SYSTEM CALLS

Jevitha K.P

Secure Coding Lab 3

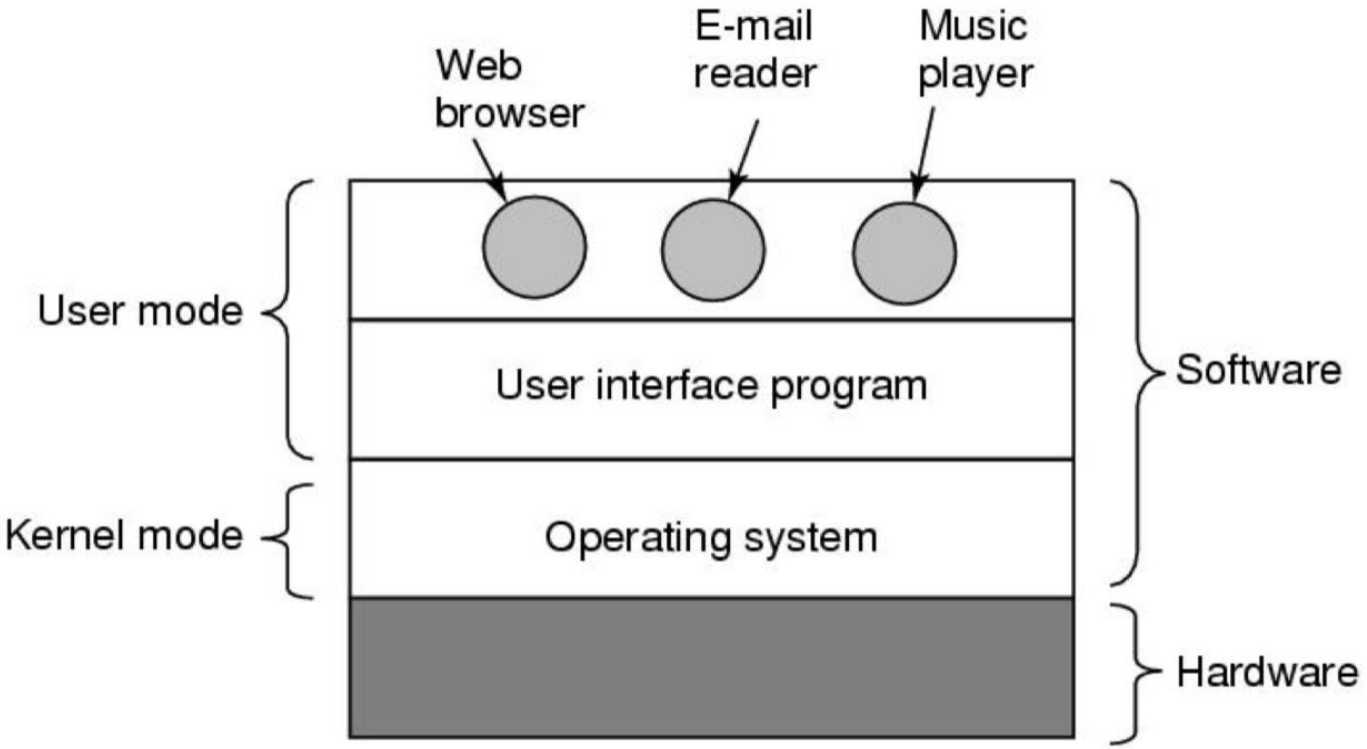
SYSTEM CALLS

- System Calls is the mechanism used by an application program to request services from the operating system (kernel) such as opening a file, forking to a new process, or requesting more memory.
- It is the real process to kernel communication mechanism used by all processes
- System calls often use a special machine code instruction which causes the processor to change mode (e.g. to "supervisor mode" or "protected mode").
- This allows the OS to perform restricted actions such as accessing hardware devices or the memory management unit.

SYSTEM CALLS

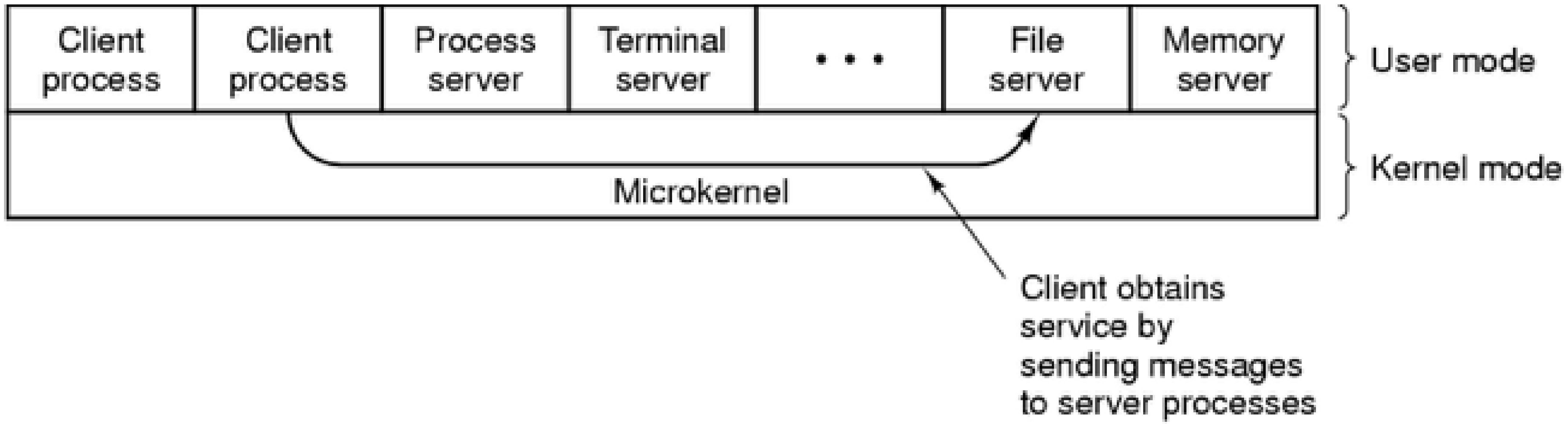
- A process, in general, is not supposed to be able to access the kernel.
- It can't access kernel memory and can't call kernel functions , which is enforced by the hardware of the CPU (hence, called 'protected mode').
- System calls are an exception to this general rule.

USER MODE VS KERNEL MODE



CLIENT-SERVER MODEL INSIDE OS

- The OS may be split into
  - a **kernel** which is always present
  - various **system programs** use facilities provided by the kernel to perform higher-level house-keeping tasks,
  - often acting as servers in a client-server relationship.



SYSTEM CALLS IN LINUX

- In Linux, System call service is provided by Linux kernel.
- In C programming, functions defined in libc library provides a wrapper for many system calls.
- It is also possible to invoke syscall() function directly.
- Each system call has a function number defined in <syscall.h> or <unistd.h>.
- Internally, system call is invoked by software interrupt 0x80 to transfer control to the kernel.
- System call table is defined in Linux kernel source file "arch/i386/kernel/entry.S ".

STEPS IN MAKING A SYSTEM CALL

- There are several steps in making the system call :
  - The calling process fills the registers with the appropriate values
  - It then sends software interrupt 0x80, which jumps to a previously defined location in the kernel (the location is readable by user processes,but not writable by them).
- The hardware knows that once we jump to this location,
  - we are no longer running in restricted user mode,
  - but as the operating system kernel --- and therefore allowed to do the task we want.

STEPS IN MAKING A SYSTEM CALL

read (fd, buffer, nbytes)

1-3 - push 3 parameters on stack

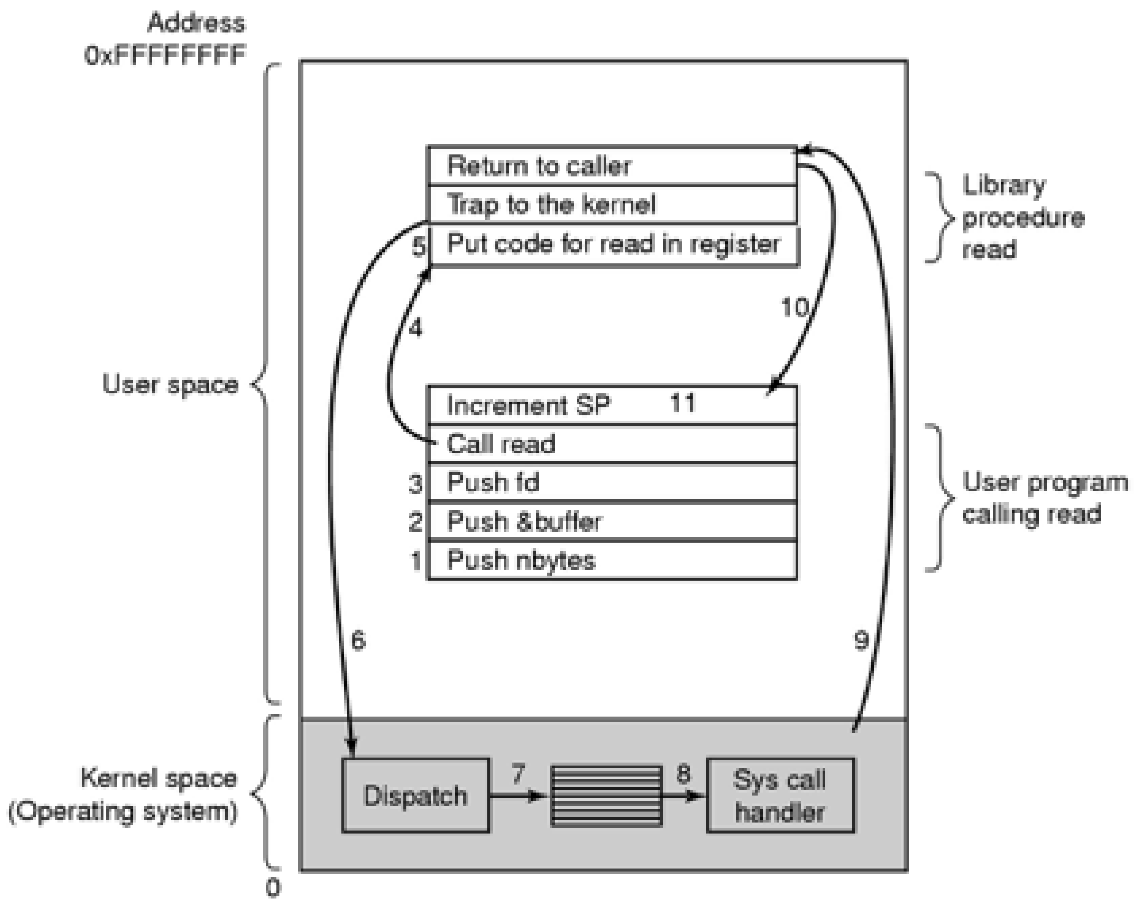
4 - invoke the system call

5 - put code for system call on register

6 - trap to the kernel

7-10 Since a number is associated with each system call, system call interface invokes/dispatch intended system call in OS kernel and return status of the system call and any return value

11- increment stack pointer



SYSTEM CALLS

- The location in the kernel a process can jump to is called *system_call*.
- The procedure at that location checks the system call number, which tells the kernel what service the process requested.
- Then, it looks at the table of system calls (sys_call_table) to see the address of the kernel function to call.
- Then it calls the function, and after it returns, does a few system checks and then returns back to the process (or to a different process, if the process time ran out).

SYSTEM CALL ERROR

- When a system call results in an error, it returns -1 and stores the reason the call failed in an external variable named "errno".
- When a system call returns successfully, it returns something other than -1, but it does not clear "errno".
- "errno" only has meaning directly after a system call that returns an error.
- The "/usr/include/errno.h" file maps these error numbers to manifest constants, and it is these constants that we should use in our programs.

SYSTEM CALL CATEGORIES

- **System calls are divided into 5 categories mainly :**
- Process Management
- File / Directory Management
- Device Management
- Information Maintenance
- Communication

PROCESS MANAGEMENT

- This system calls perform the task of process creation, process termination, etc.
- The Linux System calls under this are **fork()** , **exit()** , **exec().**

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

FILE MANAGEMENT

- File management system calls handle file manipulation jobs like creating a file, reading, and writing, etc.
- Some calls under this are **open(), read(), write(), close().**

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

DIRECTORY MANAGEMENT

Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

DEVICE MANAGEMENT

- Involves tasks like reading from device buffers, writing into device buffers, etc.
- The Linux System calls under this are ioctl(), read(), write().
- **ioctl():**
  - ioctl() is referred to as Input and Output Control.
  - ioctl is a system call for device-specific input/output operations and other operations which cannot be expressed by regular system calls.

INFORMATION MAINTENANCE

- It handles information and its transfer between the OS and the user program.
- In addition, OS keeps the information about all its processes and system calls are used to access this information.
- The System calls under this are **getpid(), alarm(), sleep().**
- **getpid()**
  - getpid stands for Get the Process ID.
  - The getpid() function shall return the process ID of the calling process.
  - The getpid() function shall always be successful and no return value is reserved to indicate an error.

INFORMATION MAINTENANCE

- **alarm()**
  - This system call sets an alarm clock for the delivery of a signal that when it has to be reached.
  - It arranges for a signal to be delivered to the calling process.
- **sleep()**
  - This System call suspends the execution of the currently running process for some interval of time
  - Meanwhile, during this interval, another process is given chance to execute

COMMUNICATION

- These types of system calls are specially used for inter-process communications.
- Two models are used for inter-process communication
  - Message Passing(processes exchange messages with one another)
  - Shared memory(processes share memory region to communicate)
- The system calls under this are **pipe() , shmget() ,mmap().**
- **pipe():**
  - The pipe() system call is used to communicate between different Linux processes.
  - It is mainly used for inter-process communication.
  - The pipe() system function is used to open file descriptors.

COMMUNICATION

- **shmget():**
  - shmget stands for shared memory segment.
  - It is mainly used for Shared memory communication.
  - This system call is used to access the shared memory and access the messages in order to communicate with the process.
- **mmap():**
  - This function call is used to map or unmap files or devices into memory.
  - The mmap() system call is responsible for mapping the content of the file to the virtual memory space of the process.

SYSTEM CALLS FOR MISCELLANEOUS TASKS

Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

FILE DESCRIPTORS

- Each UNIX process has 20 file descriptors at its disposal, numbered 0 through 19.
- The first three are already opened when the process begins
  - 0: The standard input
  - 1: The standard output
  - 2: The standard error output
- When the parent process forks a process, the child process inherits the file descriptors of the parent.

LETS MOVE TO PROGRAMMING …

SYSCALL, GETPID

```
#include <syscall.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
int main(void) {
long ID1, ID2;
/* direct system call - SYS_getpid (func
no. is 20) */
ID1 = syscall(SYS_getpid);
printf ("syscall(SYS_getpid)=%ld\n",
ID1);
/* "libc" wrapped system call */
/* SYS_getpid (Func No. is 20) */
ID2 = getpid();
printf ("getpid()=%ld\n", ID2);
return(0);
}
```

FORK

- int fork() turns a single process into 2 identical processes, known as the ***parent*** and the ***child***.
- On success, fork() returns 0 to the child process and returns the process ID of the child process to the parent process.
- The child process will have its own unique PID.
- On failure, fork() returns -1 to the parent process, sets errno to indicate the error, and no child process is created.

FORK

- Simple use of fork, where two copies are made and run together (multitasking)
-
- main()
- {

```
 int return_value;

  printf("Forking process\n");
  return_value = fork();
  printf("The process id is %d and return value is %d\n", getpid(),  return_value);
   printf("This line is not printed\n");
}
```

EXEC

Exec.c

- #include <stdio.h>
- #include <unistd.h>
- #include <stdlib.h>
- int main(int argc, char *argv[])
- {
- printf("PID = %d\n", getpid());
- char *args[] = {"Hello", "C", "Programming", NULL};
- execv("./hello", args);
- printf("Back to exec.c - This line will not be executed");
- return 0;
- }

Hello.c

- #include <stdio.h>
- #include <unistd.h>
- #include <stdlib.h>
- int main(int argc, char *argv[])
- {
- printf("We are in Hello.c\n");
- printf("PID of hello.c = %d\n", getpid());
- return 0;
- }

> gcc hello.c -o hello

STRACE

- If we want to see which system calls a program uses, the strace command is used :
- **strace <arguments>**.
- Execute the programs we did using strace.