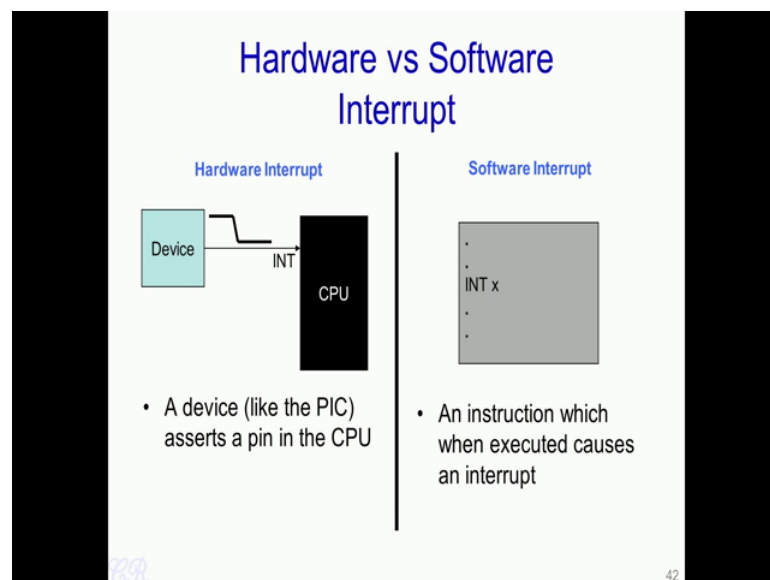


Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 04
Lecture – 16
Software Interrupts and System calls

In this video, we look at an important type of interrupts known as Software Interrupts; and their applications for in system calls.

(Refer Slide Time: 00:28)

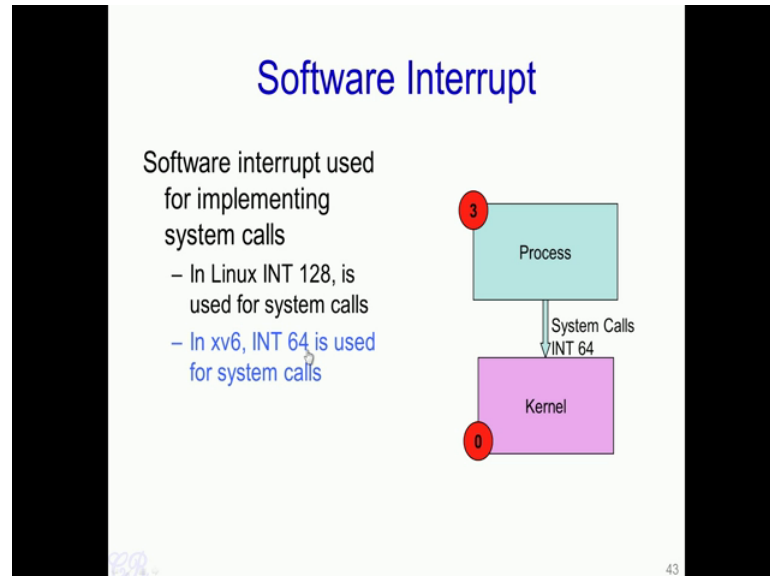


In the previous video, we had looked at hardware interrupts; we had seen how a device such as a keyboard or a network card could assert a particular signal in the CPU. And this would cause the CPU to asynchronously execute an interrupt handler corresponding to the device. So, as we have seen in the previous videos, this device would typically send a signal to the CPU through an intermediate device such as a PIC or a programmable interrupt controller.

In much the same way, we have what is known as software interrupt. However, unlike having an external device which causes the interrupt here an instruction in the program would trigger the interrupt. In this particular case, for example, an instruction such as INT would cause the interrupt to occur and the operating system to execute. So, here the

instruction is INT x, so x here is the interrupt number, it typically has a value less than 256, and it is used to specify or distinguish between software interrupts.

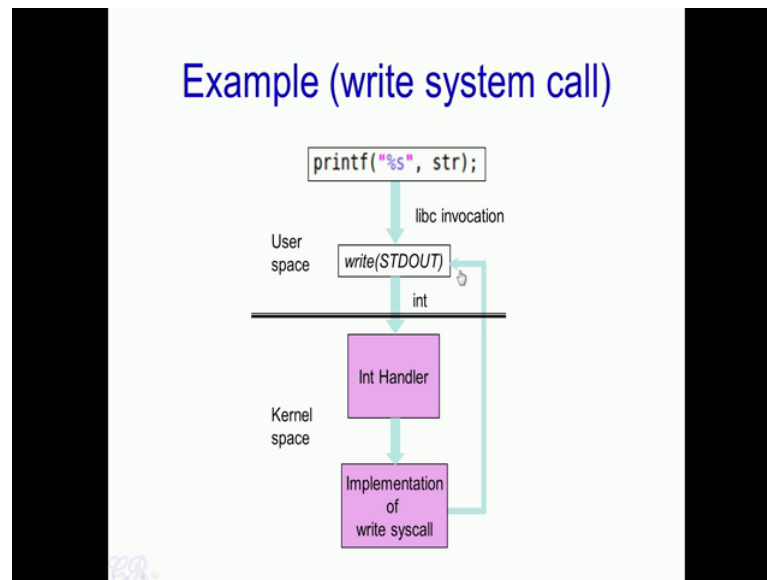
(Refer Slide Time: 01:54)



So, where is the software interrupt used? So, software interrupts are used to implement system calls. So, as we know user process could invoke a system call to perform some kernel operation, for example, it could be to read a file, or write a file, to print something to a monitor, or to send a packet through the network and so on. More specifically all operating systems implement system calls through one particular software interrupt.

For example, in the Linux operating systems, the software interrupt 128 is use to specify system calls. Therefore, in a Linux OS, if I have INT 128 which is executed in the user process, it would lead to an interrupt that occurs and cause the kernel or the operating system to execute, and thereafter the OS would execute code depending on the interrupt. In xv6, the software interrupt used to implement system calls is 64 or instruction like INT 64 in the user process would be meant to implement a system call.

(Refer Slide Time: 03:19)



So, to take an example, let us consider that our application has a `printf` statement present in it. So, `printf` would print this string to the standard output, which typically is the monitor. Now `printf` is a function present in the library `libc`, and it would cause the `libc` function to be invoked. Now in the `libc` function there is a call to the `write` system call with the specifier `STDOUT`. So, the `STDOUT` here is the file descriptor; and it is a special file descriptor which is meant for the standard output or the monitor.

In the `write` function, it would invoke `INT 64` in `xv6` or `INT 128` in `Linux` and cause a software interrupt to occur. So, the software interrupt as we know would cause the transformation from the user space to the kernel space and would it would result in the operating system executing. The OS would then determine that the interrupt was in fact due to a system call and then it would determine what system call it was from; in this case, it was a from a `write` system call and it was a write to the `STDOUT` - the standard output.

The operating system would then invoke the handler for the `write` system call, and this handler would take care of communicating with the various devices such as the video card to display the string onto the monitor. So, after this handler completes execution, the `IRET` instruction is executed which would result in a transformation back from kernel space to the user space and the program will continue to execute.

(Refer Slide Time: 05:31)

System Calls in xv6

System call	Description
fork()	Create process
exit()	Terminate current process
wait()	Wait for a child process to exit
kill(pid)	Terminate process pid
getpid()	Return current process's id
sleep(n)	Sleep for n seconds
exec(filename, *argv)	Load a file and execute it
sbrk(n)	Grow process's memory by n bytes
open(&name, flags)	Open a file; flags indicate read/write
read(id, buf, n)	Read n bytes from an open file into buf
write(id, buf, n)	Write n bytes to an open file
close(id)	Release open file id
dup(id)	Duplicate fd
pipe(p)	Create a pipe and return fd's in p
chdir(dirname)	Change the current directory
mknod(dirname)	Create a new directory
mknod(name, major, minor)	Create a device file
fstat(id)	Return info about an open file
link(f1, f2)	Create another name (f2) for the file f1
unlink(filename)	Remove a file

How does the OS distinguish between the system calls?

45

Now, typically operating systems support several different types of system calls. So, this particular table over here shows the various system calls supported by xv6. So, in a previous video we had seen some of them for example, we had seen, fork, exit, wait and so on. And you are also familiar with several types of system calls such as open, read, write, close, change directory, make directory and so on. So, each of these system calls would be executed by having a software interrupt such as INT 64, because it is the xv6, so it is 64. So, each time any of these system calls are invoked by a user process, it would trigger the operating system to execute.

Now, the next obvious question that one would ask is from the OS prospective, how does the OS distinguish between the various system calls. So, we mention that all the system function calls were used either INT 64 for xv6, and INT 128 for Linux, so how does the OS determine whether the system call was with respect to fork, wait, sleep, exit, and so on. Essentially this distinguisher comes from the user process itself.

(Refer Slide Time: 07:09)

System Call Number

System call number used to distinguish between system calls

System call number

```
mov x, %eax
INT 64
```

Based on the system call number the corresponding syscall handler is invoked

System call numbers	System call handlers
#define SYS_fork 1	[SYS_fork] sys_fork,
#define SYS_exit 2	[SYS_exit] sys_exit,
#define SYS_wait 3	[SYS_wait] sys_wait,
#define SYS_pipe 4	[SYS_pipe] sys_pipe,
#define SYS_read 5	[SYS_read] sys_read,
#define SYS_kill 6	[SYS_kill] sys_kill,
#define SYS_exec 7	[SYS_exec] sys_exec,
#define SYS_fstat 8	[SYS_fstat] sys_fstat,
#define SYS_chdir 9	[SYS_chdir] sys_chdir,
#define SYS_dup 10	[SYS_dup] sys_dup,
#define SYS_getpid 11	[SYS_getpid] sys_getpid,
#define SYS_sbrk 12	[SYS_sbrk] sys_sbrk,
#define SYS_sleep 13	[SYS_sleep] sys_sleep,
#define SYS_uptime 14	[SYS_uptime] sys_uptime,
#define SYS_open 15	[SYS_open] sys_open,
#define SYS_write 16	[SYS_write] sys_write,
#define SYS_mknod 17	[SYS_mknod] sys_mknod,
#define SYS_unlink 18	[SYS_unlink] sys_unlink,
#define SYS_link 19	[SYS_link] sys_link,
#define SYS_mkdir 20	[SYS_mkdir] sys_mkdir,
#define SYS_close 21	[SYS_close] sys_close,

ref: syscall.h, syscall() in syscall.c

46

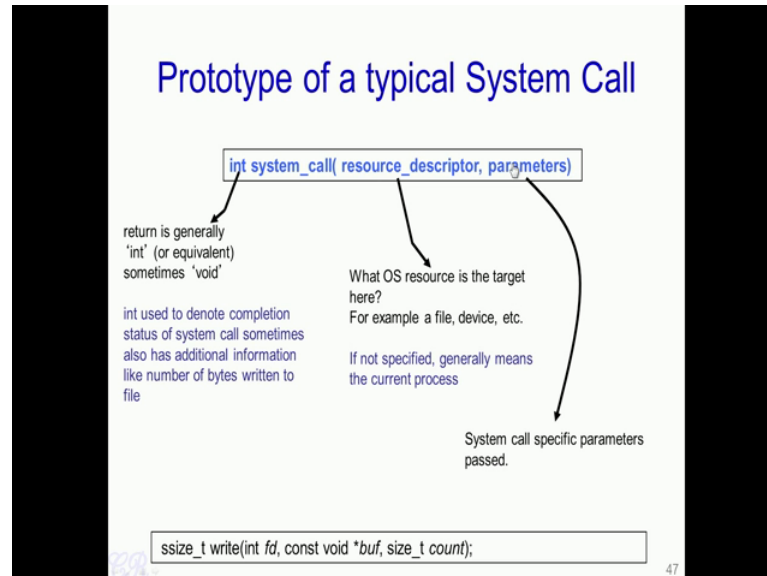
What happens is that before the INT 64 instruction, the user process will move a system call number to the eax register. So, for example, this instruction `mov x comma eax` will move the system call number to the eax register. Now each system call in the operating system will have a unique number system call number. Now the operating system when triggered by the INT instruction would look up the eax register, and then determine what system call was invoked. For example, in xv6, if we look up these particular header files, we would see the various system call numbers defined. For example, over here, we have each system call given a specific number example `sys underscore fork` given 1, `sys underscore exit` giving 2 and so on.

Now when the OS gets trigger due to the INT 64 instruction getting executed, the OS will determine the system call using this system call numbers and then invoke the corresponding system call handler. So, each of the system calls also have a corresponding system call handler. This is shown over here, corresponding to each of the system call numbers `sys underscore fork` that is 1, `sys underscore exit` it is 2 and so on.

So, these are system call functions present in the operating system which gets triggered based on the type of the system call. For example, if `eax` had a value of 11, the operating system will look in to the `eax` register and determine that this corresponded to the `get PID` system call being invoked. And then it would look into this particular table and see

that the get PID system call is handled by this system sys underscore get PID, and therefore it will then invoke this sys underscore get PID function.

(Refer Slide Time: 09:36)



Now, let us look at the typical prototype of a system call. So, a topical system call is as shown over here. So, of course, it has a system call name that is a function name, and then it is passed some resource descriptor and parameters and typically would return an integer. So, the resource descriptor specifies what operating system resource is the target here, for example, it could be a file or a device; and as we have seen in the previous slides it could also specify a particular monitor, for example, if the resource descriptor is STDOUT then the resource in use here is the monitor.

So, some system calls also do not specify this resource descriptor; in such a case, the system call is meant for that resource itself, for example, if we use the sleep system call, we only specify the time and no specific descriptor such as the file, device and so on. This means that the current process wants to sleep for that given interval time.

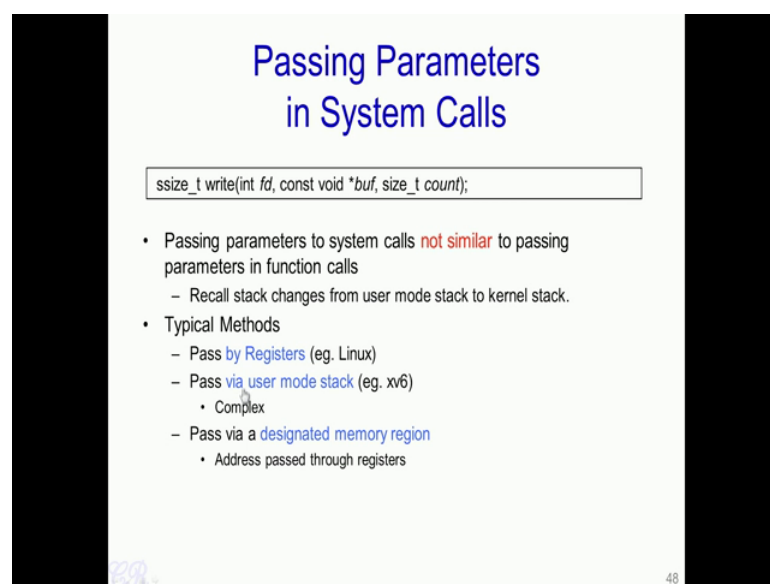
The next part is the parameters. So, these parameters are specific for the system call, for example, if we invoke read, write, open or close or any other system call, the parameters specified here is going to be very specific for each of these system calls. For example, the write system call has the parameter buf that is a void pointer and the count. So, the open or the close or any other system call would have different set of parameters. So, essentially these parameters are very specific to the type of the system call.

The return type is typically int or integer, and sometimes it is a void. So, int is typically used, because in this way the operating system will be able to send the completion status of the system call, whether it had executed successfully or it had failed and so on. So, sometimes the return is also used to specify certain specific information about the system call for example, in write the return is ssize_t which in fact, is typedef to integer and it specifies number of bytes that have been written to the file specified int fd. So, the return type could also vary depending on the type of system call.

The next thing what we will look at is how these parameters that is the resource descriptor and the parameters pass to the system call are sent to the kernel. So, note that system calls are invoked very differently from a standard function call. So, in a function call, as we know the instruction call would be used and the call would specify a address which is where the function would reside. And the various parameters for the call are passed through the local stack.

Similarly, the int return which is written from the function call would be return through the eax register. So, system calls on the other hand work very differently from function calls. So, as we have seen system calls invokes the kernel by the INT 0x80 instruction as in the case of xv6. So, how are the parameters such as the resource descriptor and the other parameters passed to the kernel?

(Refer Slide Time: 13:49)



Passing Parameters in System Calls

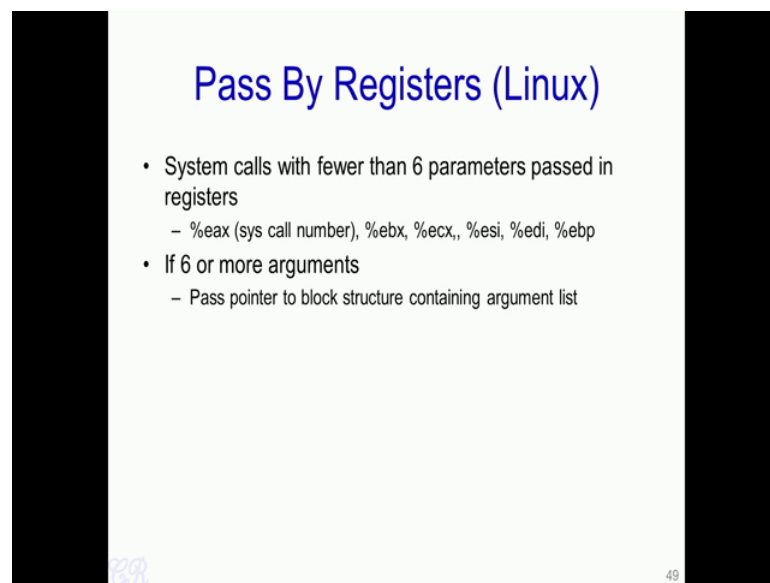
```
ssize_t write(int fd, const void *buf, size_t count);
```

- Passing parameters to system calls **not similar** to passing parameters in function calls
 - Recall stack changes from user mode stack to kernel stack.
- Typical Methods
 - Pass by Registers (eg. Linux)
 - Pass via user mode stack (eg. xv6)
 - Complex
 - Pass via a designated memory region
 - Address passed through registers

48

Essentially, there are three ways of doing so. The first is by pass by registers which is typically done in Linux; the second way is by passing through the user mode stack which is done in xv6; and the third way is by passing through a designated memory region. So, in this particular case, what is done is that in the user process itself, a designated region most likely in the heap would be used to save the various parameters that are needed to be passed to the system call; and the address to this region in the heap is passed through the registers. So, we will look at the other two cases that is pass by registers and pass via user mode stack in more detail.

(Refer Slide Time: 14:44)



The slide is titled "Pass By Registers (Linux)" in blue text. It contains a bulleted list of rules for system call parameter passing. The first rule states that system calls with fewer than 6 parameters are passed in registers, with a sub-bullet listing the registers: %eax (sys call number), %ebx, %ecx, %esi, %edi, %ebp. The second rule states that if there are 6 or more arguments, a pointer to a block structure containing the argument list is passed. The slide number "49" is visible in the bottom right corner.

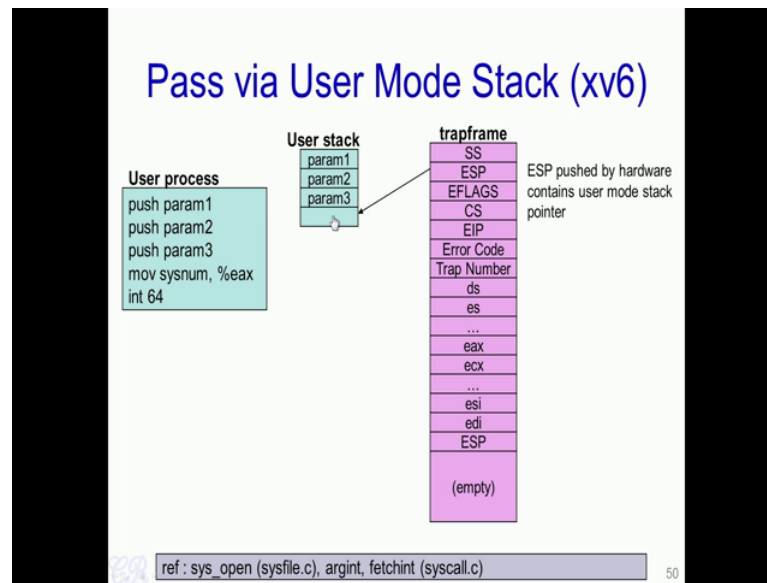
Pass By Registers (Linux)

- System calls with fewer than 6 parameters passed in registers
 - %eax (sys call number), %ebx, %ecx, %esi, %edi, %ebp
- If 6 or more arguments
 - Pass pointer to block structure containing argument list

49

Now pass by registers which is used by Linux system calls would use the register present in the processor to pass parameters to the kernel. So, we know we have already seen an example of this of how the eax register is used to pass the system call number from the user process to the operating system. In a similar way, other register such as the ebx, ecx, esi, edi, and ebp are used to pass the various parameters of the system call from the user process to the kernel. If the system call has more than 6 arguments, and then a pointer to a block structure containing the argument list is passed to the kernel.

(Refer Slide Time: 15:41)



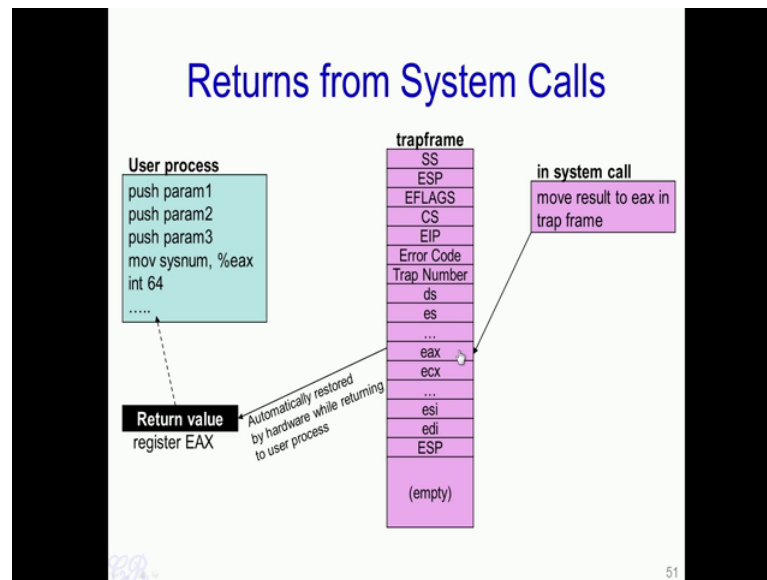
Now, let us look at the second case that is pass via the user mode stack, and this is what is done xv6. So, in this particular way, before the INT 64 instruction is present various parameters in the system call are pushed onto the stack. For example, if the system call had 3 parameters param 1, param 2, and param 3, so these three parameters are pushed into the user space stack and then the system call number as we have seen is moved in to the eax register. So, this here is the user space stack in of the user process containing the three parameters.

Now, when the INT instruction is executed, as we know, it triggers an interrupt causing the switch from the user space in to the kernel space. Also as a result of this interrupt execution, as we have seen, there is the switch in the stack from the user space stack to the kernel space stack. And what we have seen in the previous video that this kernel stack is used to create what is known as the trapframe. So, this trapframe is shown over here. So, what we have seen in the previous video that some of these entries in the trapframe are pushed into the stack automatically by the CPU. So, in particular, these registers specified in capitals are all pushed onto the kernel stack that is on to the trapframe by the CPU.

Now, the SS and ESP here is important for us. So, these are the stack segment and the stack pointer, and these registers correspond to the user space stack. So, as we know before the INT 64 has been executed, the last known stack pointer was pointing to this

particular location. And therefore, the contents of ESP will also point to this location in the user space stack. So, in this way, the kernel will then use the SS and the ESP from the trapframe to determine the various parameters for the system call.

(Refer Slide Time: 18:22)



The next thing we will look at is how the return value is passed from the system call back to the user process. Again we will recollect that the entire reason for creating this trapframe in the kernel stack for the process is due to the reason that when the interrupt or the system call completes its execution. The entire state in the trapframe is restored back into the corresponding CPU registers, and it could result in the user process continuing to execute from where it had stopped, and also the context of the user process is restored with the help of the trapframe.

Now, in order to return a value from the system call this is executing in the kernel space back to the user space. So, what is done is that the eax register in the trapframe is modified; essentially, we had seen that the eax register because of this particular instruction would contain the system call number.

Now this system call number is overridden by the return value of the system call. So, this could be a negative number like minus 1 or a positive number as we have seen in the earlier slide. So, now when the system call executes and completes its execution and the context is transferred back to the user process, the entire trapframe including the new value of eax in the trapframe is restored back in to the registers of the CPU. The process

continues to execute from this particular instruction with the new value of `eax`, which contains the return from the system call.

Thank you.