DEPARTMENT OF INFORMATION TECHNOLOGY

IT 253Operating Systems Lab

LAB1: 21/02/2023

Objective

To understand the system call and creating child process.

System Calls: A **system call** is a procedure that provides the interface between a process and the operating system. It is the way by which a computer program requests a service from the kernel of the operating system.

Different operating systems execute different system calls. In Linux, making a system call involves transferring control from unprivileged user mode to privileged kernel mode; the details of this transfer vary from architecture to architecture. The libraries take care of collecting the system-call arguments and, if necessary, arranging those arguments in the special form necessary to make the system call.

Following are the various categories of system calls

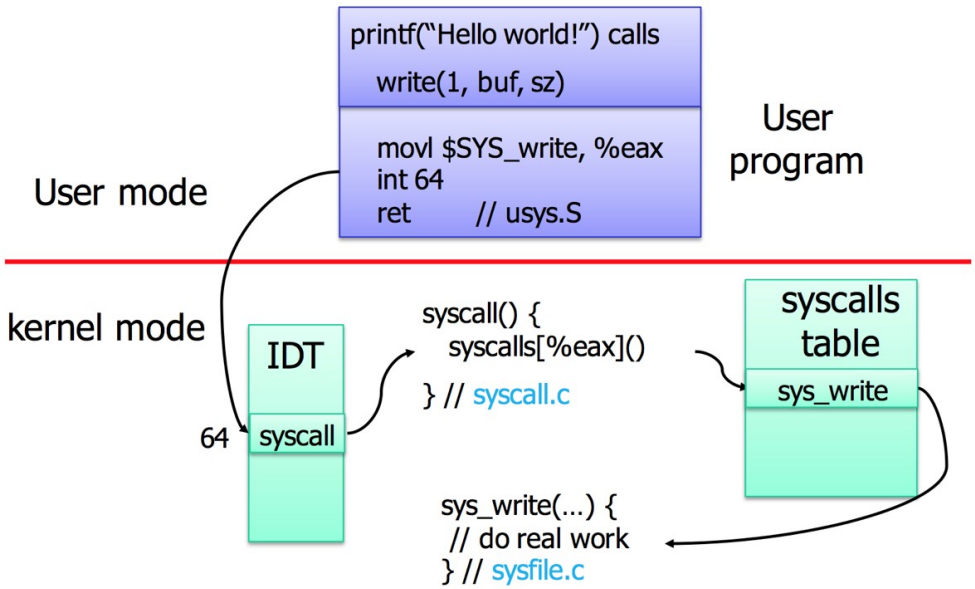
1. Process control
2. File management
3. Device management
4. Information Maintenance
5. Communication

Exercise 1: In the following example the syscall() function provides interface for running the System command in kernel mode and provides the result.

At the level of assembly language, a system call involves executing a trap instruction. In modern x86 code, the trap instruction is **syscall**, which acts in a manner analogous to call. Instead of jumping to a function within the same program, though, **syscall** triggers a mode switch and jumps to a routine in the kernel portion of memory. The kernel validates the system call parameters and checks the process's access permissions. For instance, if the system call is a request to write to a file, the kernel will determine whether the user running the program is allowed to perform this action. Once the kernel has finished performing the system call, it uses the **sysret** instruction, which performs a role similar to the standard **ret** instruction. The difference is that **sysret** also changes the privilege level, returning the system to user mode.

syscall(SYS_call,arg1,arg2,...);

The first argument, SYS_call, is a definition that represents the number of the system call. When you include sys/syscall.h, these are included. The first part is SYS_ and the second part is the name of the system call. Arguments for the call go into arg1, arg2 above. Some calls require more arguments, and they'll continue in order from their man page. Remember that most arguments, especially for returns, will require pointers to char arrays or memory allocated via the malloc function.



When a system call is executed, it is typically treated by the hardware as a software interrupt. Control passes through the **interrupt vector** to a service routine in the operating system, and the

mode bit is set to kernel mode. The **system-call service routine** is a part of the operating system. The kernel examines the interrupting instruction to determine what system call has occurred; a parameter indicates what type of service the user program is requesting. Additional information needed for the request may be passed in registers, on the stack, or in memory (with pointers to the memory locations passed in registers). The kernel verifies that the parameters are correct and legal, executes the request, and returns control to the instruction following the system call.

Compile it with gcc and execute

gcc filename.c -o excercise1

./excercise1

Observe the result and mention the CPU and NUMA node. Execute the same program 4-5 times and observe and analyse the result. What is your observation.

```
#include <stdio.h>

#include <unistd.h>

#include <sys/syscall.h>

#include <sys/types.h>


int main() {

    unsigned cpu, node;

    // Get current CPU core and NUMA node via system call

   // Note this has no glibc wrapper so we must call it directly

   syscall(SYS_getcpu, &cpu, &node, NULL);

    // Display information

   printf("This program is running on CPU core %u and NUMA node %u.\n\n", cpu,
node);

    return 0;  }
```

Exercise 2: **The system calls can be used to invole writing and exiting in the program as follows: Execute and compare with printf().**

```
/* Using syscall() in C to invoke Linux system calls for writing and exiting
 */
#include <unistd.h>
char *message = "Hello, world\n";
int main (void)
{
  syscall (1, 1, message, 13);
  syscall (60, 0);
  /* should never reach here */
  return 0;
}
```

syscall (1, 1, message, 13); //Here Syscal number is 1. file descriptor is 1 for stdout, message is the data to be written, 13 indicates size.

Syscall (60, 0); //This is to exit.

For details on how read and write operations happen in file refer following link.

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/fs/read_write.c

Exercise 3: Process control system calls in Unix are fork(), exit() and wait().

Fork() creates child process. When fork() is executed it creates one parent and one child process.

Note down the output of following program. How many times hello World is printed?

```
//fork() and executes all code.
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    // make two process which run same
```

```
    // program after this instruction
    fork();

    printf("Hello world!\n");
    return 0;
}
```

Calculate how many times Hello World in printed in following case.

```
#include <stdio.h>
#include <sys/types.h>
int main()
{
    fork();    // fork1
    fork();    //fork 2
    fork();    // fork 3
    printf("Hello World\n");
    return 0;
}
```

Analyze number of process created by each fork(). For N number of fork() called like this, how many child process gets created?

When a child process is created, fork() returns 0 in the child process and positive integer in the parent process.

Exercise 4: Some commands to identify the status of the process

a) **$ps**: report a snapshot of current processes

b) **$ps -eo pid,tid,class,rtprio,ni,pri,psr,pcpu,stat,wchan:14,comm**

This lists all the processes with user format

c) **$ps -U root -u root u**

To see every process running on root in user format