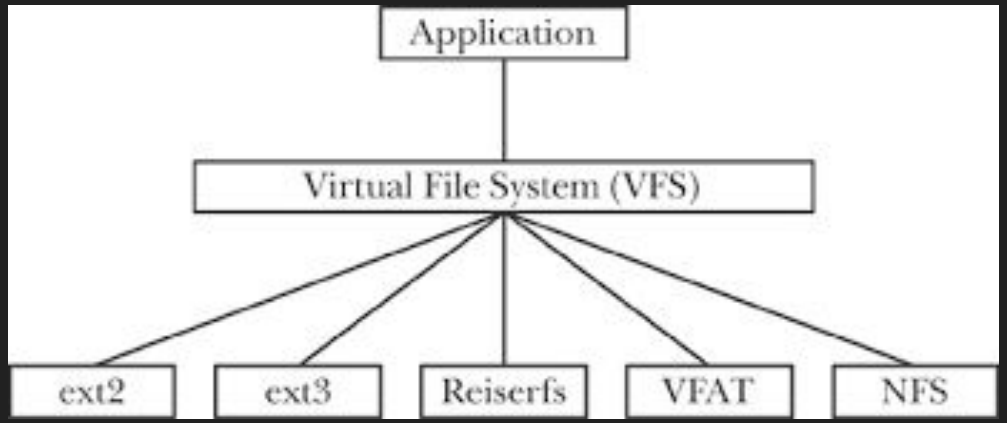
SIS2 - Virtual File System

Done by:
Seitzhan Sanzhar
Bolat Aldiyar
Akzholtayev Kazybek
Kozhakhmetov Adilkhan

Virtual File System

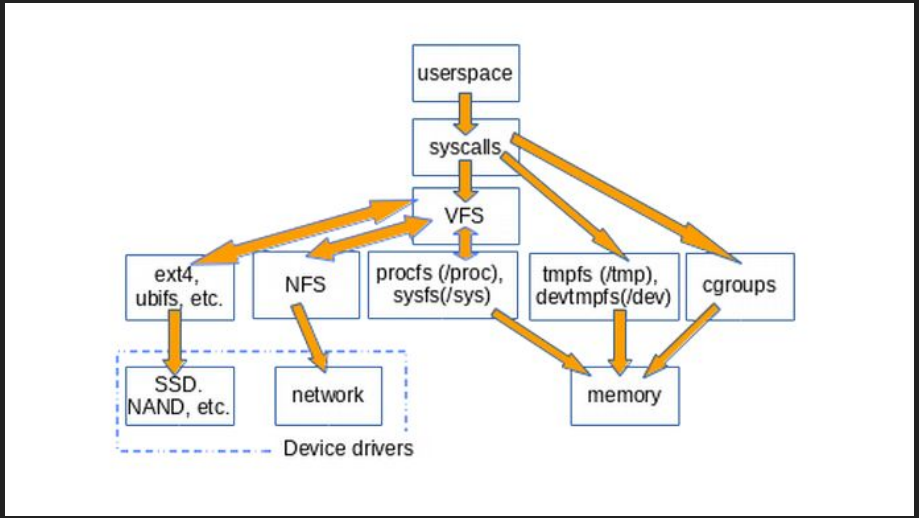
A **virtual file system** (**VFS**) or **virtual filesystem switch** is an abstract layer on top of a more concrete file system. The purpose of a VFS is to allow client applications to access different types of concrete file systems in a uniform way.

A VFS can, for example, be used to access local and network storage devices transparently without the client application noticing the difference.



Filesystem basics

The Linux kernel requires that for an entity to be a filesystem, it must also implement the **open()**, **read()**, and **write()** methods on persistent objects that have names associated with them. From the point of view of object-oriented programming, the kernel treats the generic filesystem as an abstract interface, and these big-three functions are "virtual," with no default definition. Accordingly, the kernel's default filesystem implementation is called a virtual filesystem (VFS).



kazybek@kazybek-VirtualBox: ~/sys-programming/Cshell

File Edit View Search Terminal Help

```
kazybek@kazybek-VirtualBox:~$ cd sys-programming/
kazybek@kazybek-VirtualBox:~/sys-programming$ cd Cshell/
kazybek@kazybek-VirtualBox:~/sys-programming/Cshell$ make clean
rm Cshell myshell
kazybek@kazybek-VirtualBox:~/sys-programming/Cshell$ make
gcc shell.c -o myshell
gcc run.c -o Cshell
kazybek@kazybek-VirtualBox:~/sys-programming/Cshell$ ./myshell
```

Terminal

File Edit View Search Terminal Help

```
+/home/kazybek/sys-programming/Cshell ~>
```

The following functions have been written explicitly in C.

- cd - Change directory
- pwd - Current Working directory
- mkdir
  - Make a folder (Alerts, if folder already exists)
- rmdir
  - Remove the folder (Alerts if no such file or folder exists)
- ls - List contents of pwd
- ls -l - List the contents in long listing format
- exit - Exit the shell ; also works for z char

```c
void about();

void getInput();

int function_exit();

void function_pwd(char *, int);

void function_cd(char *);

void function_mkdir(char *);

void function_rmdir(char *);

void function_clear();

void nameFile(struct dirent *, char *);

void function_ls();

void function_lsl();

void executable();

void pipe_dup(int, instruction *);

void run_process(int, int, instruction *);
```

```c
// Next 2 functions are called by executable() */
/* use execvp to run the command, check path, and handle errors*/
void runprocess(char *cli, char *args[], int count) {
    int ret = execvp(cli, args);
    char *pathm;
    pathm = getenv("PATH");
    char path[1000];
    strcpy(path, pathm);
    strcat(path, ":");
    strcat(path, cwd);
    char *cmd = strtok(path, ":\r\n");
    while (cmd != NULL) {
        char loc_sort[1000];
        strcpy(loc_sort, cmd);
        strcat(loc_sort, "/");
        strcat(loc_sort, cli);
        printf("execvp : %s\n", loc_sort);
        ret = execvp(loc_sort, args);
        if (ret == -1) {
            perror("+--- Error in running executable ");
            exit(0);
        }
        cmd = strtok(NULL, ":\r\n");
    }
}
```

```c
/* executables like ./a.out */
void executable() {
    instruction command[INPBUF];
    int i = 0, j = 1, status;
    char *curr = strsep(&input1, " \t\n");// need to do all over again
    // since we need to identify distinct commands
    command[0].argval[0] = curr;

    while (curr != NULL) {
        curr = strsep(&input1, " \t\n");
        if (curr == NULL) {
            command[i].argval[j++] = curr;
        } else if (strcmp(curr, "|") == 0) {
            command[i].argval[j++] = NULL;
            command[i].argcount = j;
            j = 0;
            i++;// move to the next instruction
        } else if (strcmp(curr, "<") == 0) {
            externalIn = 1;
            curr = strsep(&input1, " \t\n");
            strcpy(inputfile, curr);
        } else if (strcmp(curr, ">") == 0) {
            externalOut = 1;
            curr = strsep(&input1, " \t\n");
            strcpy(outputfile, curr);
        } else if (strcmp(curr, "&") == 0) {
            inBackground = 1;
        } else {
            command[i].argval[j++] = curr;
```

```c
            command[i].argcount = j;
            j = 0;
            i++;// move to the next instruction
        } else if (strcmp(curr, "<") == 0) {
            externalIn = 1;
            curr = strsep(&input1, " \t\n");
            strcpy(inputfile, curr);
        } else if (strcmp(curr, ">") == 0) {
            externalOut = 1;
            curr = strsep(&input1, " \t\n");
            strcpy(outputfile, curr);
        } else if (strcmp(curr, "&") == 0) {
            inBackground = 1;
        } else {
            command[i].argval[j++] = curr;
        }
    }
    command[i].argval[j++] = NULL; // handle last command separately
    command[i].argcount = j;
    i++;

    // parent process waits for execution and then reads from terminal
    filepid = fork();
    if (filepid == 0) {
        pipe_dup(i, command);
    } else {
        if (inBackground == 0) {
            waitpid(filepid, &status, 0);
        } else {
            printf("+--- Process running in inBackground. PID:%d\n", filepid);
        }
    }
    filepid = 0;
    free(input1);
}
```

```c
    command[i].argval[j++] = NULL; // handle last command separately
    command[i].argcount = j;
    i++;

    // parent process waits for execution and then reads from terminal
    filepid = fork();
    if (filepid == 0) {
        pipe_dup(i, command);
    } else {
        if (inBackground == 0) {
            waitpid(filepid, &status, 0);
        } else {
            printf("+--- Process running in inBackground. PID:%d\n", filepid);
        }
    }
    filepid = 0;
    free(input1);
}
```

```c
/*Stop processes if running in terminal, close terminal if only Ctrl+C*/
void stopSignal() {
    if (filepid != 0) {
        int temp = filepid;
        kill(filepid, SIGINT);
        filepid = 0;
    }
}
```

```c
void executable() {
    instruction command[INPBUF];
    int i = 0, j = 1, status;
    char *curr = strsep(&input1, " \t\n");// need to do all over again
    // since we need to identify distinct commands
    command[0].argval[0] = curr;

    while (curr != NULL) {
        curr = strsep(&input1, " \t\n");
        if (curr == NULL) {
            command[i].argval[j++] = curr;
        } else if (strcmp(curr, "|") == 0) {
            command[i].argval[j++] = NULL;
            command[i].argcount = j;
            j = 0;
            i++;// move to the next instruction
        } else if (strcmp(curr, "<") == 0) {
            externalIn = 1;
            curr = strsep(&input1, " \t\n");
            strcpy(inputfile, curr);
        } else if (strcmp(curr, ">") == 0) {
            externalOut = 1;
            curr = strsep(&input1, " \t\n");
            strcpy(outputfile, curr);
        } else if (strcmp(curr, "&") == 0) {
            inBackground = 1;
        } else {
            command[i].argval[j++] = curr;
        }
    }
    command[i].argval[j++] = NULL; // handle last command separately
    command[i].argcount = j;
    i++;
```