**Chapter 12: ADVANCED PROGRAMMING**

**Topic – 1: Procedures**

**Introduction**

* Procedures are also known as **sub-routines**.
* It is same as **function** in **C**.
* We use a **return** statement to end them.

***proc\_name:***

***; Procedure body***

***ret***

***call proc\_name ; Calling our procedure***

* When a procedure is **called**, we say program is **returning control** to the called procedure.
* We can also call the procedure **first** and define it later on in some line.

**Stack Data Structure**

* **Top:** The location from where stack element is **pushed** or **popped**.
* **PUSH** takes any operand, whereas **POP** can be an address or a register with address.
* Registers **SS** and **RSP** (or **SP**) are used for **implementing the stack**.
* **SP** register points to the **top** of stack.
* **SS** register which is **no** more used in today’s **64-bit architecture**, points to the **beginning** of the stack.
* We can store **WORDs** and **DWORDs** in **stack** & **not BYTE**.
* Stack is filled in **reverse direction** of memory, meaning toward **lower memory** address.
* Or simply saying, register **RSP** is **decremented**.

**Displaying All ASCII Characters**

* **Step 1:** Insert value of **256** in **ECX**.
* **Step 2:** Display it to the screen.
* **Step 3:** Decrement **ECX**.
* **Step 4:** Repeat **step 2 & 3** until **ECX** is **zero**.

**Topic – 2: Recursions**

* **Direct recursion:** In this type of recursion, a **procedure** calls **itself**.
* **Indirect recursion:** In this type of recursion, a **procedure** calls **another procedure** & then that procedure calls the **first procedure**.

**Topic – 3: Macros**

**Introduction**

* ***Macros*** are **sequence of instructions** which can be assigned using a **name**.
* It can be defined **anywhere** in the program.
* Starts with **%macro** & ends with **%endmacro**.

***%macro name no\_of\_parameters***

***; Macro body***

***%endmacro***

**Example**

***; Defining macro***

***%macro display 2***

***mov rdx, %1***

***mov rcx, %2***

***mov rbx, 1***

***mov eax, 4***

***int 80***

***%endmacro***

***; Calling macro***

***display len, msg***

**Procedures v/s Macros**

|  |  |
| --- | --- |
| **Procedures** | **Macros** |
| **Has overhead** | **No overhead** |
| **Used for large tasks.** | **Used more for small tasks.** |
| **Runtime evaluation (dynamic)** | **Compile-time evaluation (static)** |
| **Return value** | **No return value** |
| **Easier to debug & maintain.** | **Difficult to debug & maintain.** |

**Topic – 4: File Management**

**Standard File Streams**

* Standard input (**stdin**)
* Standard output (**stdout**)
* Standard error (**stderr**)

**File Descriptor**

* **File descriptor** is a **16-bit** integer **ID** provided to each file involved in program.
* It is used to refer to those files in program.

**File Descriptor Codes**

* **stdin –** 0
* **stdout –** 1
* **stderr –** 2

**File Pointer**

* Points to **offset** of contents in file (a byte at a time).
* This **offset value** is represented in terms of **bytes**.

**File Handling System Calls**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **%eax** | **Name** | **%ebx** | **%ecx** | **%edx** |
| **2** | **sys\_fork** | **struct pt\_regs** | **-** | **-** |
| **3** | **sys\_read** | **unsigned int** | **char \*** | **size\_t** |
| **4** | **sys\_write** | **unsigned int** | **const char \*** | **size\_t** |
| **5** | **sys\_open** | **const char \*** | **int** | **int** |
| **6** | **sys\_close** | **unsigned int** | **-** | **-** |
| **8** | **sys\_creat** | **const char \*** | **int** | **-** |
| **19** | **sys\_lseek** | **unsigned int** | **off\_t** | **unsigned int** |

**Steps Involved (System Call)**

* **Step 1:** Put the system call in **EAX** register.
* **Step 2:** Store arguments to **system call** in register **EBX** & **ECX**.
* **Step 3:** Call a relevant interrupt like **80h**.

**Note!**

**🡪 Result is returned in EAX register.**

**🡪 This result for sys\_open is a file descriptor (+ve for success & -ve for failure).**

**🡪 For sys\_read its number of bytes read.**

**🡪 For sys\_read its number of bytes written.**

**🡪 0 is returned to EAX for successful closing of file, in sys\_close.**

**Register Values For Operations**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **File Operation** | **EDX** | **ECX** | **EBX** | **EAX** |
| **Create + Open** | **-** | **File permissions** | **Filename** | **8 (sys\_creat)** |
| **Open** | **File permissions** | **Access mode** | **Filename** | **5 (sys\_open)** |
| **Read** | **Bytes to read** | **Pointer to input buffer** | **File descriptor** | **3 (sys\_read)** |
| **Write** | **Bytes to write** | **Pointer to output buffer** | **File descriptor** | **4 (sys\_write)** |
| **Close** | **-** | **-** | **File descriptor** | **6 (sys\_close)** |
| **Update** | **Offset reference position** | **Offset value** | **File descriptor** | **19 (sys\_lseek)** |

**Reference Position Values**

* Beginning of file – **0**
* Current position – **1**
* End of file – **2**

**File Permission Codes**

* Read, write & execute by all – **0777**

**Topic – 5: 64-Bit Specific Changes**

**System Call**

* For **interrupts** in **64-bit** architecture, we call ***syscall*** instead of int ***0x80***.

**Argument Passing Order**

|  |  |  |
| --- | --- | --- |
| **Argument Number** | **32-Bit Architecture** | **64-Bit Architecture** |
| **1st** | **EBX** | **RDI** |
| **2nd** | **ECX** | **RSI** |
| **3rd** | **EDX** | **RDX** |
| **4th** | **-** | **R10** |
| **…** | **…** | **…** |
| **13th** | **-** | **R1** |

**Topic – 6: Memory Management**

**Introduction**

* We pass **sys\_bark()** system call to kernel for **manually allocating memory** to us.
* This memory is allocated in the part memory occupied by our program.
* We call this occupied memory as **application image**.
* The limit to allocating memory depends on **how much** of it is occupied by the program.

**Steps Involved**

* When we call interrupt with **45** in **EAX** & **0** in **EBX**, system break value (pointer to end of file’s memory) is returned to **EAX** after calling **interrupt** (**0x80**).
* Then we can **again** move **45** to **EAX** & any new breakpoint we want to **EBX** before calling **interrupt** (**0x80**).

**Code Example**

***; Get current break***

***mov eax, 45***

***mov ebx, 0 ; To return default break point to EAX***

***int 0x80 ; Returns break pointer to EAX***

***mov [curr\_brk], eax ; Setting break (curr\_brk is a variable)***

***; Allocate 100 bytes***

***mov eax, 45***

***mov ebx, [curr\_brk] + 100 ; To set the new break point***

***int 0x80***

***mov [last\_point], eax ; Move new break point to last\_point***

***; Fill allocated memory***

***mov byte [last\_point], 72***

***; Deallocate 100 bytes***

***mov eax, 45***

***mov ebx, [last\_point] - 100 ; Shrinking from latest break point***

***int 0x80***

* We used two variables (**curr\_brk** and **last\_point**) because we may use first variable for restoring back to original state.

***mov eax, 45***

***mov ebx, [curr\_brk]***

***int 0x80***