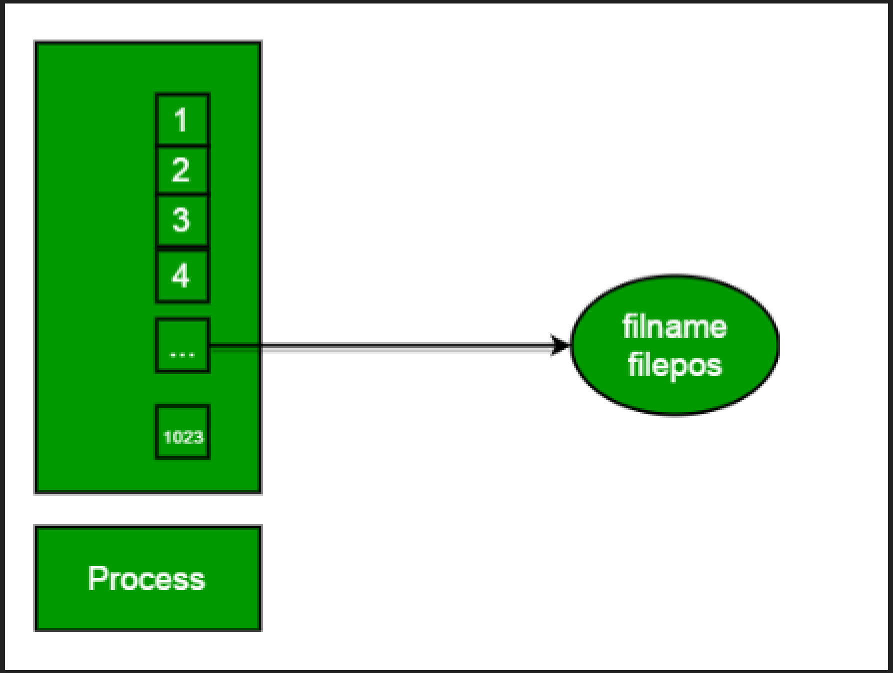
Unix System Interface

{C}
Programming

What is the File Descriptor?

- The file descriptor is an integer that uniquely identifies an open file of the process.
- **File Descriptor table: A file** descriptor table is the collection of integer array indices that are file descriptors in which elements are pointers to file table entries. One unique file descriptors table is provided in the operating system for each process.



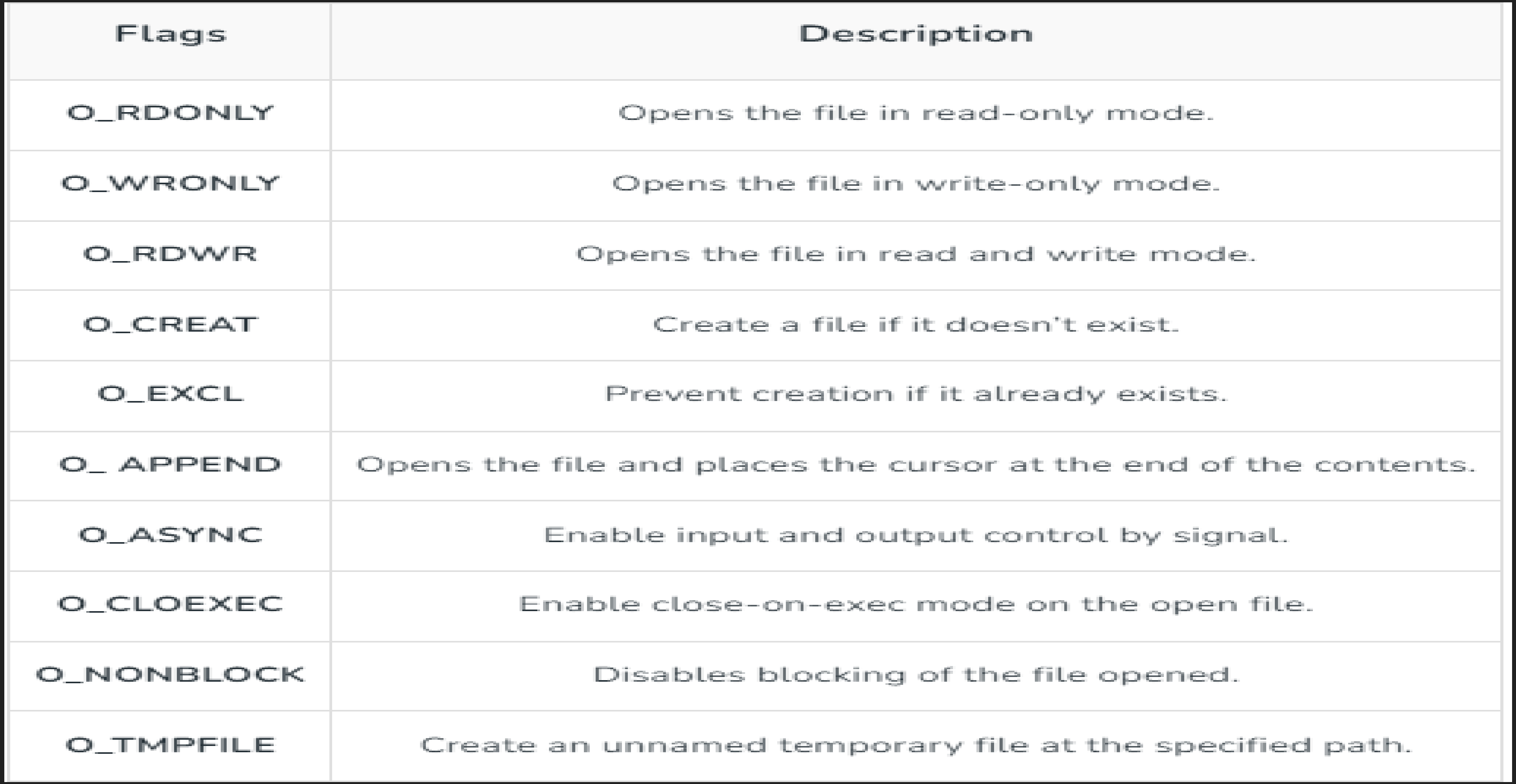
What is the File Descriptor?

- **File Table Entry:** File table entries are a structure In-memory surrogate for an open file, which is created when processing a request to open the file and these entries maintain file position.
- **Standard File Descriptors**: When any process starts, then that process file descriptors table’s fd(file descriptor) 0, 1, 2 open automatically, (By default) each of these 3 fd references file table entry for a file named **/dev/tty**
- **/dev/tty**: In-memory surrogate for the terminal.
- **Terminal**: Combination keyboard/video screen.

Input/Output System Calls

- Basically, there are total 5 types of I/O system calls:
- **1. C create**
- The create() function is used to create a new empty file in C. We can specify the permission and the name of the file which we want to create using the create() function. It is defined inside **<unistd.h>** header file and the flags that are passed as arguments are defined inside **<fcntl.h>** header file.
- **Syntax of create() in C**
- int **create**(char **filename*, mode_t *mode*);
- **Parameter**
- **filename:** name of the file which you want to create
- **mode:** indicates permissions of the new file.
- **Return Value**
- return first unused file descriptor (generally 3 when first creating use in the process because 0, 1, 2 fd are reserved)
- return -1 when an error

- **2. C open**
- The open() function in C is used to open the file for reading, writing, or both. It is also capable of creating the file if it does not exist. It is defined inside **<unistd.h>** header file and the flags that are passed as arguments are defined inside **<fcntl.h>** header file.
- **Syntax of open() in C**
- int **open** (const char* *Path*, int *flags*);
- **flags:** It is used to specify how you want to open the file. We can use the following flags.



Flags	Description
O_RDONLY	Opens the file in read-only mode.
O_WRONLY	Opens the file in write-only mode.
O_RDWR	Opens the file in read and write mode.
O_CREAT	Create a file if it doesn't exist.
O_EXCL	Prevent creation if it already exists.
O_ APPEND	Opens the file and places the cursor at the end of the contents.
O_ASYNC	Enable input and output control by signal.
O_CLOEXEC	Enable close-on-exec mode on the open file.
O_NONBLOCK	Disables blocking of the file opened.
O_TMPFILE	Create an unnamed temporary file at the specified path.

```
// C program to illustrate
// open system call
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

extern int errno;

int main()
{
        // if file does not have in directory
        // then file foo.txt is created.
        int fd = open("foo.txt", O_RDONLY |
O_CREAT);

        printf("fd = %d\n", fd);
```

```
        if (fd == -1) {
                // print which type of error have
in a code
                printf("Error Number % d\n",
errno);

                // print program detail "Success
or failure"
                perror("Program");
        }
        return 0;
}
```

3. C close

- The close() function in C tells the operating system that you are done with a file descriptor and closes the file pointed by the file descriptor. It is defined inside **<unistd.h>** header file.
- **Syntax of close() in C**
- int close(int fd);
  Parameter
- **fd: F**ile descriptor of the file that you want to close.
- **Return Value**
- **0** on success.
- **-1** on error.

```
// C program to illustrate
close system Call
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
      int fd1 =
open("foo.txt", O_RDONLY);
      if (fd1 < 0) {
            perror("c1");
            exit(1);
```

```
printf("opened the fd = % d\n", fd1);

// Using close system Call
if (close(fd1) < 0) {
      perror("c1");
      exit(1);
}
printf("closed the fd.\n");
```

```
}
```

4. C read

- From the file indicated by the file descriptor fd, the read() function reads the specified amount of bytes **cnt** of input into the memory area indicated by **buf**. A successful read() updates the access time for the file. The read() function is also defined inside the <unistd.h> header file.
- **Syntax of read() in C**
- size_t **read** (int *fd*, void* *buf*, size_t *cnt*);

Parameters

fd: file descriptor of the file from which data is to be read.

buf: buffer to read data from

cnt: length of the buffer

```
// C program to illustrate
// read system Call
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
        int fd, sz;
        char* c = (char*)calloc(100,
sizeof(char));
```

```
fd = open("foo.txt", O_RDONLY);
        if (fd < 0) {
                perror("r1");
                exit(1);
        }
        sz = read(fd, c, 10);
        printf("called read(% d, c, 10).
returned that"
                " %d bytes were read.\n",
                fd, sz);
        c[sz] = '\0';
        printf("Those bytes are as follows:
% s\n", c);
        return 0;
}
```

5. C write

Writes cnt bytes from buf to the file or socket associated with fd. cnt should not be greater than INT_MAX (defined in the limits.h header file). If cnt is zero, write() simply returns 0 without attempting any other action.

The write() is also defined inside **<unistd.h>** header file.

Syntax of write() in C

size_t **write** (int *fd*, void* *buf*, size_t *cnt*);
Parameters

fd: file descriptor

buf: buffer to write data from.

cnt: length of the buffer.

```
// C program to illustrate
// write system Call
#include<stdio.h>
#include <fcntl.h>
main()
{
int sz;

int fd = open("foo.txt", O_WRONLY |
O_CREAT | O_TRUNC, 0644);
if (fd < 0)
{
        perror("r1");
        exit(1);
}

sz = write(fd, "hello geeks\n", strlen("hello
geeks\n"));

printf("called write(% d, \"hello geeks\\n\", %d)."
        " It returned %d\n", fd, strlen("hello
geeks\n"), sz);

close(fd);
}
```

Dynamic memory allocation in C(Storage Allocator)

The concept of **dynamic memory allocation in c language** *enables the C programmer to allocate memory at runtime*. Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

malloc()	allocates single block of requested memory.
calloc()	allocates multiple block of requested memory.
realloc()	reallocates the memory occupied by malloc() or calloc() functions.
free()	frees the dynamically allocated memory.

malloc() function in C

- The malloc() function allocates single block of requested memory.
- It doesn't initialize memory at execution time, so it has garbage value initially.
- It returns NULL if memory is not sufficient.
- The syntax of malloc() function is given below:
- ptr=(cast-type*)malloc(byte-size)

```c
#include<stdio.h>
#include<stdlib.h>
int main(){
  int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc
    if(ptr==NULL)
    {
        printf("Sorry! unable to allocate memory");
        exit(0);
}
    printf("Enter elements of array: ");
    for(i=0;i<n;++i)
    {
        scanf("%d",ptr+i);
        sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
return 0;
}
```

calloc() function in C

- The calloc() function allocates multiple block of requested memory.
- It initially initialize all bytes to zero.
- It returns NULL if memory is not sufficient.
- The syntax of calloc() function is given below:
- ptr=(cast-type*)calloc(number, byte-size)

```c
#include<stdio.h>
#include<stdlib.h>
int main(){
 int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)calloc(n,sizeof(int));  //memory allocated using calloc
    if(ptr==NULL)
    {
        printf("Sorry! unable to allocate memory");
        exit(0);
    }
    printf("Enter elements of array: ");

    for(i=0;i<n;++i)
    {
        scanf("%d",ptr+i);
        sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
return 0;
}
```

realloc() method

- **“realloc”** or **“re-allocation”** method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**. re-allocation of memory maintains the already present value and new blocks will be initialized with the default garbage value.
- **Syntax of realloc() in C**
- ptr = realloc(ptr, newSize);
  where ptr is reallocated with new size 'newSize'.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{

        // This pointer will hold the
        // base address of the block created
        int* ptr;
        int n, i;
```

```
else {

                // Memory has been successfully
allocated
                printf("Memory successfully allocated
using calloc.\n");


                // Get the elements of the array
                for (i = 0; i < n; ++i) {
                        ptr[i] = i + 1;
                }


                // Print the elements of the array
                printf("The elements of the array are:
");
                for (i = 0; i < n; ++i) {
                        printf("%d, ", ptr[i]);
                }
```

```
// Get the number of elements for the array
        n = 5;
        printf("Enter number of elements: %d\n", n);

        // Dynamically allocate memory using calloc()
        ptr = (int*)calloc(n, sizeof(int));

        // Check if the memory has been successfully
        // allocated by malloc or not
        if (ptr == NULL) {
                printf("Memory not allocated.\n");
                exit(0);
        }
        // Get the new size for the array
        n = 10;
        printf("\n\nEnter the new size of the
array: %d\n", n);

        // Dynamically re-allocate memory
using realloc()
        ptr = (int*)realloc(ptr, n * sizeof(int));

        // Memory has been successfully
allocated
        printf("Memory successfully re-
allocated using realloc.\n");

        // Get the new elements of the array
        for (i = 5; i < n; ++i) {
            ptr[i] = i + 1;
```

```
}

        // Print the elements of the array
        printf("The elements of the array
are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }

        free(ptr);
    }

    return 0;
}
```

Thank you