KOM206 Laboratory (Dept. of Computer Science, IPB) Module 09 - Mar 22, 2015

System-call Utilization

Lecturers: ARD, SWJ *Lab tutors: VDE, WUL*

Objectives

1. able to utilize system calls (in Linux)
2. able to perform IO operation using stdin and stdout
3. able to write and read text files

1 Theory

A system call is essentially a function call which changes the CPU into kernel mode and executes a function which is part of the kernel. When you run a process on Linux it runs in user mode which means that it is limited to executing only "safe" instructions. It can move data within the program, do arithmetic, do branching, call functions, etc, but there are instructions which your program can't do directly. For example it would be unsafe to allow any program to read or write directly to the disk device, so this is prevented by preventing user programs from executing input or output instructions. Another prohibited action is directly setting page mapping registers.

When a user program needs to do something like open a disk file, it makes a system call. This changes the CPU's operating mode to kernel mode where the CPU can execute input and output instructions. The kernel open function will verify that the user program has permission to open the file and then open it, performing any input or output instructions required on behalf of the program. Notice that the Linux system call interface is different for 32 bit mode and 64 bit mode. Under 64 bit Linux the 32 bit interface is still available to support 32 bit applications and this will work to some extent for 64 bit programs.

You need to take the following steps for using Linux system calls in your program:

1. Put the system call number in the EAX register.
2. Store the arguments to the system call in the following 6 registers, namely: EBX, ECX, EDX, ESI, EDI, and EBP. These registers take the consecutive arguments, starting with the EBX register. If there are more than six arguments then the memory location of the first argument is stored in the EBX register.
3. Call the relevant interrupt, i.e. INT 0x80. INT means interrupt, and the number 0x80 is the interrupt number. An interrupt transfers the program flow to whomever is handling that interrupt, which is interrupt 0x80 in this case. In Linux, 0x80 interrupt handler is the kernel, and is used to make system calls to the kernel by other programs. The kernel is notified about which system call the program wants to make, by examining the value in the register EAX. Each system call have different requirements about the use of the other registers, see figure 1. The result of a system call is returned in the EAX register.

Show 10 entries Search:

#	Name	Registers						Definition
		eax	ebx	ecx	edx	esi	edi	
0	sys_restart_syscall	0x00	-	-	-	-	-	kernel/signal.c:2058
1	sys_exit	0x01	int error_code	-	-	-	-	kernel/exit.c:1046
2	sys_fork	0x02	struct pt_regs *	-	-	-	-	arch/alpha/kernel/entry.S:716
3	sys_read	0x03	unsigned int fd	char __user *buf	size_t count	-	-	fs/read_write.c:391
4	sys_write	0x04	unsigned int fd	const char __user *buf	size_t count	-	-	fs/read_write.c:408
5	sys_open	0x05	const char __user *filename	int flags	int mode	-	-	fs/open.c:900
6	sys_close	0x06	unsigned int fd	-	-	-	-	fs/open.c:969
7	sys_waitpid	0x07	pid_t pid	int __user *stat_addr	int options	-	-	kernel/exit.c:1771
8	sys_creat	0x08	const char __user *pathname	int mode	-	-	-	fs/open.c:933
9	sys_link	0x09	const char __user *oldname	const char __user *newname	-	-	-	fs/namei.c:2520

Showing 1 to 10 of 338 entries First Previous 1 2 3 4 5 Next Last

Figure 1: Linux system call references, more at http://syscalls.kernelgrok.com/

1.1 IO Management

The system considers any input or output data as stream of bytes. There are three standard file streams: Standard input (stdin), Standard output (stdout) and Standard error (stderr). Refer to task01 anda task02 for using stdin and stdout.

A file pointer specifies the location for a subsequent read/write operation in the file in terms of bytes. Each file is considered as a sequence of bytes. Each open file is associated with a file pointer that specifies an offset in bytes, relative to the beginning of the file. When a file is opened, the file pointer is set to zero.

A file descriptor is a 16-bit integer assigned to a file as a file id. When a new file is created, or an existing file is opened, the file descriptor is used for accessing the file. File descriptor of the standard file streams: stdin, stdout and stderr are 0, 1 and 2 respectively.

Creating and Opening a File: For creating and opening a file, perform the following tasks: 1) Put the system call sys_creat() number 8, in the EAX register, 2) Put the filename in the EBX register, 3) Put the file permissions in the ECX register. The system call returns the file descriptor of the created file in the EAX register, in case of error, the error code is in the EAX register.

Opening an Existing File: For opening an existing file, perform the following tasks: 1) Put the system call sys_open() number 5, in the EAX register, 2) Put the filename in the EBX register, 3) Put the file access mode in the ECX register, 4) Put the file permissions in the EDX register. The system call returns the file descriptor of the created file in the EAX register, in case of error, the error code is in the EAX register. Among the file access modes, most commonly used are: read-only (0), write-only (1), and read-write (2).

Reading from a File: For reading from a file, perform the following tasks: 1) Put the system call sys_read() number 3, in the EAX register, 2) Put the file descriptor in the EBX register, 3) Put the pointer to the input buffer in the ECX register, 4) Put the buffer size, i.e., the number of bytes to read, in the EDX register. The system call returns the number of bytes read in the EAX register, in case of error, the error code is in the EAX register.

Writing to a File: For writing to a file, perform the following tasks: 1) Put the system call sys_write() number 4, in the EAX register, 2) Put the file descriptor in the EBX register, 3) Put the pointer to the output buffer in the ECX register, 4) Put the buffer size, i.e., the number of bytes to write, in the EDX register. The system call returns the actual number of bytes written in the EAX register, in case of error, the error code is in the EAX register.

Closing a File: For closing a file, perform the following tasks: 1)Put the system call sys_close() number 6, in the EAX register 2)Put the file descriptor in the EBX register. The system call returns, in case of error, the error code in the EAX register.

Updating a File For updating a file, perform the following tasks: 1) Put the system call sys_lseek () number 19, in the EAX register, 2) Put the file descriptor in the EBX register, 3) Put the offset value in the

ECX register, 4) Put the reference position for the offset in the EDX register. The reference position could be: Beginning of file - value 0; Current position - value 1; End of file - value 2. The system call returns, in case of error, the error code in the EAX register.

2 Including Other Files

Using a very similar syntax to the C preprocessor, NASM's preprocessor lets you include other source files into your code. This is done by the use of the %include directive:

```
%include "macros.mac"
```

will include the contents of the file macros.mac into the source file containing the Include files are searched for in the current directory (the directory youre in when you run NASM, as opposed to the location of the NASM executable or the location of the source file), plus any directories specified on the NASM command line using the i option. The standard C idiom for preventing a file being included more than once is just as applicable in NASM: if the file macros.mac has the form

```
%ifndef MACROS\_MAC
%define MACROS\_MAC
; now define some macros
%endif
```

then including the file more than once will not cause errors, because the second time the file is included nothing will happen because the macro MACROS_MAC will already be defined. You can force a file to be included even if there is no %include directive that explicitly includes it, by using the -p option on the NASM command line.

3 Tasks

1. Write an assembly program to write a string to stdout.
2. Write an assembly program to receive inputs from a keyboard, then print the input (as a string) to stdout.
3. Write an assembly program to create a file and write some messages to it.
4. Write an assembly program to open a file and read its content as well as print it to stdout. Handle a case when the file does not exist.
5. Good programming practice. Modularize some procedure you have written to some individual files e.g. print.asm, file_io.asm, etc. Spend time to write good documentation of each procedure including a brief description, input arguments, output variables.
6. Write an assembly program that convert a string to an integer.
7. Write an assembly program that convert an integer to a string.
8. Write an assembly language program not embedded in a C program that uses a procedure to obtain the area of a triangle from two integers that are entered from the keyboard. Enter several sets of single-digit numbers for the base and height and display the areas.
9. Write a procedure that produces N values in the Fibonacci number series and stores them in an array of doubleword. Input parameters should be a pointer to an array of doubleword, a counter of the number of values to generate. Write a test program that calls your procedure, passing N = 47. The first value in the array will be 1, and the last value will be 2,971,215,073.

4 Homework

1. What happens if we forget to write RET in a procedure definition?

5 Miscellany

The content is mainly based on the previous lab module and the following resources: [1], of [2] and [3]. (compiled on 17/04/2015 at 8:36pm)

References

[1] R. Seyfarth, *Introduction to 64 Bit Intel Assembly Language Programming for Linux*. CreateSpace, 2012.

[2] K. Irvine, *Assembly Language for X86 Processors*. Pearson Education, Limited, 2011. [Online]. Available: http://books.google.co.id/books?id=0k20RAAACAAJ

[3] J. Cavanagh, *X86 Assembly Language and C Fundamentals*. CRC Press, 2013.