Interrupt and System calls for Linux

Prepared by Mrs. J. D. Pakhare

Interrupts and Exceptions

- ***Interrupts***
- An interrupt is a way for the processor to “interrupt” the current instruction code path and switch to a different path.

- Interrupts come in two varieties i.e. sources of interrupts:
  - **Software interrupts**
    - Programs generate software interrupts.
    - They are a signal to hand off control to another program.
    - The INT N instruction permits interrupts to be generated within software by supplying an interrupt vector number as an operand.
    - Interrupts generated in software with the INT N instruction cannot be masked by the IF flag in the EFLAGS register.

Interrupts and Exceptions

- **Hardware interrupts**
  - Hardware devices generate hardware interrupts
  - External interrupts are received through pins on the processor
  - They are used to signal events happening at the hardware level (such as when an I/O port receives an incoming signal)

Difference: Interrupts and Exceptions

- **Interrupts** are used to handle asynchronous events external to the processor

- Two sources for external interrupts:
  - Maskable interrupts, which are signalled via the INTR pin.
  - Nonmaskable interrupts, which are signalled via the NMI (Non-Maskable Interrupt) pin.

  - 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.
  - When execution of the handler is complete, the processor resumes execution of the interrupted procedure or task

Difference: Interrupts and Exceptions

- **Exceptions** handle conditions detected by the processor itself while executing instructions. E.g. divide by zero.

- Two sources for exceptions:
  - Processor detected. These are further classified as faults, traps, and aborts.
  - Programmed. The instructions INTO, INT 3, INT can trigger exceptions.
  - These instructions are often called "software interrupts", but the processor handles them as exceptions.

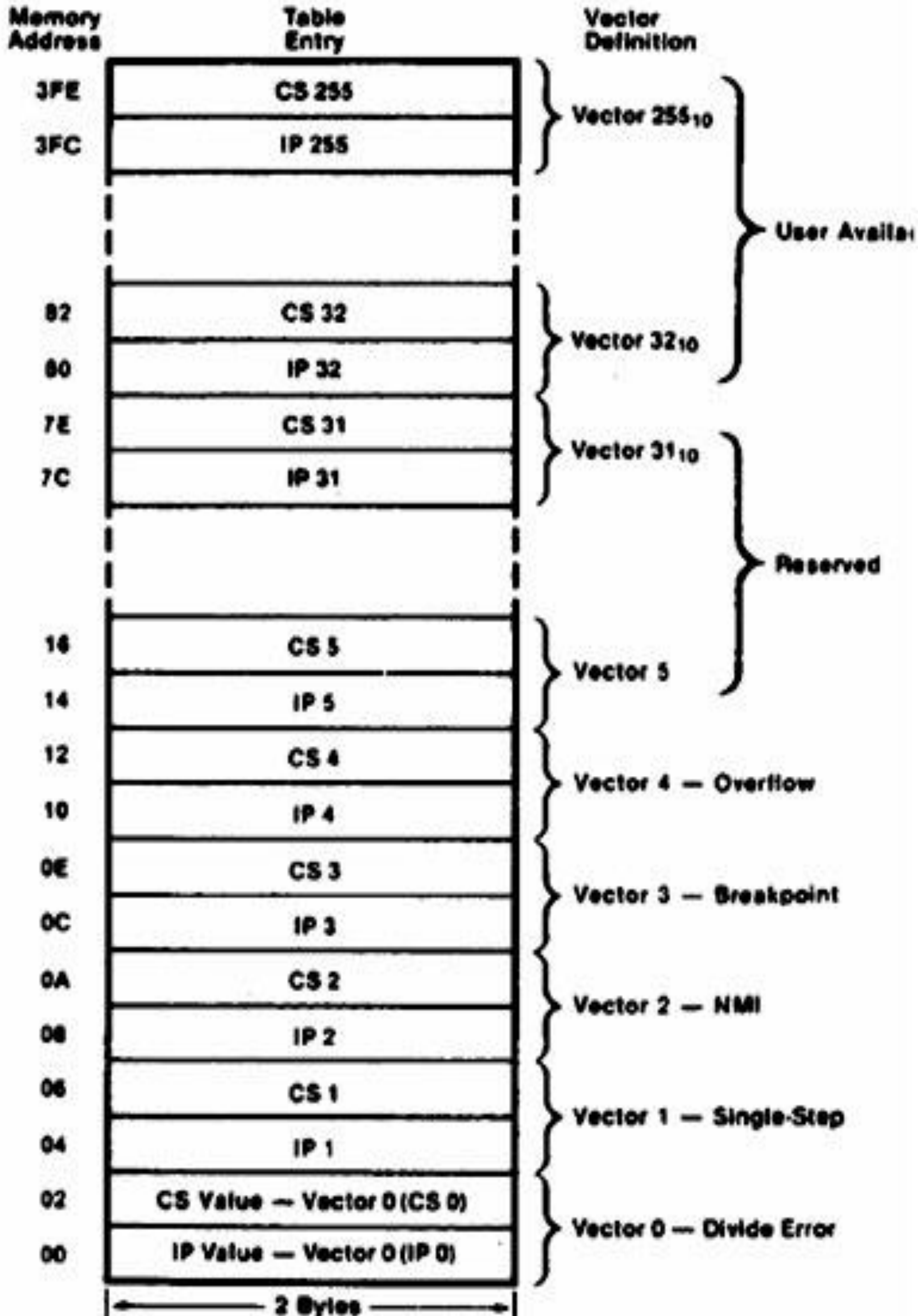
- Interrupts and exceptions vector to interrupt procedures via an interrupt table.

- The entries of the interrupt table are pointers to the entry points of interrupt or exception handler procedures.

- When an interrupt occurs, the processor pushes the current values of CS:IP onto the stack, disables interrupts, clears TF (the single-step flag), then transfers control to the location specified in the interrupt table.

Interrupt

- Internal or external interrupt is assigned with type (N) or specified with instruction INT N (00 to FFh)
- Intel has reserved 1024 locations for storing interrupt vector table.
- Total 256 types of interrupts from (00 to FFh)
- Each interrupt requires 4 bytes, 2 bytes for IP and 2 bytes for CS.
- interrupt vector table (IVT) starts at 0000:0000 and ends at 0000:03FFh
- (IVT) contains IP and CS of all interrupt types from 0000:0000 to 0000:03FFh
- 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



system call format:

- To initiate a system call, the INT instruction is used.
- The Linux system calls are located at interrupt 0x80.
- When the INT instruction is performed, all operations transfer to the system call handler in the kernel.
- When the system call is complete, execution transfers back to the next instruction after the INT instruction (otherwise the exit system call is performed).

- ***System call values***:
  - Each system call is assigned a unique number to identify it.
  - EAX register is used to hold the system call value.
  - This value defines which system call is used from the list of system calls supported by the kernel.
  - e.g exit system call

```
MOVL $1, %eax
int $0x80
```

Linux Kernel/system call interface

- Linux supports multiple processes running simultaneously and time sharing the same CPU.
- So that memory images of multiple processes exists simultaneously within the same physical memory.
- OS ensures that a process gets operate on its own memory image.

Linux Kernel /system call interface

- Other resources such as I/O devices used by a process in a mutually exclusive manner with other processes.
- In such a mechanism, resource 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 implement Protection of a process by other processes, Linux do not permit a process to do the direct I/O .

Linux Kernel /system call interface

- Process makes a call to a function within the OS to perform I/O operations.
  - e.g. file can be read only if the process has a permission to read that file.
- Such functionality within the OS is called a mechanism of making **system calls.**
- OS do not permit any part of OS code to be executed by a process in the normal user mode.
- Calls to the OS function are not made by normal call instructions.

Linux Kernel /system call interface

- Processors provide a mechanism for making system calls by using an instruction.
  – e.g. IA32 provides a trap kind instructions for handling Operating system calls.
  – Execution of this trap kind of instructions causes the CPU to execute predetermined program within the OS.

e.g. int type

  - type –immediate constant value between 0 to 255.

- In Linux**, type 0x80** is used for making system call (exit) to the OS.

System call identification

- Linux on IA32 provides only one mechanism for entering into a system call by using **int 0x80 instruction.**

- After executing this instruction the control of the program is transferred to the predetermined location within the OS kernel program. This code is known as system call handler.

- This system call distinguishes other system calls such as read , write, open etc.

- After identifying system call , appropriate function call is made within the OS to serve the system call.

- Each system call in Linux has unique identification number. This number is passed as an argument.

- The register **EAX is used to pass the system call number**.

- Many OS provide core functions that application programs can access i.e. access files, determine user and group permissions, access network resources, and retrieve and display data. These functions are called **system calls**.

System call identification:

- EAX register is used to hold the system call value.
- This value defines which system call is used from the list of system calls supported by the kernel.
- e.g exit system call

```
MOVL $1, %eax
int $0x80
```

Finding system calls

/usr/include/asm/unistd.h

The unistd.h file contains definitions for each of the system calls available in the kernel.

```
#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
```

Each system call is defined as a name (preceded by __NR_), and its system call number.

Parameter passing for System call

- In GNU/Linux parameters are passed to a system call through registers .
- All system call take less than or equal to six paramters.
- Parameters are passed using registers ebx , ecx , edx , esi, edi and ebp
- In addition to the parameter of the system call, a system call identification no is passed in register eax.
- In GNU/Linux all system call take only integer parameter or address of memory variables.

- e.g.
- Write system call takes 3 parameters other than system call identification no in register eax.
  - First parameter is an integer known as file descriptor is in ebx register.
  - Second parameter is the address of the buffer from where the data is written to the file identified by file descriptor and is in ecx register.
  - Third parameter is an integer that provides the size of the data to be written in the file and is in edx register.

  - All parameters are given below--

- The input values would be assigned to the following registers:
  - **EBX:** The integer file descriptor
  - **ECX:** The pointer (memory address) of the string to display
  - **EDX:** The size of the string to display

- 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 and is used by the read system call to read data from the keyboard.
  - **1 (STDOUT):** The standard output for the terminal device normally the terminal screen . The output of this file by means of write system call and appears on screen.
  - **2 (STDERR):** The standard error output for the terminal device. Output of this file appears on the terminal or output screen

Example of passing parameters /input values to write system call to display message

```
.section .data
output:
.ascii "Computer Science.\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.**
  - The system call value for the write() system call (4) is placed in the EAX register
- **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.**

- After assembling and linking the program —

```
$ ./syscall1test
 Computer Science.
$
```

- .equ directive is used to define the length value by subtracting the two labels:

- The .equ directive is used to set a constant value to a symbol that can be used in the text section.
- Once set, the data symbol value cannot be changed within the program.
- The .equ directive can appear anywhere in the data section,

Return values from System calls :

- Return value mechanism is implemented in a way similar to that in function calls.

- Many system calls return a value after they complete.

- System calls return a 32 bit value as an integer.

- The return value from a system call is placed in the EAX register.
- E.g. exit system call never returns and therefore there is no return value of the exit system call.

- It is your job to check the value in the EAX register, especially for failure conditions.

- All non-negative values returned by the system call represent successful execution of the system call.
- All negative values represent error condition .
- e.g.
  - In the case of the write system call, it returns the size of the string written to the file descriptor, or a negative value if the call fails.
  - The exit system call never returns and therefore there is no return value of the exit system call.

Starting a process in GNU/Linux

- in GNU/Linux a program is executed by creating a process and then loading a program in the newly created process.

- A process is created using fork system call while the program is loaded using the execve system call.

- When a program is loaded it is made to run from a memory location whose address is specified in the executable file.

- This address is known as the start address of the program.

- By default the start address of the program is indicated by a symbolic label _start.
- Programs written in Assembly Language can define any address as the start address of the program.
- When a program is loaded in the memory upon execution of execve system call, the OS performs several operations.
  - Like it initializes the stack and various CPU registers.

- **Process** is assigned a **process ID**, or **PID**. This is how the operating system identifies and keeps track of **processes.**

- Operating systems identify a **user** by a value called a **user** identifier (**UID**)

- and Identify group by a group identifier (**GID**), are used to determine which system resources a **user** or group can access.

An example to display return values of system calls related to process

```
.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
```

After moving each system call value to the EAX register and executing the INT instruction, the return value in EAX is placed in the appropriate memory location.

An example of getting a return value from a system call

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

```
$ gdb -q syscalltest2
(gdb) break *end
Breakpoint 1 at 0x8048098: file syscalltest2.s, line 21.
(gdb) run
Starting program: /home/rich/palp/chap12/syscalltest2

Breakpoint 1, end () at syscalltest2.s:21
21          movl $1, %eax
Current language:  auto; currently asm
(gdb) x/d &pid
0x80490a4 <pid>:        4758
(gdb) x/d &uid
0x80490a8 <uid>:        501
(gdb) x/d &gid
0x80490ac <gid>:        501
```

- The values in the pid, uid, and gid memory locations can be displayed as integer values using the x/d debugger command.

- While the process ID is unique to the running program, you can check the uid and gid values using the id shell command:

```
$ id
uid=501(rich) gid=501(rich) groups=501(rich), 22(cdrom), 43(usb), 80(cdwriter),
81(audio), 503(xgrp)
$
```

System calls related to process management:

- A process in GNU/Linux has 3 user ids- real user ID, effective user ID and saved user ID.

- Also it has 3 group ids –real group ID, effective group ID and saved group ID.

- Processes have parent child relationship

- For job control reasons , the processes may be grouped together and given a unique process group ID.

- Various system calls in GNU/Linux that handle the process and process related information.

- **System call:** getuid
- Input:

  eax: SYS.getuid

  – It returns the Real user ID of the process.

- **System call:** getgid
- Input:

  eax: SYS.getgid

  – It returns the group ID of the process.

- **System call:** getpid
- Input:

  eax: SYS.getpid

  – It returns the process ID of the calling process.

Process management:

- **System call:** fork
- Input:

  eax: SYS.fork

  – It returns child process ID in the parent process and 0 in the child process.

- Fork system creates a new process
- Calling process becomes the parent of the newly created process .
- Child process so created has its own address space which is initialized to the values from the parent process.
- The newly created process belongs to the same session and process group as the parent.
- This system call has 2 returns. One in parent and one in child process.
  - To the parent process the return value is the process ID of the child while to the child process the return value is 0.

- **System call:** execve
- Input:

  eax: SYS.execve

  ebx: address of pathname string

  ecx: address of an array of multiple command line arguments

  edx: address of an array of multiple environment strings.

  – Does not return.
- This system call initializes the memory map of the calling process by a new program loaded from an executable file given in register ebx.

- **System call:** exit
- Input:

  eax: SYS.exit

  - Does not return.
- The exit system call is used to terminate the calling process.

- **System call:** kill
- Input:

  eax: SYS.kill

  ebx:ID

  ecx:signal

  - The kill system call is used to send any signal to any process or processes.
  - The ID in ebx register will determine how signal is sent.

- **System call:** nice
- Input:

  eax: SYS.nice

  ebx: increment

  - The nice system call can change the priority of the calling process.
  - A normal process can only provide a positive increment.

Thank You