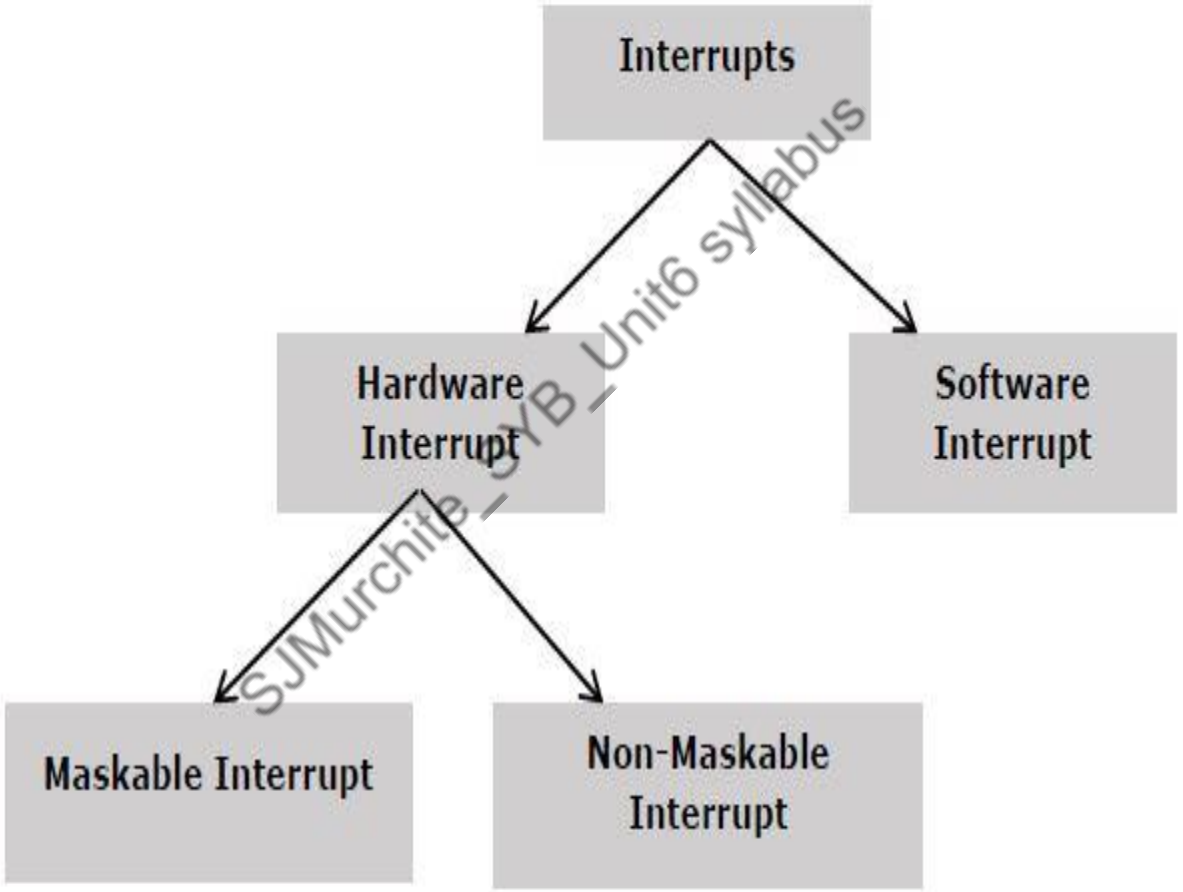
Unit 6: Interrupt and System calls

Sandip J.Murchite

An interrupt is an event that changes the sequence of instructions executed by the processor.

There are two different kinds of interrupts:
Synchronous interrupt (Exception) produced by the CPU while processing instructions
Asynchronous interrupt (Interrupt) issued by other hardware devices

Overview of interrupt and exception



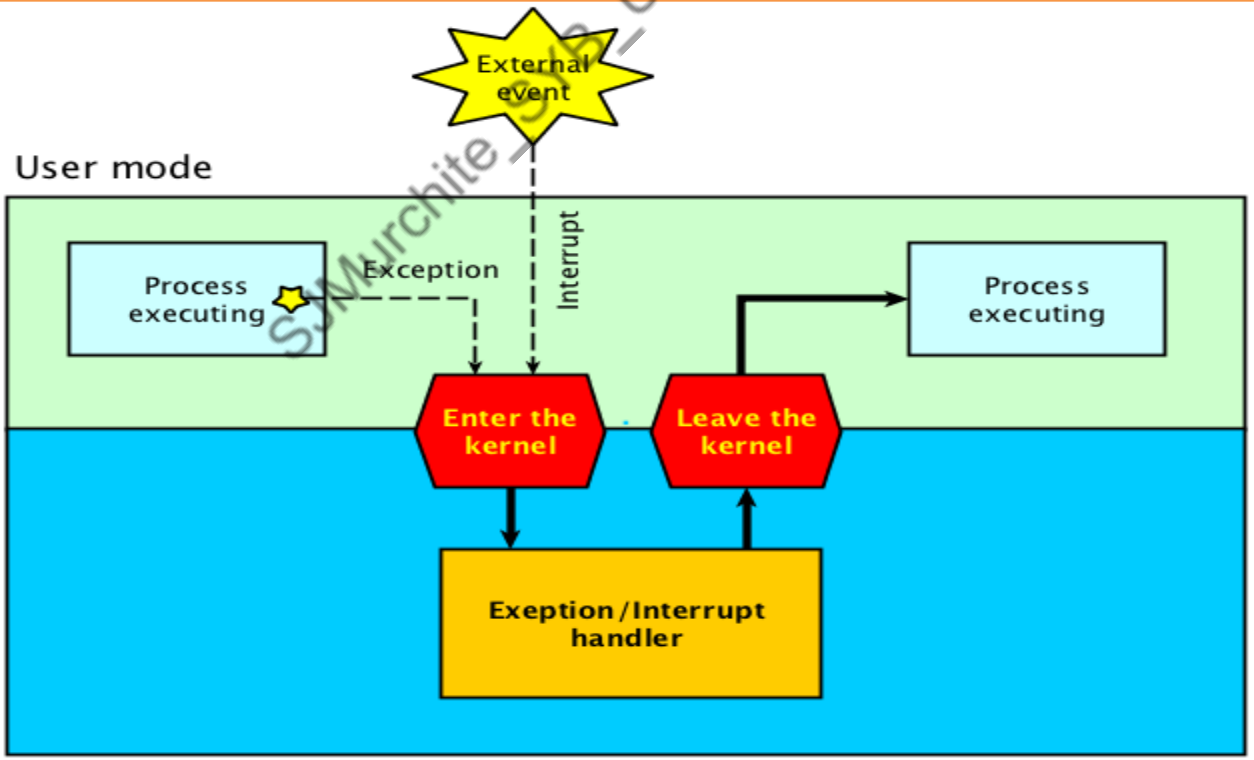
Overview of interrupt and exception

- **Interrupts** occur at random times during the execution of a program, in response to signals from hardware
- Software can also generate interrupts by executing the INT *n instruction.*
- **Exceptions** occur when the processor detects an error condition while executing an instruction, such as division by zero.

Overview of interrupt and exception

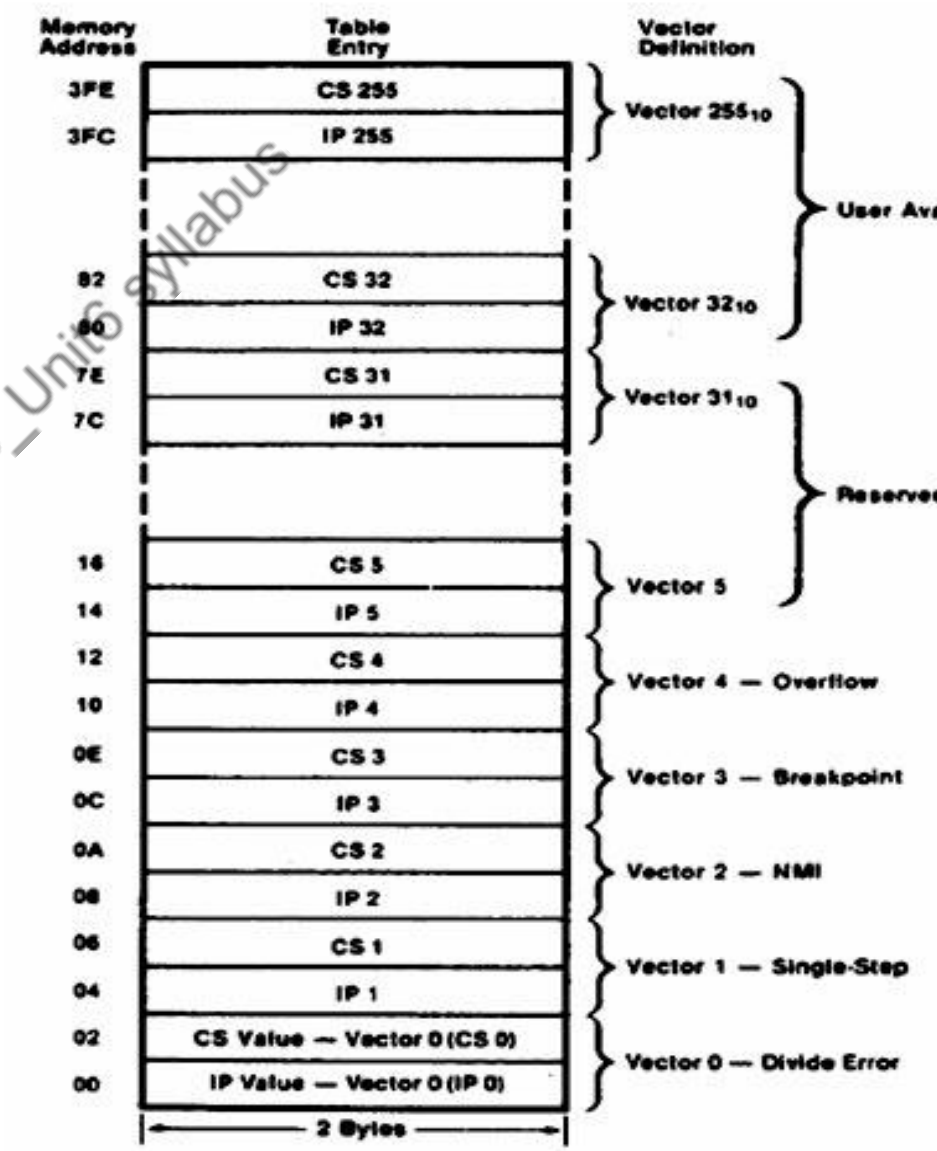
1) When an interrupt is received or an exception is detected, the currently running procedure or task is suspended while the processor executes an interrupt or exception handler.

2) When execution of the handler is complete, the processor resumes execution of the interrupted procedure or task



Exception and interrupt vector

- The processor uses the vector number assigned to an exception or interrupt as an index into the interrupt descriptor table (IDT).
- The allowable range for vector numbers is 0 to 255.
- Vector numbers in the range 5 through 31 are reserved by the Intel 64 and IA-32 architectures
- Vector numbers in the range 32 to 255 are designated as user-defined interrupts
- These interrupts are generally assigned to external I/O devices to enable those devices to send interrupts to the processor through one of the external hardware interrupt



Source of interrupts

- The processor receives interrupts from two sources:
- • **External (hardware generated) interrupts.**
- External interrupts are received through pins on the processor or through the local APIC
- **Maskable Hardware Interrupts**
- Any external interrupt that is delivered to the processor by means of the INTR pin or through the local APIC is called a mask able hardware interrupt.
- • **Software-generated interrupts.**
- The INT n instruction permits interrupts to be generated from within software *by* supplying an interrupt vector number as an operand.
- **For example**, the INT 35 instruction forces an implicit call to the interrupt handler for interrupt 35.
- Interrupts generated in software with the INT n instruction cannot be masked by the IF flag in the EFLAGS register.

ENABLING AND DISABLING INTERRUPTS

- IE and ID are machine instructions used to set in processor status register.
- To avoid interruption by same device during execution of ISR (interrupt service routine)

First instruction of ISR can be interrupt disable

Last instruction of ISR can be interrupt enable

- The IF flag does not affect non- mask able interrupts (NMIs) delivered to the NMI pin

Unit6: Part B- System call

Definition of system call

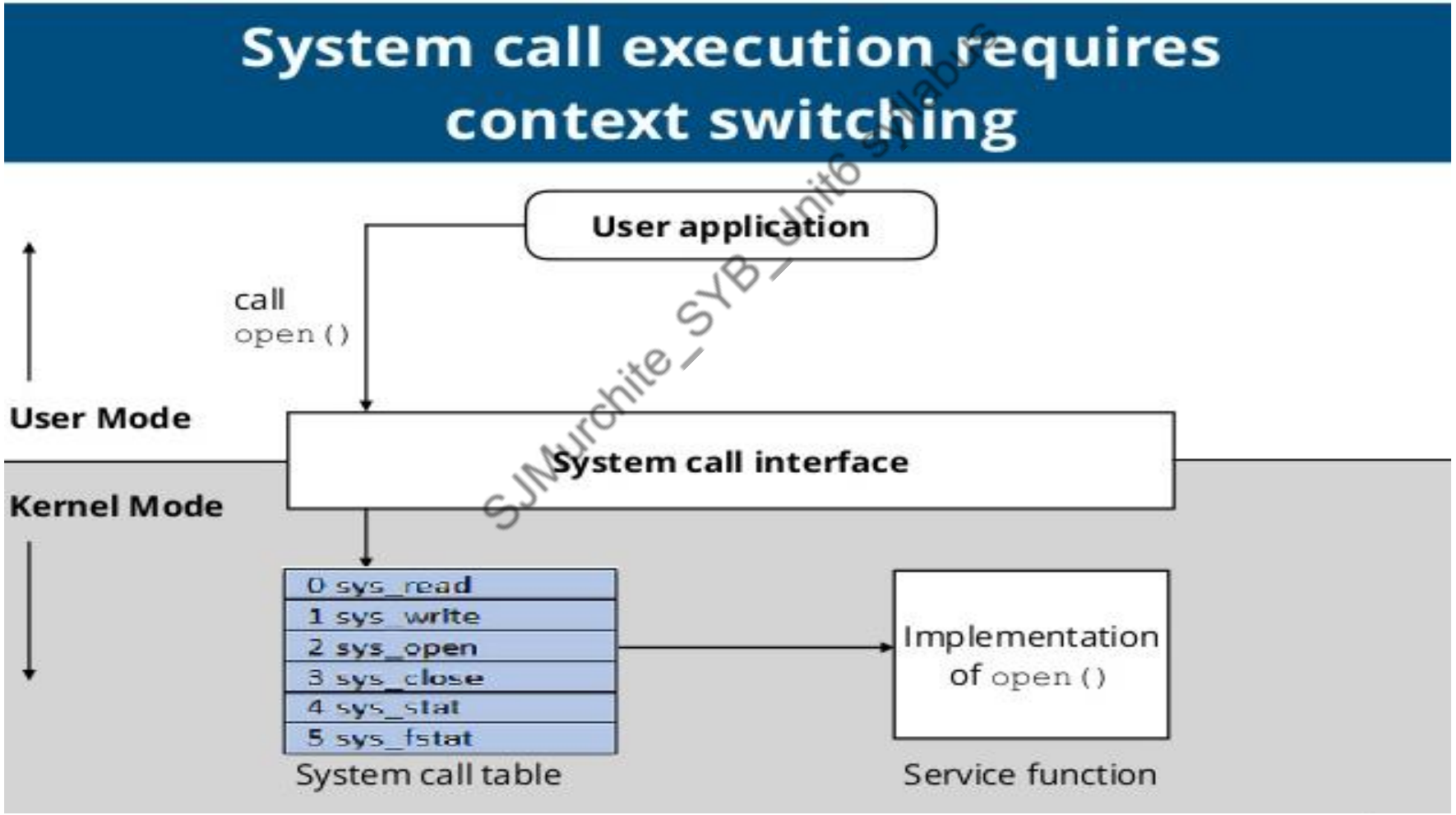
It is a method by which program makes a request to OS.

Many OS provide core functions that application programs can access called system calls.

Linux kernel interface

- Linux support multiple processes running simultaneously
- At any time, memory images of multiple processes exit simultaneously with the same physical memory.
- Multiple resources (IO devices) are used by a process in mutually exclusive manner with other processes.
- A recourse is used only by one process at a time and other processes wait for this process to release the resource before they can use it.
- To protect a process by other processes, Linux do not permit a process to do direct IO i.e. The OS does not permit OS code to be executed by a process in user mode.
- A process makes a call to a function within an OS to perform such operations. Such functionality within os is called system call.
- IA32 processors provide instruction for handling operating system calls.
- INT N N is an immediate constant ranging from 0 to 255
- In linux , Instruction int $0x80 is used for making system call to the operating system.

Example- open system call



System call identification

- Linux provide a mechanism for entering into system call by using int $0x80 instruction.
- After executing int $0x80, control of program transfers to a predefined location within os. This code is called as system call handler.
- System call handler handle various system call such as read , write, open etc as per the system call identification process.
- Each system cal in Linux is given a unique number for identification
- In Linux register EAX is used to pass the system call number.

System call values

- System call values
  - EAX register hold the system call value. This value defines which system call is used from list of system calls supported by kernel.
  - MOVL $1, %eax
  - Int $0x80

Finding system calls

- **/usr/include/asm/unistd.h**
- #define __NR_exit 1
- #define __NR_fork 2
- #define __NR_read 3
- #define __NR_write 4
- #define __NR_open 5
- #define __NR_close 6
- #define __NR_waitpid 7
- #define __NR_creat 8
- #define __NR_link 9
- #define __NR_unlink 10
- #define __NR_execve 11
- #define __NR_chdir 12
- #define __NR_time 13
- #define __NR_mknod 14
- #define __NR_chmod 15
- #define __NR_lchown 16

/ arch / arm / include / asm / unistd.h

```
24  /*
25   * This file contains the system call numbers.
26   */
27
28  #define __NR_restart_syscall          (__NR_SYSCALL_BASE+  0)
29  #define __NR_exit                     (__NR_SYSCALL_BASE+  1)
30  #define __NR_fork                     (__NR_SYSCALL_BASE+  2)
31  #define __NR_read                     (__NR_SYSCALL_BASE+  3)
32  #define __NR_write                    (__NR_SYSCALL_BASE+  4)
33  #define __NR_open                     (__NR_SYSCALL_BASE+  5)
34  #define __NR_close                    (__NR_SYSCALL_BASE+  6)
35                                        /* 7 was sys_waitpid */
36  #define __NR_creat                    (__NR_SYSCALL_BASE+  8)
37  #define __NR_link                     (__NR_SYSCALL_BASE+  9)
38  #define __NR_unlink                   (__NR_SYSCALL_BASE+ 10)
39  #define __NR_execve                   (__NR_SYSCALL_BASE+ 11)
40  #define __NR_chdir                    (__NR_SYSCALL_BASE+ 12)
41  #define __NR_time                     (__NR_SYSCALL_BASE+ 13)
42  #define __NR_mknod                    (__NR_SYSCALL_BASE+ 14)
43  #define __NR_chmod                    (__NR_SYSCALL_BASE+ 15)
44  #define __NR_lchown                   (__NR_SYSCALL_BASE+ 16)
45                                        /* 17 was sys_break */
46                                        /* 18 was sys_stat */
47  #define __NR_lseek                    (__NR_SYSCALL_BASE+ 19)
48  #define __NR_getpid                   (__NR_SYSCALL_BASE+ 20)
49  #define __NR_mount                    (__NR_SYSCALL_BASE+ 21)
50  #define __NR_umount                   (__NR_SYSCALL_BASE+ 22)
51  #define __NR_setuid                   (__NR_SYSCALL_BASE+ 23)
52  #define __NR_getuid                   (__NR_SYSCALL_BASE+ 24)
53  #define __NR_stime                    (__NR_SYSCALL_BASE+ 25)
54  #define __NR_ptrace                   (__NR_SYSCALL_BASE+ 26)
55  #define __NR_alarm                    (__NR_SYSCALL_BASE+ 27)
```

Parameter passing for system calls

- In linux, parameters are passed to a system call through registers.
- All system calls in linux takes less than or equal to 6 parameters
- The parameters are passed using registers ebx,ecx,edx,esi,edi,ebp.

Parameter passing for system calls

- Example1: write system call takes four parameters (including system call number)
- Movl $4,%eax
- Movl $1,%ebx
- Movl $output,%ecx
- Movl $len,%edx
- Int $0x80
- File descriptor
- 0- it normally bound to keyboard and used by read system call
- 1-it normally appear on console and used by write system call to write the data
- Example2: exit system call is used to terminate the process
- Movl $1,%eax
- Movl $0,%ebx
- Int $0x80

An Example of passing input values to a system call

```
.section .data
output:
.ascii “This is a test message.\n”
output_end:
.equ len, output_end - output
.section .text
.globl _start
_start:
movl $4, %eax
movl $1, %ebx
movl $output, %ecx
movl $len, %edx
int $0x80
movl $1, %eax
movl $0, %ebx
int $0x80
```

- **EAX register is used to hold the system call value.**
- **EBX: The integer file descriptor**
- The file descriptor value for the output location is placed in the EBX. Linux systems contain three special file descriptors:
- **0 (STDIN): The standard input for the terminal device (normally the keyboard)**
- **1 (STDOUT): The standard output for the terminal device (normally the terminal screen)**
- **2 (STDERR): The standard error output for the terminal device (normally the terminal screen)**
- **ECX: The pointer (memory address) of the string to display**
- **EDX: The size of the string to display.**

Return values from system calls

- Return value of system calls is implemented similar to function calls.
- System call return their values in eax register.
- There are two kinds of values
  - A) non negative
  - B) negative
- All non negative values returned by the system call represents successful execution of system call, whereas negative values represents error conditions.
- For write system call, it returns the size of the string written to the file descriptor.

Return values from system calls

System Call Value	System Call	Description
20	getpid	Retrieves the process ID of the running program
24	getuid	Retrieves the user ID of the person running the program
47	getgid	Retrieves the group ID of the person running the program

An example of getting a return value from a system call

```
.section .bss
.lcomm pid, 4
.lcomm uid, 4
.lcomm gid, 4

.section .text
.globl _start
_start:
movl $20, %eax
int $0x80
movl %eax, pid
movl $24, %eax
int $0x80
movl %eax, uid
movl $47, %eax
int $0x80
movl %eax, gid
end:
movl $1, %eax
movl $0, %ebx
int $0x80
```



```
(gdb) x/d &pid
0x80490a4 <pid>:        4758
(gdb) x/d &uid
0x80490a8 <uid>:        501
(gdb) x/d &gid
0x80490ac <gid>:        501
```

Starting a process in linux

- In Linux, a program is executed by creating a process and loading a program in newly created process.
- A fork system call is used to create a process and a program is loaded using execve system call.
- When a program is loaded , it run from memory location whose address is specified in executable file.
- By default starting address of the program is indicated by label _start

System call related to process information

System call name	System call value	description
Getuid	24	Retrieves user id of a person running the program
Getpid	20	Retrieves process id of running program.
Getgid	47	Retrieves group id of a person running the program.

Strace program

- The **strace** program intercepts system calls made by a program and displays them for you.

```
$ strace ./syscalltest2
execve("./syscalltest2", ["./syscalltest2"], [/* 38 vars */]) = 0
getpid()                                = 7616
getuid()                                = 501
getgid()                                = 501
_exit(0)                                = ?
$
```


- This output shows all of the system calls the syscalltest2 program made, in the order in which they were performed by the application.
- The left side shows the system call names, and the right side shows the return values produced by the system calls

System call related to process management

System call name	System call value	description
Fork		Create a child process
Execve		Execute a program
Exit		Terminate the current process
Nice		Change the process priority.
Kill		Send a signal to kill a process.

Thank you