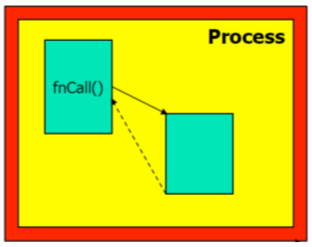
# System Calls for Working with Files

## System Call

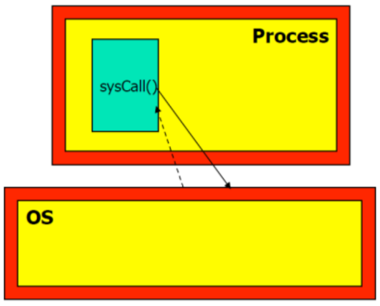
- A system call can be considered as a request to the operating system to perform some activity.
- System Calls versus Function Calls

**Function Call**



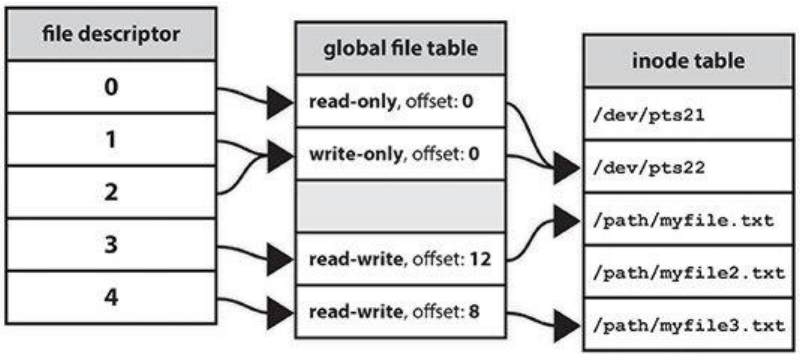
- Caller and callee are in the same process
- Same user
- Same "domain of trust"

**System Call**



- OS is trusted; user is not
- OS has super-privileges; user does not

- File Descriptor
  - A file descriptor is a number or identifier that uniquely identifies each opened file in a computer's operating system.
  - When opening or creating a new file, the system returns a file descriptor to the process that executed the call.
    - When we open a file, the operating system creates an entry to represent that file and store the information about that opened file. So, if there are 100 files opened in our operating system, then there will be 100 entries (somewhere in kernel). These entries are represented by integers like (100, 101, 102). This entry number is the file descriptor.



    - When a process makes a successful request to open a file, the kernel returns a file descriptor which points to an entry in the kernel's global file table. The file table entry contains information such as the inode of the file and the access restrictions for that data stream (read-only, write-only, etc.).

        - The inode (index node) is a data structure in a Unix-style file system that describes a file-system object such as a file or a directory. Each inode stores the attributes and disk block locations of the object's data.
        - Short for index node, an inode is information contained within a Unix-like system with details about each file, such as the node, owner, file, location of file, etc.
    - **stdin**, **stdout**, and **stderr** File Descriptors - In a Unix-like operating system, the first three file descriptors, by default, are STDIN (standard input – **keyboard (file descriptor = 0)**), STDOUT (standard output – **screen (file descriptor = 1)**), and STDERR (standard error – **screen (file descriptor = 2)**).

## OPEN System Call

Opening or creating a file can be done using the system call open. The syntax is:

```
#include <sys/types.h>  //(A Collection of typedef symbols and structures)
#include <sys/stat.h>
#include <fcntl.h>  //(File Control)
int open(const char *path, int flags,/* mode_t mod*/);
```

- This function returns the file descriptor.
- The number of arguments that this function can have is two or three.
- The third argument is used only when creating a new file.
- When we want to open an existing file only two arguments are used.
- The argument **flags** is formed by a bitwise OR (**|**) operation made on the constants defined in the **fcntl.h** header.
    - O_RDONLY - Opens the file for reading.
    - O_WRONLY - Opens the file for writing.
    - O_RDWR - The file is opened for reading and writing.
    - O_APPEND - It writes successively to the end of the file.
    - O_CREAT - The file is created in case it does not already exist.
    - O_TRUNC - If the file exists all of its content will be deleted.
    - … …
- The third argument, **mod**, is a bitwise OR made between a combination of two from the following list (access rights defined in the **sys/stat.h**):
    - S_IRUSR, S_IWUSR, S_IXUSR (Owner: Read, Write, Execute)
    - S_IRGRP, S_IWGRP, S_IXGRP (Group: Read, Write, Execute)
    - S_IROTH, S_IWOTH, S_IXOTH (Other: Read, Write, Execute)

## CREAT System Call

A new file can be created by:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int creat(const char *path, mode_t mod);
```

## READ System Call

When we want to read a certain number of bytes starting from the current position in a file, we use the read call. The syntax is:

`#include <unistd.h>` (Point to various constant, type and function declarations)

```
ssize_t read(int fd, void* buf, size_t noct);
```

- The function returns the number of bytes read.
- It reads **noct** bytes from the open file referred by the **fd** descriptor and it puts it into a buffer **buf**.
- The pointer (current position) is incremented automatically after a reading that certain amount of bytes.
- The process that executes a read operation waits until the system puts the data from the disk into the buffer.

## WRITE System Call

For writing a certain number of bytes into a file starting from the current position we use the write call. Its syntax is:

`#include <unistd.h>` (Point to various constant, type and function declarations)

```
ssize_t write(int fd, const void* buf, size_t noct);
```

- The function returns the number of bytes written and the value -1 in case of an error.
- It writes **noct** bytes from the buffer **buf** into the file that has as its descriptor **fd**.

## CLOSE System Call

For closing a file and thus eliminating the assigned descriptor we use the system call close.

`#include <unistd.h>` (Point to various constant, type and function declarations)

```
int close(int fd);
```

- The function returns 0 in case of success and -1 in case of an error. At the termination of a process an open file is closed anyway.

## ACCESS System Call

When opening a file with system call open the root verifies the access rights in function of the UID and the effective GID.

Sometimes it is necessary to test whether the real user can or cannot access the file. For this we can use access which allows verifying the access rights of a file.

The syntax for this system call is:

```
#include <unistd.h>
int access(const char* path, int mod);
```

- The function returns 0 if the access right exists and -1 otherwise.
- The argument **mod** is a bitwise AND between R_OK (permission to read), W_OK (permission to write), X_OK (execution right), F_OK (the file exists).

## CHMODE System Call

To modify the access rights for an existing file, we use:

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char* path, mode_t mod);
```

- The function returns 0 in case of a success and -1 otherwise.
- The **chmod** call modifies the access rights of the file specified by the path depending on the access rights specified by the **mod** argument.
- To be able to modify the access rights, the effective UID of the process has to be identical to the owner of the file or the process must have root rights.
- The mod argument can be specified by one of the symbolic constants defined in the **sys/stat.h** header.
- Their effect can be obtained by making a bitwise OR operation on them:
- S_IRWXU - Read, write, execute rights for the owner obtained from S_IRUSR | S_IWUSR | S_IXUSR
- S_IRWXG - Read, write, execute rights for the group obtained from S_IRGRP | S_IWGRP | S_IXGRP
- S_IRWXO - Read, write, execute rights for others obtained from S_IROTH | S_IWOTH | S_IXOTH

## UMASK System Call

When creating a file those bits that are set to 1 in the mask invalidate the corresponding bits in the argument that specify the access rights.

The syntax for this system call is:

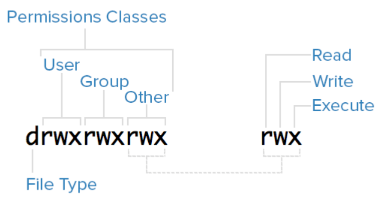
```
#include <sys/types.h>
#include <sys/stat.h>
mode_t umask(mode_t mask);
```

- The function returns the value of the previous mask.
- The effect of the call is shown below:

```
main()
{
  int fd;
  umask(022);
  fd = creat("temp", 0666);
  system("ls -l temp");
}
```

- The result will be of the following form:

```
-rw-r--r-- temp
```

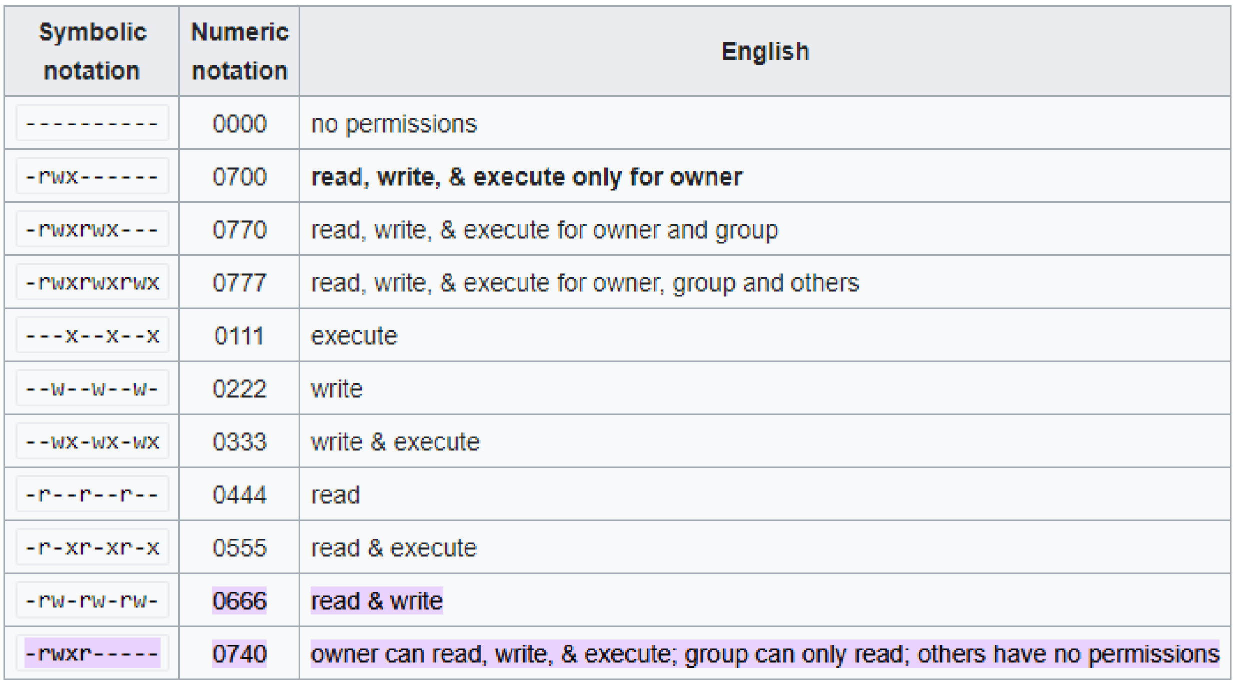


## File Access Example

```
#include<stdio.h>
#include<fcntl.h>
#include<unistd.h>
#include<sys/stat.h>
int main()
{
  char fileName[20];
  int filePermission;

  printf("Enter the File Name: ");
  scanf("%s", fileName);

  printf("\nEnter the Permission: ");
  scanf("%d", &filePermission);

  int fd;
  fd = open(fileName, O_CREAT);

  printf("\nFILE PERMISSIONS BEFORE CHANGE:");
  printf("\nFile Exist: %d", access(fileName,F_OK));
  printf("\nRead Permission: %d", access(fileName,R_OK));
  printf("\nWrite Permission: %d", access(fileName,W_OK));
  printf("\nExecute Permission: %d", access(fileName,X_OK));

  printf("\n\nFILE PERMISSIONS BEFORE CHANGE:");
  chmod(fileName, S_IRWXU);
  printf("\nFile Exist: %d", access(fileName,F_OK));
  printf("\nRead Permission: %d", access(fileName,R_OK));
  printf("\nWrite Permission: %d", access(fileName,W_OK));
  printf("\nExecute Permission: %d", access(fileName,X_OK));

  return(0);
}
```

## File Handling Example

```
#include<stdio.h>
#include<fcntl.h>
#include<unistd.h>
#include<sys/stat.h>
#include<stdlib.h>
int main()
{
  int i=0, ie=0, io=0, n, temp;
  printf("Input the value of n: ");
  scanf("%d", &n);

  char bufAllW[n], bufAllR[n], bufOddW[n], bufOddR[n], bufEvenW[n], 
bufEvenR[n];
  for(i=0; i<n; i++)
  {
    bufAllW[0]=i;
  }

  int fdAll, fdEven, fdOdd;
  fdAll = open("All.txt", O_CREAT | O_WRONLY, S_IRWXU);
  write(fdAll, bufAllW, n);

  read(fdAll, bufAllR, n);

  for(i=0; i<n; i++)
  {
    temp = bufAllR[i];
    if(temp%2 == 0)
    {
      bufEvenW[ie]=bufAllR[i];
      ie++;
    }
    else
    {
      bufOddW[io]=bufAllR[i];
      ie++;
    }
  }
  close(fdAll);
  fdEven = open("Even.txt", O_CREAT | O_RDWR, S_IRWXU);
  write(fdEven, bufEvenW, n/2);
  close(fdEven);

  fdOdd = open("Odd.txt", O_CREAT | O_RDWR, S_IRWXU);
  write(fdOdd, bufOddW, n/2);
  close(fdOdd);

  return 0;
}
```



| Symbolic notation | Numeric notation | English |
|---|---|---|
| `----------` | 0000 | no permissions |
| `-rwx------` | 0700 | **read, write, & execute only for owner** |
| `-rwxrwx---` | 0770 | read, write, & execute for owner and group |
| `-rwxrwxrwx` | 0777 | read, write, & execute for owner, group and others |
| `---x--x--x` | 0111 | execute |
| `--w--w--w-` | 0222 | write |
| `--wx-wx-wx` | 0333 | write & execute |
| `-r--r--r--` | 0444 | read |
| `-r-xr-xr-x` | 0555 | read & execute |
| `-rw-rw-rw-` | 0666 | read & write |
| `-rwxr-----` | 0740 | owner can read, write, & execute; group can only read; others have no permissions |