SSN COLLEGE OF ENGINEERING, KALAVAKKAM
(An Autonomous Institution, Affiliated to Anna University, Chennai)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

**UCS1411 - OPERATING SYSTEMS LAB**

---

# LAB EXERCISE 1

## Study of System calls and System Commands

## Submission Date:10-03-2022

*Name: Jayannthan P T* *Dept: CSE 'A'* *Roll No.: 205001049*

1. Study the following system calls and system commands (using Linux manual pages)

a. System Commands

1) cp - -i

**Name:** Copy
**Purpose:** Copy SOURCE to DEST, or multiple SOURCE(s) to DIRECTORY.
**Options:** -i, --interactive
prompt before overwrite
**Syntax:** cp [*OPTION*]... *SOURCE... DIRECTORY*
**Example:** cp file1.txt file2.txt

2)mv - -i

**Name:** move
**Purpose:** Rename SOURCE to DEST, or move SOURCE(s) to DIRECTORY.
**Options:** -i, --interactive
prompt before overwrite
**Syntax:** mv [*OPTION*]... *SOURCE... DIRECTORY*
**Example:** mv file1.txt file2.txt

3)ls - -l

**Name:** list
**Purpose:** list directory contents
**Options:** -l use a long listing format
**Syntax:** ls [OPTION]... [FILE]...
**Example:** ls

4)grep - -c,-v

**Name:** Global Regular Expression Print
**Purpose:** print lines that match patterns
**Options:** -c uppress normal output; instead print a count of matching lines for each input file.
-v, --invert-match
option (see below), count non-matching lines
**Syntax:** grep [OPTION...] PATTERNS [FILE...]
**Example:** grep "^hello" file1

5)chmod

**Name:** Change mode
**Purpose:** changes the file mode bits of each given file according to mode, which can be either a symbolic representation of changes to make, or an octal number representing the bit pattern for the new mode bits.
**Options:** -R, --recursive
change files and directories recursively
**Syntax:** chmod [OPTION]... MODE[,MODE]... FILE...
chmod [OPTION]... OCTAL-MODE FILE...
**Example:** chmod 644 file

6)cat

**Name:** Concatenate
**Purpose:** concatenate files and print on the standard output
**Options:** -n number all output lines
**Syntax:** cat [OPTION]... [FILE]
**Example:** cat file1

7)mkdir

**Name:** Make directories
**Purpose:** print lines that match patterns
**Options:** - v print a message for each created directory
**Syntax:** mkdir [OPTION]... DIRECTORY
**Example:** mkdir files

8)rm

**Name:** Remove
**Purpose:** remove files or directories
**Options:** -i prompt before every removal

-R remove directories and their contents recursively

**Syntax:** rm [OPTION]... [FILE]

**Example:** rm files

9) rmdir

**Name:** Remove Directories

**Purpose:** remove empty directories

**Options:** -p remove DIRECTORY and its ancestors

-v output a diagnostic for every directory processed

**Syntax:** rmdir [OPTION]... DIRECTORY

**Example:** rmdir files

10) wc

**Name:** Word Count

**Purpose:** print newline, word, and byte counts for each file

**Options:** -c, print the byte counts

-m print the character counts

-l print the newline counts

**Syntax:** wc [OPTION]... [FILE]

**Example:** wc file

11) who

**Name:** who

**Purpose:** show who is logged on

**Options:** -q, all login names and number of users logged on

-t, last system clock change

**Syntax:** who [*OPTION*]

**Example:** who

**12)** head - -n

**Name:** head

**Purpose:** output the first part of files

**Options:** -n, print the first n lines

**Syntax:** head [*OPTION*]... [*FILE*]

**Example:** `head file`

**13)** tail - -n

**Name:** tail

**Purpose:** output the first part of files

**Options:** -n, print the first n lines
**Syntax:** head [*OPTION*]... [*FILE*]
**Example:** `head file`

**14)** nl

**Name:** number lines
**Purpose:** Write each FILE to standard output, with line numbers added.
**Options:** **-i, line number increment at each line**
**-p, do not reset line numbers for each section**
**Syntax:** nl [OPTION]... [FILE]
**Example:** nl file1

**15)** awk

**Name:**Aho, Weinberger, and Kernighan
**Purpose:** pattern scanning and processing language
**Options**: -F define the input field separator
-f Specify the pathname of the file progfile containing an awk program.
**Syntax:** awk [-F *sepstring*] [-v *assignment*]... *program* [*argument*...]

awk [-F *sepstring*] -f *progfile* [-f *progfile*]... [-v *assignment*]...[*argument*...]

**Example:**awk '{print}' file.txt

b. System Calls

**1)** fork()

**Description:** fork() creates a new process by duplicating the calling process.The new process is referred to as the *child* process. The calling process is referred to as the *parent* process.

**Header File:** unistd.h
**Syntax:** pid_t fork(void);
**Arguments:** none
**Return type:** *Negative Value*: creation of a child process was unsuccessful.
*Zero*: Returned to the newly created child process.
*Positive value*: Returned to parent or caller. The

value contains process ID of newly created child process.

**2)** execl()

**Description:** The *const char *arg* and subsequent ellipses can be thought of as *arg0*, *arg1*, ..., *argn*. Together they describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. The first argument, by convention, should point to the filename associated with the file being executed. The list of arguments *must* be terminated by a null pointer, and, since these are variadic functions, this pointer must be cast *(char *) NULL*.

**Header File:** unistd.h

**Syntax:** int execl(const char **pathname*, const char **arg*, /*, (char *) NULL */);

**Arguments:** char **pathname*, char **arg*

**Return type:** return only if an error has occurred. The return value is -1,

3) getpid()

**Description:** getpid() returns the process ID (PID) of the calling process.

**Header File:** unistd.h

**Syntax:** pid_t getpid(void);

**Arguments:** Nil

**Return type:** returns the process ID of the parent of the current process. It never throws any error therefore is always successful.

4) getppid()

**Description:** getppid() returns the process ID of the parent of the calling process.

**Header File:** unistd.h

**Syntax:** pid_t getppid(void);

**Arguments:** Nil

**Return type:** returns the process ID of the parent of the current process. It never throws any error therefore is always successful.

5) exit()

**Description:** The exit() function causes normal process termination and the least significant byte of *status* is returned to the parent

**Header File:** stdlib.h

**Syntax:** void exit(int *status*);

**Arguments:** Status to return the parent process

**Return type:** No return value

**6)** wait()

**Description:** wait for process to change state

**Header File:** sys/wait.h

**Syntax:** pid_t wait(int *wstatus);

**Arguments:** wstatus - store status information in the int to which it points.

**Return type:** on success, returns the process ID of the terminated child; on failure, -1 is returned.

7) close()

**Description:** close() closes a file descriptor, so that it no longer refers to any file and may be reused.

**Header File:** unistd.h

**Syntax:** int close(int fd);

**Arguments:** fd is the last file descriptor referring to the underlying open file description

**Return type:** close() returns zero on success. On error, -1 is returned

8) opendir()

**Description:** The opendir() function opens a directory stream corresponding to the directory name, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory.

**Header File:** sys/types.h, dirent.h

**Syntax:** DIR *opendir(const char *name);

**Arguments:** directory name

**Return type:** functions return a pointer to the directory stream. On error, NULL is returned

9) readdir()

**Description:** The readdir() function returns a pointer to a *dirent* structure representing the next directory entry in the directory stream pointed to by *dirp*.

**Header File:** dirent.h
**Syntax:** struct dirent *readdir(DIR *dirp);
**Arguments:** directory pointer
**Return type:** returns a pointer to a dirent structure, If the end of the directory stream is reached, NULL is returned and errno is not changed. If an error occurs, NULL is returned and errno is set to indicate the error.

10) open()

**Description:** The open() system call opens the file specified by pathname. If the specified file does not exist, it may be created by open().
**Header File:** fcntl.h
**Syntax:** int open(const char *pathname, int flags, mode_t mode);
int open(const char *pathname, int flags);
**Arguments:** Pathname of the file, The argument flags must include one of the following access modes: O_RDONLY, O_WRONLY, or O_RDWR
**Return type:**The return value of open() is a file descriptor. On error, -1 is returned

11) read()

**Description:** read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf and read from file descriptor
**Header File:** unistd.h
**Syntax:** ssize_t read(int *fd*, void **buf*, size_t *count*);
**Arguments:** file descriptor fd, starting buffer size, read size
**Return type:** On success, the number of bytes read is returned, else -1 is returned

12) write()

**Description:** write() writes up to count bytes from the buffer starting at buf to the file referred to by the file descriptor fd.
**Header File:** unistd.h
**Syntax:** ssize_t write(int *fd*, void **buf*, size_t *count*);

**Arguments:** file descriptor fd, starting buffer size, write size
**Return type:** On success, the number of bytes written is returned, else -1 is returned

13) creat()

**Description:** write() writes up to count bytes from the buffer starting at buf to the file referred to by the file descriptor fd.
**Header File:** sys/stat.h, fcntl.h
**Syntax:** int creat(const char *path, mode_t mode);
**Arguments:** path of file, open mode
**Return type:** On success, the number of bytes read is returned, else -1 is returned

14) sleep()

**Description:** sleep() causes the calling thread to sleep either until the number of real-time seconds specified in seconds have elapsed or until a signal arrives which is not ignored.
**Header File:** unistd.h>
**Syntax:** unsigned int sleep(unsigned int seconds);
**Arguments:** no. of seconds to sleep
**Return type:** Zero if the requested time has elapsed, or the number of seconds left to sleep, if the call was interrupted by a signal handler.

2. Develop a C program to understand the working of fork()

**Algorithm:**

1) Print a line before calling fork() to intimate it executes before calling fork().
2) Call fork() and store the return value in id.
3) If id is equal to zero then print it is a child process.
4) Else print it is a parent process.

**Code:**

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    printf("Before forking- this line is printed once\n\n");
    int id = fork();
    printf("After forking\n");
    if (id == 0)
        printf("Child process underway\n\n");
```

```
    else
        printf("Parent process underway\n\n");
    return 0;
}
```

**Output:**

```
Before forking...

After forking
Parent process underway

After forking
Child process underway
```

3. Develop a C program using system calls to open a file, read the contents of the same, display it and close the file. Use command line arguments to pass the file name to the program

**Algorithm:**

1) If argc greater than 2, then print error : too many arguments
2) Else if argc is lesser than 1, then print error : arguments required
3) Else
   i. Open file using call open() using filename as argument provided in read-only mode and store the file pointer in file_descriptor
   ii. If file_descriptor is equal to -1 then print error and exit
   iii. Else then read the contents using call read() and store the return value in contents
   iv. Print the contents
   v. Close the file

**Code:**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
int main(int argc, char *argv[])
{

    if (argc > 2)
        printf("Too many arguements\n");
    else if (argc < 1)
        printf("Atleast one arguement required\n");
    else
    {
        int file_descriptor = open(argv[1], O_RDONLY);
        if (file_descriptor == -1)
            printf("File does not exist\n");
```

```
        else
        {
            printf("File descriptor is: %d\n", file_descriptor);
            char contents[100];
            read(file_descriptor, contents, 100);
            printf("File contents : %s\n", contents);
            close(file_descriptor);
        }
    }
    return 0;
}
```

**Output:**

```
File descriptor is: 3
File contents : #include <stdio.h>

int main(void) {
  printf("Hello World\n");
  return 0;
}
```

**Learning Outcome:**

- Learned system commands and system calls
- Implemented fork() in C program