Lab 3 : SYSTEM CALL (CÁC DỊCH VỤ HỆ THỐNG)

1. **Chuẩn đầu ra :** Sau bài này, người học có thể :
   - ✓ Sử dụng một số dịch vụ hệ thống trong việc lập trình.
2. **Chuẩn bị :** Đọc trước phần lý thuyết về System call.
3. **Phương tiện :**
   - ✓ Máy vi tính.
   - ✓ Chương trình hợp dịch nasm.
4. **Thời lượng : 4 tiết**
5. **Tóm tắt lý thuyết**

Syscalls

Syscalls are the interface between user programs and the Linux kernel. They are used to let the kernel perform various system tasks, such as file access, process management and networking. In the C programming language, you would normally call a wrapper function which executes all required steps or even use high-level features such as the standard IO library.

On Linux, there are several ways to make a syscall. This page will focus on making syscalls by calling a software interrupt using int $0x80 (x86 and x86_64) or syscall (x86_64). This is an easy and intuitive method of making syscalls in assembly-only programs

Making a syscalls

To make a syscall using an interrupt, you have to pass all required information to the kernel by copying them into general purpose registers. Each syscall has a fixed number (note the numbers differ between int $0x80 and syscall in the following text). You specify the syscall by writing the number into the eax/rax register and pass the parameters by writing them in the appropriate registers before making the actual calls. Parameters are passed in the order they appear in the function signature of the corresponding C wrapper function.

After everything is set up correctly, you call the interrupt using int $0x80 or syscall and the kernel performs the task.

The return or error value of a syscall is written to eax or rax.

The kernel uses its own stack to perform the actions. The user stack is not touched in any way.

int 0x80

On both Linux x86 and Linux x86_64 systems you can make a syscall by calling interrupt 0x80 using the int $0x80 command. Parameters are passed by setting the general purpose registers as following:

Syscall #	Param 1	Param 2	Param 3	Param 4	Param 5	Param 6
eax	ebx	ecx	edx	esi	edi	ebp

The return value is in the eax register

Syscall

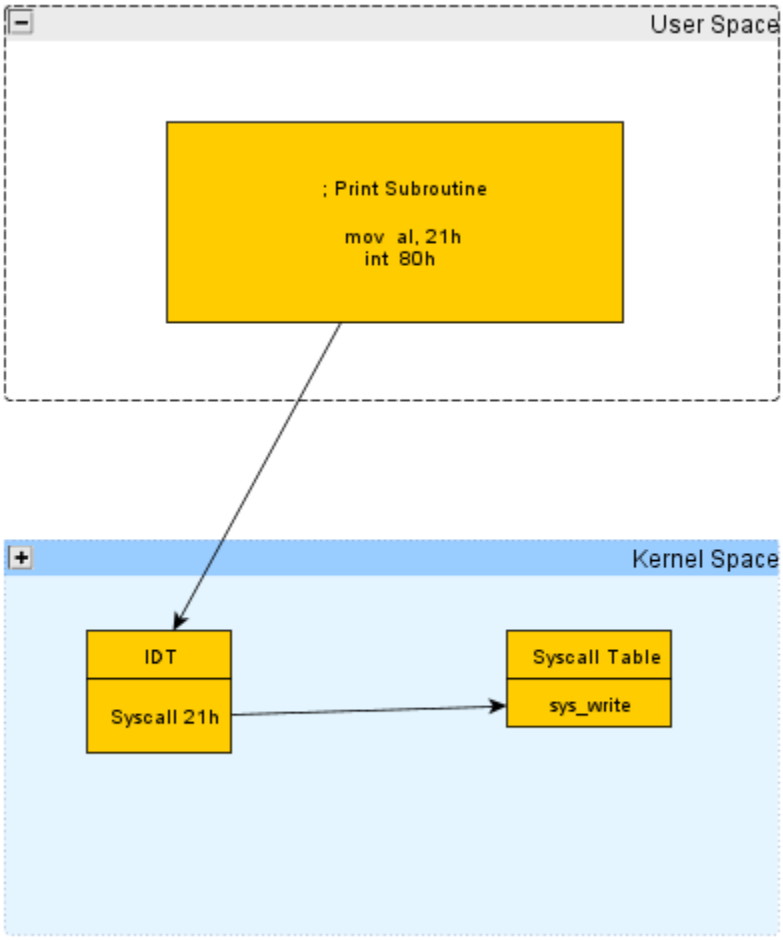
The x86_64 architecture introduced a dedicated instruction to make a syscall. It does not access the interrupt descriptor table and is faster. Parameters are passed by setting the general purpose registers as following

Syscall #	Param 1	Param 2	Param 3	Param 4	Param 5	Param 6
rax	rdi	rsi	rdx	rcx	r8	r9

The return value is in the rax register

Systemcalls in Linux

Simply Systemcalls (or Syscalls) are the way to get service from OS kernel. For example, you want to write simple hello world program in assembly. You can go ahead and write your own code for displaying output on computer-screen. Though possible, it is tedious task. so, you use WRITE syscall to display output on the screen.



In above figure, we can observe how the syscall WRITE works.

1. From user space (where our program is running), value 21h is loaded in AL register, 21h (33 in decimal) is a syscall for print on screen.
2. On executing int 80h, control is passed to Interrupt Description Table (IDT). In IDT, entry for provided syscall number is searched.
3. If syscall is correct, then control is passed to Syscall table where the actual subroutine executed.

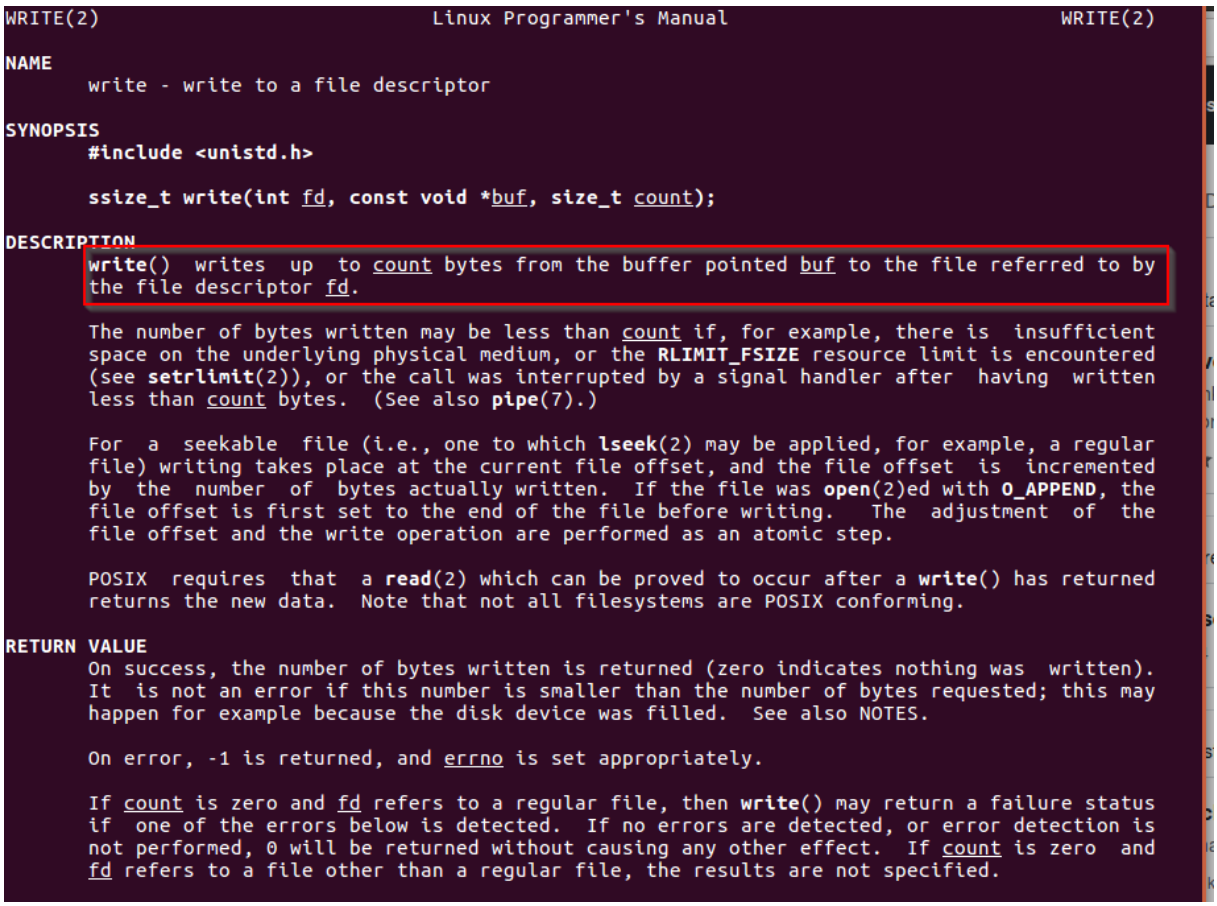
/usr/src/linux-headers-4.13.0-36/arch/sh/include/uapi/asm/unistd_32.h

Linux Programmer Manual

To use syscalls in code, we need to understand how they actually works. i.e. parameters to be passed, datatypes etc. Linux programmer manual will help us with this. Lets see an example for WRITE syscall.

$ man 2 write

Result is shown in below screenshot:



```
WRITE(2)                        Linux Programmer's Manual                       WRITE(2)

NAME
       write - write to a file descriptor

SYNOPSIS
       #include <unistd.h>

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

DESCRIPTION
       write()  writes  up  to count bytes from the buffer pointed buf to the file referred to by
       the file descriptor fd.

       The number of bytes written may be less than count if, for example, there is  insufficient
       space on the underlying physical medium, or the RLIMIT_FSIZE resource limit is encountered
       (see setrlimit(2)), or the call was interrupted by a signal handler after  having  written
       less than count bytes.  (See also pipe(7).)

       For  a  seekable  file (i.e., one to which lseek(2) may be applied, for example, a regular
       file) writing takes place at the current file offset, and the file offset  is  incremented
       by  the  number  of  bytes actually written.  If the file was open(2)ed with O_APPEND, the
       file offset is first set to the end of the file before writing.   The  adjustment  of  the
       file offset and the write operation are performed as an atomic step.

       POSIX  requires  that  a read(2) which can be proved to occur after a write() has returned
       returns the new data.  Note that not all filesystems are POSIX conforming.

RETURN VALUE
       On success, the number of bytes written is returned (zero indicates nothing was  written).
       It  is not an error if this number is smaller than the number of bytes requested; this may
       happen for example because the disk device was filled.  See also NOTES.

       On error, -1 is returned, and errno is set appropriately.

       If count is zero and fd refers to a regular file, then write() may return a failure status
       if  one of the errors below is detected.  If no errors are detected, or error detection is
       not performed, 0 will be returned without causing any other effect.  If count is zero  and
       fd refers to a file other than a regular file, the results are not specified.
```

Parameters are generally passed in the registers as follow:

- EAX = Syscall number
- EBX = Param 1
- ECX = Param 2
- EDX = Param 3
- ESI = Param 4
- EDI = Param 5

Of-course we will look into this in more details while writing actual code.

Syscalls are generally OS specific, hence you need to find out the list of the syscalls available in OS. In our case, location is as below:

Linux system calls are called in exactly the same way as DOS system calls:

1. You put the system call number in EAX (we're dealing with 32-bit registers here, remember)
2. You set up the arguments to the system call in EBX, ECX, etc.
3. You call the relevant interrupt (for DOS, 21h; for Linux, 80h)
4. The result is usually returned in EAX

There are six registers that are used for the arguments that the system call takes. The first argument goes in EBX, the second in ECX, then EDX, ESI, EDI, and finally EBP, if there are so many. If there are more than six arguments, EBX must contain the memory location where the list of arguments is stored - but don't worry about this because it's unlikely that you'll use a syscall with more than six arguments. The wonderful thing about this scheme is that Linux uses it consistently – all system calls are designed this way, there are no confusing exceptions.

Some example code always helps:

```
        mov     eax,1     ; The exit syscall number
        mov     ebx,0     ; Have an exit code of 0
        int     80h       ; Interrupt 80h, the thing that pokes the kernel and says, "Yo, do this"
```

But how do you find out what these system calls are, and what they do, and what arguments they take? Firstly, all the syscalls are listed in /usr/include/asm/unistd.h, together with their numbers (the value to put in EAX before you call int 80h). However, for your convenience you can simply find them in this Linux System Call Table, together with some other useful information (eg. what arguments they take). Take a look at the list of syscalls – there are things like sys_write (4), sys_nice (34) and of course sys_exit (1). To find out just what these things do, you can look them up in the Linux manual pages (commonly called "the manpages"). That is what the next section is about.

6. Nội dung thực hành

6.1. "Hello World!" in Linux Assembly

- **Nhập chương trình sau vào**

```
section .data
      hello:     db 'Hello world!',10    ; 'Hello world!' plus a linefeed character
      helloLen:  equ $-hello ; Length of the 'Hello world!' string

section .text
      global _start

_start:
      mov eax,4            ; The system call for write (sys_write)
      mov ebx,1            ; File descriptor 1 - standard output
      mov ecx,hello        ; Put the offset of hello in ecx
      mov edx,helloLen     ; helloLen is a constant, so we don't need to say
                           ;  mov edx,[helloLen] to get it's actual value
      int 80h              ; Call the kernel
      mov eax,1            ; The system call for exit (sys_exit)
```

```
mov ebx,0              ; Exit with return code of 0 (no error)
int 80h
```

- The appropriate way to begin would be to print out "Hello world!" To print to the screen, we write to the special file called STDOUT (standard output), which is file descriptor 1

- **Lưu chương trình với tên hello.asm**

- **Biên dịch và liên kết**
  - If you don't have a terminal or console open, open one now.
  - Make sure you are in the same directory as where you saved hello.asm.
  - To assemble the program, type
    nasm -f elf hello.asm
    If there are any errors, NASM will tell you on what line you did what wrong.
  - Now type ld -s -o hello hello.o
    This will link the object file NASM produced into an executable file.
  - Run your program by typing ./hello

-

6.2. Bảng một số system call

rax	System call	rdi	rsi	rdx	r10	r8	r9
0	sys_read	unsigned int fd	char *buf	size_t count			
1	sys_write	unsigned int fd	const char *buf	size_t count			
2	sys_open	const char *filename	int flags	int mode			
3	sys_close	unsigned int fd					
4	sys_stat	const char *filename	struct stat *statbuf				

5	sys_fstat	unsigned int fd	struct stat *statbuf				

6.3. Write text to console

%rax	Name	Entry point	Implementation
1	write	sys_write	fs/read_write.c

%rdi	%rsi	%rdx
unsigned int fd	**const char __user *** buf	**size_t** count

So to write text to the console, we must do these things:

- Set **rax** to **1** to specify the "write" syscall
- Set **rdi** to **1** (the file descriptor for stdout, the console)
- **Push** the string onto the stack
- Set **rsi** to **rsp** (the stack pointer)
- Set **rdx** to the length of the string
- Call **syscall**
- **Example 1:** This is the simplest program I know in Assembler. It uses syscall once to print the letters 'ABCDEFGH'
- **Enter this code**

```
section .text
global _start
_start:
   mov  rax, 0x4142434445464748    ; 'ABCDEFGH'
   push rax
   mov  rdx, 0x8     ; length of string is 8 bytes
   mov  rsi, rsp     ; Address of string is RSP because
string is on the stack
   mov  rax, 0x1     ; syscall 1 is write
   mov  rdi, 0x1     ; stdout has a file descriptor of 1
   syscall           ; make the system call
```

- Lưu tập tin với tên abc.nasm
- Biên dịch chương trình
- Liên kết
- Chạy thử
- The program prints out the letters in reverse order, and then crashes with a "Segmentation fault" message, as shown below

- The program crashes at the end rather than exiting normally. To fix that, we need to add a second syscall, to "exit"
-

6.4. Exit from console

So to exit, we must do these things:

- Set **rax** to **0x3c** (60 in decimal) to specify the "exit" syscall
- Call **syscall**
- **Example 2 :** Add these lines at the bottom of the program:

```
mov  rax, 0x3c    ; syscall 3c is exit
syscall           ; make the system call
```

- **Lưu tập tin với tên abc2.nasm**
- **Biên dịch**
- **Liên kết**
- **Chạy thử**

6.5. "Hello World" Using a .data Section

- Enter this code

```
section .data
string1 db  "Hello World!",10   ; '10' at end is line feed
section .text
global _start
_start:
  mov  rdx, 0xd         ; length of string is 13 bytes
  mov  rsi, dword string1; set rsi to pointer to string
  mov  rax, 0x1                 ; syscall 1 is write
  mov  rdi, 0x1     ; stdout has a file descriptor of 1
  syscall                       ; make the system call
  mov  rax, 0x3c                ; syscall 3c is exit
  syscall                       ; make the system call
```

-

6.6. "Echo" Using a .data Section

- Enter this code

```
section .data
string1 db  "AAAABBBBCCX"; Reserve space for 10 characters
section .text
global _start
_start:
```

```
mov  rdx, 0xa                ; length of string is 10 bytes
mov  rsi, dword string1      ; set rsi to pointer to string
mov  rax, 0x0                ; syscall 0 is read
mov  rdi, 0x0              ; stdin has a file descriptor of 0
syscall                      ; make the system call
mov  rdx, 0xa                ; length of string is 10 bytes
mov  rsi, dword string1      ; set rsi to pointer to string
mov  rax, 0x1                ; syscall 1 is write
mov  rdi, 0x1              ; stdout has a file descriptor of 1
syscall                      ; make the system call
mov  rax, 0x3c               ; syscall 3c is exit
syscall                      ; make the system call
```

- Lưu chương trình
- Biên dịch
- Liên kết
- Chạy thử

6.7. Sloppy Caesar Cipher

This program is like "Echo" but it increments each byte of the input before printing it out. It doesn't work correctly for "Z" or "z", which should wrap around to "A" or "a", and has other flaws

- Enter this code

```
section .data
string1 db  "AAAABBBB"     ; Reserve space for 8 characters
section .text
global _start
_start:
mov  rdx, 0x8                ; length of string is 8 bytes
mov  rsi, dword string1      ; set rsi to pointer to string
mov  rax, 0x0                ; syscall 1 is read
mov  rdi, 0x0              ; stdin has a file descriptor of 0
syscall                      ; make the system call
mov  rbx, dword string1      ; set rbx to pointer to string
mov  rcx, [rbx]              ; Put string value into rcx
add  rcx, 0x0101010101010101 ; Add 1 to each byte, not fixing
rollover
mov  [rbx], rcx              ; Put modified byte on string
mov  rdx, 0x8                ; length of string is 8 bytes
mov  rsi, dword string1      ; set rsi to pointer to string
mov  rax, 0x1                ; syscall 1 is write
mov  rdi, 0x1            ; stdout has a file descriptor of 1
syscall                      ; make the system call
mov  rax, 0x3c               ; syscall 3c is exit
```

```
    syscall                     ; make the system call
```

6.8.