, 0x1), ...}) = 0**

**write(1, "Hello World\n", 12Hello World**

**) = 12**

**close(1) = 0**

**close(2) = 0**

**exit\_group(0) = ?**

**+++ exited with 0 +++**

The first one that appears is the execve execution file system call function.

**execve("/usr/bin/echo",["echo", "Hello World"], 0xffffc4ef37c8 /\* 8 vars \*/) = 0**

execve has three arguments, path, argv[] and envp[]. execve reads the echo executable by reading the file path "/usr/bin/echo", in fact, there are many systems commands under /usr/bin/, which are executed by the corresponding command file. echo is one of the system commands, it writes Here, the echo command writes "Hello World" to the standard output port, usually followed by a string of characters, separated by whitespace, with a newline at the end. The execve command is actually implemented by do\_execve, and after entering the kernel the process allocates and uses its own file After entering the kernel, the process allocates and uses its own file descriptor, allocates a new address space for the process, allocates a stack memory area, opens the executable file, checks the permissions of the executable file and pre-reads the executable file information into buf, then calculates the environment variables and various parameters and copies them all to the stack in the new address space, then parses the executable file and maps the file code, data, etc. into memory. Then we specify the command line arguments to be passed to the new process by argv[] and print out the text content. Allow me to give an example here to illustrate my understanding. When I click on a random file on my computer and try to open it, the moment I click on the file and the moment I open it, the execve function is called in my operating system to perform a system interrupt, and the kernel interface is opened to receive the called function and arguments and pack them into the user heap. When the execve system call enters the kernel, it allocates a new process to store the executable, loads the new process program into the memory space of the child process, and then discards the part behind the original child process, except for the process ID, the stack and data of the child process will be replaced by the corresponding part of the new process. In execve's system function call, 0xffffc4ef37c8 I understand is the ID of the incoming parameter, and 8 vars is the parameter variable.

**brk(NULL) = 0xaaaafdac2000**

brk is a command to change the data space size allocation. In essence, the argument of the brk() function is an address. If we already know the starting address of the heap, and the size of the heap, then we can achieve the purpose of adjusting the heap by modifying the address argument in brk(). In this function, brk(NUll) means that the size of the data segment has not been changed and its value is null. 0xaaaafdac2000 is the address of the data segment content padding of brk.

**faccessat(AT\_FDCWD, "/etc/ld.so.preload", R\_OK) = -1 ENOENT (No such file or directory).**

faccessat is a system call function to test the accessibility of a file. When we open a file, the kernel permissions test for access is performed based on the valid user ID and valid group ID of the process as the basis for execution, and when it runs, it can be checked if the given file can be accessed by the faccessat system call function command, i.e., the permission test for the file. faccessat has two arguments, one is the file path and the other is mode. faccessat() uses the absolute path, mode contains R\_OK to test read permissions, W\_OK to test write permissions, X\_OK to test execute permissions, and F\_OK to test existence. In this paragraph system call function mode result failure is shown as -1, success is shown as 0. In this paragraph instruction, AT\_FDCWD indicates that the current working directory is used when the value is passed in the system call, in this case, the system call of accessat() works the same as the system call of access(). accessat() reads the file by its The result of reading a file by its absolute path is R\_OK = -1 ENOENT (No such file or directory), i.e., reading the file failed and the error shows that the file or folder was not found.

**openat(AT\_FDCWD, "/etc/ld.so.cache", O\_RDONLY|O\_CLOEXEC) = 3**

The openat() function opens or creates a file for reading or writing. openat() contains the file identifier, the path and the flags specified by the oflag argument. oflag uses different constants that can be "or" to form the oflag argument. When we use openat(), a file identifier fd is opened, but AT\_FDCWD in this command indicates that the current working directory is used when the value is passed in the system call. In addition, ld.so.cache in the openat() function is a high speed binary library file, O\_RDONLY is to open a read-only file, and O\_CLOEXEC is marked as close-on-exec. In this command, a read-only file is opened by openat(), and the read-only file is assigned to the 3rd file descriptor, because when running any program, by default three file descriptors are opened, i.e. 0 marks the standard input file descriptor, 1 marks the standard output file descriptor, and 2 marks the standard error output file descriptor. So this command will start from the third one when allocating identifiers. openat() will convert the path of the incoming character into the corresponding inode node and dentry in the kernel, the inode will point to an inode structure, and the inode structure will point to a region in the disk, and finally access to the file data in the disk, according to our The openat() command generates a system interrupt, and then we go to the dentry, and through the inode pointer to a region of the disk, we can access the disk, read a data from the disk, read the corresponding data and copy it to a data buffer, and then the openat() function reads the data copied from the disk in the data buffer. The openat() function then reads the data copied from the disk in the data buffer.

**fstat(3, {st\_mode=S\_IFREG|0644, st\_size=6263, ...}) = 0**

fstat() system call function is to get the file status function, including file and buf two parameters, through the file name filename to get the file information, and saved in the buf pointed to the structure stat, if the execution is successful, return 0, the execution fails to return -1, the error code from errno. The command indicates that through open () to return a file descriptor st\_mode=S\_IFREG|0644 is to record the file type as common and the permission of the file is user read/write and other users can read. st\_size=6263 records the actual size of the file, and finally fstat() returns 0 value, which means the execution of reading file status is successful. fstat() finds the file state, is a combination of dentry structure and inode structure analysis, from the inode structure to obtain file information, including attribute information, file permissions, file creation time file data storage location in the hard disk, etc.. The st\_mode is used to determine the type of the file, to save the settings of the executable file and to save the access rights of the file. In st\_mode, each bit corresponds to a corresponding permission.

**mmap(NULL, 6263, PROT\_READ, MAP\_PRIVATE, 3, 0) = 0xffffbd444000**

mmap() is a memory mapping function that maps a file to the process address space so that the address of the file on disk corresponds to the address in the process virtual address space. The process can then read and write to this section of memory using pointers, and the system will automatically write back to the corresponding file on disk, i.e. to complete the operation of the file but without the need to call system functions such as read and write. In addition, changes to this area in kernel space can be directly reflected in user space, thus enabling file sharing between different processes. First, the mmap() system call sets the starting address of the specified map, which is usually set to NULL, and reflects the length of the file mapped to memory. The changes made in this area will not be written to the original file. The file descriptor is then returned by open, indicating the file to be mapped into memory. offset is the offset from the start of the file, and must be an integer multiple of the paging size, usually set to 0, which means that the mapping starts at the start of the file. The call to mmap() in this function allocates a memory to map the 6263 byte length page read into memory. The mapped area is protected in such a way that it can be read and MAP\_PRIVATE is private when modified and copied when written. Place a file descriptor in the third bit. The file descriptor bit 3 is returned via open() and the last file start offset is 0. 0xffffbd444000 is the location parameter of the mapped memory file.

**close(3) = 0**

close() is a call to close the file descriptor so that the file descriptor is no longer thought of as any file and can be used again in a new file operation. Usually, when a program terminates, the kernel closes the file descriptor by calling close() even if the user does not close the file descriptor, but it is worth saying that since some programs are executed for a long time, the kernel may not be able to close the file descriptor, so we should remember to close the open file descriptor, otherwise it will take up a lot of system resources. After the success of closing the file descriptor, the return value is 0, otherwise the return value is -1, and the errno setting indicates an error, in this command, close() closes the file descriptor returned by open successfully.

**openat(AT\_FDCWD,"/lib/aarch64-linux-gnu/libc.so.6", O\_RDONLY|O\_CLOEXEC) = 3**

The openat() function is to open or create a file to read or write. In this command AT\_FDCWD indicates that the current working directory is used when the value is passed in the system call. openat() converts the path of the incoming character into the corresponding inode node and dentry in the kernel. inode will point to an inode structure, which in turn will point to a region in the disk, and finally access to the Then we enter the dentry, the inode pointer points to a region in the disk, we can enter the disk, read a data from the disk, read the corresponding data and copy it to a data buffer, then the openat() function reads the data copied from the disk in the data buffer. reads the data copied from the disk. In this instruction, openat() opens the current working directory /lib/aarch64-linux-gnu/libc.so.6, which is a very important library file that holds the C, C++, and other compiled software. This file is a read-only file and returns a file descriptor.

**read(3,"\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0\267\0\1\0\0\0\350A\2\0\0\0\0\0"..., 832) = 832**

read () is a read input system call command. read read reads n bytes of data from the object referenced in the file descriptor into the buf buffer. When the system interrupts, the interrupt handler continues and looks up the system call table according to the system call number and gets the kernel function sys\_read from the system call table to handle the read system call, and finally passes the arguments and runs the sys\_read function, which is the kernel entry for the read system call. In this command read reads the 832 bytes of the 177ELF file under the file descriptor.

**fstat(3, {st\_mode=S\_IFREG|0755, st\_size=1449992, ...}) = 0**

fstat() system call function is to get the file status function, command execution success will return 0, execution failure will return -1, error code from errno. fstat() to find the file status, is a combination of dentry structure and inode structure analysis, from the inode structure to get the file information, including attribute information, file permissions, file creation The time of the file's data storage location in the hard disk, etc. The st\_mode is used to determine the type of the file, to save the settings of the executable and to save the access rights of the file. In st\_mode, each bit corresponds to a corresponding permission. This command returns a file descriptor in the third bit by open(), and reads the file information. st\_mode=S\_IFREG|0755 records the type of the file as common and the permission of the file as readable, writable and executable by user and readable and executable by other users. st\_size=1449992 records the actual size of the file, and finally fstat() returns a value of 0, indicating that the read file status was executed successfully.

**mmap(NULL, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0xffffbd442000**

mmap() is a memory mapping function that maps a file to the process address space so that the address of the file on disk corresponds to the address in the process virtual address space. The process can then read and write to this section of memory using pointers, and the system will automatically write back to the corresponding file on disk, i.e. to complete the operation of the file but without the need to call system functions such as read and write. In addition, changes to this area in kernel space can be directly reflected in user space, thus enabling file sharing between different processes. First, the mmap() system call sets the starting address of the specified map, which is usually set to NULL and reflects the length of the file mapped to memory. MAP\_ANONYMOUS is an anonymous map that can be returned to the system immediately after unmapping, and can be modified in terms of allocation size, permissions, etc. Each allocation is independent and can be used to pass some Mach VM flags, or - 1 if no flags are associated. 1. The file descriptor is then returned by open, indicating the file to be mapped into memory. offset is the offset from the start of the file, and must be an integer multiple of the paging size, usually set to 0, which means that the mapping starts from the start of the file. In this function call in mmap () to allocate a memory, will read the 8192 bytes length of the page mapped into memory, the mapping area is protected in a way that can be read or can be written, and MAP\_PRIVATE modified when private, in the write copy, MAP\_ANONYMOUS for anonymous mapping, its flag is -1, the offset is 0. The offset is 0. 0xffffbd442000 is the location parameter of the mapped memory file.

**mmap(NULL, 1518672, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_DENYWRITE, 3, 0) = 0xffffbd2a5000**

Since I have described above about how the mmap() system call command is invoked, here I will briefly describe how it implements the system call in this command. In this command, the mmap() system call first sets the start address of the specified map to NULL. mmap() allocates a memory to map the 1518672-byte-long page read into memory, and the PROT\_READ|PROT\_EXEC mapped area protection attribute is mapped area allowed to read or mapped area allowed to execute. MAP\_ MAP\_DENYWRITE only allows writes to the mapped area, other direct writes to the file will be rejected. The file descriptor is then returned by open, indicating the file to be mapped into memory. The file starts at an offset of 0. 0xffffbd2a5000 is the location parameter of the mapped memory file.

**mprotect(0xffffbd3ff000, 65536, PROT\_NONE) = 0**

mprotect() controls the protection of the memory area. mprotect() parameters include the address of the memory interval, the size of the memory interval, and the new protection flag. If the execution succeeds it returns 0, and if it fails, it returns -1 and sets the erron variable indicating the reason for the failure. mprotect() changes the protection of the memory map to the specified protection, and the entire page of the package from the start address (the start address of the memory page) for the length of len of the capital space will be affected by the access protection change. mprotect() function can be be used to modify the protection attribute of a specified section of memory area. In this command

0xffffbd3ff000 is the start address of the memory protection area, 65536 is the byte size of the memory area, PROT\_NONE indicates that the contents of the memory segment are inaccessible, and the final return value of 0 indicates the successful execution of the command for memory area protection

**mmap(0xffffbd40f000,****24576,PROT\_READ|PROT\_WRITE,MAP\_PRIVATE|****MAP\_FIXED|MAP\_DENYWRITE, 3,** **0x15a000) =** **0xffffbd40f000**

mmap() is a memory mapping function that maps a file to the process address space so that the address of the file on disk corresponds to the address in the process virtual address space. The process can then read and write to this section of memory using pointers, and the system will automatically write back to the corresponding file on disk, i.e., to complete the operation of the file but without the need to call system functions such as read and write. In addition, changes to this area in kernel space can be directly reflected in user space, thus enabling file sharing between different processes. First, the mmap() system call sets the starting address of the specified mapping, in this command, the starting address of the specified mapping space is 0xffffbd40f000, and the 24576-byte-long page read is mapped into memory, the mapping area is protected by allowing reads or writes, and MAP\_PRIVATE is private when modified and copied when written. MAP\_DENYWRITE only allows write operations to the mapped area, other direct write operations to the file will be rejected. The file descriptor is then returned by open, indicating the file to be mapped into memory. The offset of the file is 0x15a000. 0xffffbd40f000 is the location parameter of the mapped memory file.

**mmap(0xffffbd415000,11344,PROT\_READ|PROT\_WRITE,MAP\_PRIVATE|MAP\_FIXED|MAP\_ANONYMOUS, -1, 0) = 0xffffbd415000**

mmap() is a memory mapping function that maps a file to the process address space so that the address of the file on disk corresponds to the address in the process virtual address space. The process can then read and write to this section of memory using pointers, and the system will automatically write back to the corresponding file on disk, i.e., to complete the operation of the file but without the need to call system functions such as read and write. In addition, changes to this area in kernel space can be directly reflected in user space, thus enabling file sharing between different processes. First, the mmap() system call sets the starting address of the specified mapping, in this command, the starting address of the specified mapping space is 0xffffbd415000, and the 24576-byte-long page read is mapped into memory. The MAP\_FIXED parameter start refers to an address that cannot be successfully mapped, then the mapping is dropped, and no correction is made to the address. MAP\_ANONYMOUS performs an anonymous mapping with a flag of -1 and a file offset of 0. 0xffffbd415000 is the location parameter of the mapped memory file.

**close(3) = 0**

close() is a system function call that closes a file descriptor so that it will not run any more files and can be used again in new file operations. The return value is 0 if the file descriptor is closed successfully, otherwise the return value is -1, and the errno setting indicates an error.

**mprotect(****0xffffbd40f000, 12288,** **PROT\_READ) = 0**

mprotect() controls the protection of the memory area. mprotect() parameters include the address of the memory interval, the size of the memory interval, and the new protection flag. If the execution succeeds it returns 0, and if it fails, it returns -1 and sets the erron variable indicating the reason for the failure. mprotect() changes the protection of the memory map to the specified protection, and the entire page of the package from the start address (the start address of the memory page) for the length of len of the capital space will be affected by the access protection change. mprotect() function can be be used to modify the protection attribute of a specified section of memory area. In this command

0xffffbd40f000 is the start address of the memory protection area, 12288 is the byte size of the memory area, PROT\_READ indicates that the memory segment is readable, and the final return value of 0 indicates that the command to protect the memory area was executed successfully.

**mprotect(****0xaaaabf496000,** **4096, PROT\_READ) = 0**

In this command 0xaaaabf496000 is the start address of the memory protection area, 4096 is the byte size of the memory area, PROT\_READ means the memory segment is readable, and finally the return value 0 indicates the successful execution of the command for memory area protection.

**mprotect(****0xffffbd449000, 4096, PROT\_READ) = 0**

0xffffbd449000 is the start address of the memory protection area, 4096 is the byte size of the memory area, PROT\_READ indicates that the memory segment is readable, and the final return value of 0 indicates that the command to protect the memory area was executed successfully.

**munmap(****0xffffbd444000, 6263) = 0**

munmap() is a command to delete a mapping. munmap() has two parameters, address and byte size, and returns 0 when the command completes successfully, or -1 if not, and sets errno to indicate an error. munmap() unmaps the relationship, and access to the original address will cause a segment error to occur. In this command, munmap() removes the mapping with address 0xffffbd444000 and byte size 6263, and it runs successfully.

**brk(NULL) = 0xaaaafdac2000**

brk （）is a command to change the data space size allocation. In essence, the argument of the brk() function is an address. If we already know the starting address of the heap, and the size of the heap, then we can achieve the purpose of adjusting the heap by modifying the address argument in brk(). In this command, brk(NUll) means that the size of the data segment has not been changed and its value is null. 0xaaaafdac2000 is the address of the data segment content padding of brk.

**brk(0xaaaafdae3000) = 0xaaaafdae3000**

In this command, brk() indicates that the size of data segment is 0xaaaafdae3000, and 0xaaaafdae3000 is the data segment content padding address of brk.

fstat(1, {st\_mode=S\_IFCHR|0620, st\_rdev=makedev(0x88, 0x1), ...}) = 0

The fstat() system call function is a function to get the file status, containing two arguments: file and buf. It gets the file information by filename and saves it in the structure stat pointed to by buf, returning 0 if the execution succeeds and -1 if it fails, with an error code derived from errno. fstat() finds the file status by combining the dentry structure and inode structure analysis, from the inode structure to obtain file information, including attribute information, file permissions, file creation time file data storage location in the hard disk, etc. The st\_mode is used to determine the type of the file, to save the settings of the executable and to save the access rights of the file. In st\_mode, each bit corresponds to a corresponding permission. st\_mode= S\_IFCHR|0620 is to record the file type as normal and the permission of the file as readable and writable by user and readable by other users. st\_rdev=makedev(0x88, 0x1) is the special device type, used for the inode of special files. Finally fstat() returns a value of 0, indicating that the read file status was executed successfully.

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

write() is an input command. write() is implemented by the operating system calling the sys\_write function, which first gets the inode pointer and flip pointer of the device node file from the file identifier. inode pointer has device number information, which tells the operating system which device driver to use, and flip pointer has fops information, which tells the operating system where the corresponding fops method function can be found. Then sys\_write will call the write method of the driver to write to the device. In this command, write() writes a 12-byte message "Hello World\n" to the buffer pointed to by the file.

**close(1) = 0**

close() is a system function call to close a file descriptor so that it is no longer thought of only as any file and can be used again in new file operations. The return value is 0 if the file descriptor is closed successfully, otherwise the return value is -1. errno is set to error, in this command, close() closes the standard error output file descriptor successfully.

**close(2) = 0**

In this command, close() closes the standard error output file descriptor successfully.

**exit\_group(0) = ?**

exit\_group() exits all threads in the process.

cd ex1  
docker build -t lab1\_ex1 .  
docker run --rm -ti lab1\_ex1 strace echo "Hello World"  
*# This is an alternative way of calling strace, what's the -e flag?*docker run --rm -ti lab1\_ex1 strace -e trace=read echo "Hello"  
*# This is how to run a docker to run the strace in the hello command*docker run --rm -ti -v $(pwd)/:/root/lab lab1\_ex1 /bin/bash -c "cd /root/lab; gcc - o hello hello.c; strace ./hello"

I executed the above commands on docker and compared them with the detailed system calls of the command "strace echo "Hello World"". Next, I will first explain my understanding of these lines of commands and my distinction between the system calls of these commands.

First I created an image of lab1\_ex1 using docker, docker run --rm -ti lab1\_ex1 strace echo "Hello World" In this command, docker run is starting the container -rm can automatically clean up the container's internal file system, lab1\_ex1 is the image name, strace echo "Hello World" in this command, docker run is to start the container -rm can automatically clean up the container internal file system when the container is launched, lab1\_ex1 is the image name, strace echo "Hello World" in the execution of the system call command in the previous has been discussed in detail, here will not be repeated, the other same command in the previous discussed, here are not discussed one by one.

**docker run --rm -ti lab1\_ex1 strace** **-e trace=read echo "Hello"**

docker run --rm -ti lab1\_ex1 strace -e trace=read echo "Hello"

read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0\267\0\1\0\0\0\350A\2\0\0\0\0\0"..., 832) = 832

Hello

+++ exited with 0 +++

After running docker run --rm -ti lab1\_ex1 strace -e trace=read echo "Hello", you can see that in this command, only the specified read "Hello" is traced, and the invocation command executes a byte 832 library file. -e is the specified trace content.

**docker run --rm -ti -v $(pwd)/:/root/lab lab1\_ex1 /bin/bash** **-c "****cd /root/lab; gcc - o hello hello.c; strace ./hello"**

In this command, - v is to bind the current path and container into a data volume, - c will pass the command inside the quotation marks into the terminal and execute it, and finally print out "hello", the executed system command file is as follows：

docker run --rm -ti -v $(pwd)/:/root/lab lab1\_ex1 /bin/bash -c "cd /root/lab; gcc -o hello hello.c; strace ./hello"

execve("./hello", ["./hello"], 0xfffffd1dc6f0 /\* 8 vars \*/) = 0

brk(NULL) = 0xaaaaf2457000

faccessat(AT\_FDCWD, "/etc/ld.so.preload", R\_OK) = -1 ENOENT (No such file or directory)

openat(AT\_FDCWD, "/etc/ld.so.cache", O\_RDONLY|O\_CLOEXEC) = 3

fstat(3, {st\_mode=S\_IFREG|0644, st\_size=10620, ...}) = 0

mmap(NULL, 10620, PROT\_READ, MAP\_PRIVATE, 3, 0) = 0xffff8f2f8000

close(3) = 0

openat(AT\_FDCWD, "/lib/aarch64-linux-gnu/libc.so.6", O\_RDONLY|O\_CLOEXEC) = 3

read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0\267\0\1\0\0\0\350A\2\0\0\0\0\0"..., 832) = 832

fstat(3, {st\_mode=S\_IFREG|0755, st\_size=1449992, ...}) = 0

mmap(NULL, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0xffff8f2f6000

mmap(NULL, 1518672, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_DENYWRITE, 3, 0) = 0xffff8f15a000

mprotect(0xffff8f2b4000, 65536, PROT\_NONE) = 0

mmap(0xffff8f2c4000, 24576, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x15a000) = 0xffff8f2c4000

mmap(0xffff8f2ca000, 11344, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_ANONYMOUS, -1, 0) = 0xffff8f2ca000

close(3) = 0

mprotect(0xffff8f2c4000, 12288, PROT\_READ) = 0

mprotect(0xaaaacde60000, 4096, PROT\_READ) = 0

mprotect(0xffff8f2fe000, 4096, PROT\_READ) = 0

munmap(0xffff8f2f8000, 10620) = 0

fstat(1, {st\_mode=S\_IFCHR|0620, st\_rdev=makedev(0x88, 0), ...}) = 0

brk(NULL) = 0xaaaaf2457000

brk(0xaaaaf2478000) = 0xaaaaf2478000

write(1, "Hello, World!\n", 14Hello, World!

) = 14

exit\_group(0) = ?

+++ exited with 0 +++

By interpreting the system function commands, I found that the commands executed are the same, with some minor changes in the parameters. By comparing the two commands, I have the following findings. When I execute the command strace echo "Hello World", it is using the echo command which is based on the Linux terminal and prints the values of the variables we input to the display again, while " cd /root/lab; gcc - o hello hello.c; strace . /hello" is a command that opens the C file that we have already compiled, by . /hello to print out the compiled contents of the program, the other system calls are almost the same.

The above is my understanding of the contents of lab1, I will continue to learn more about Linux related content.