****

**BSPML’s**

**JAIKRANTI COLLEGE OF COMPUTER SCIENCE AND MANAGEMENT STUDIES, KATRAJ**

**CERTIFICATE**

This is to certify that **“ Pawar Swapnil ”** student of Jaikranti College of Computer Science and Management Studies, Katraj has successfully completed Lab course **“ OS Practical ”** which was carried out in partial fulfilment for the post degree of M.Sc. (Computer science) of Savitribai Phule Pune University.

**Practical in charge HOD**

**Internal Examiner External Examiner**

**Date:**

**Place: Pune**

**Index**

|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **Program Title** | **Sign** |
| **1** | **To create ‘n’ children. When the children will terminate, display total cumulative time children spent in user and kernel mode.** |  |
| **2** | **To generate parent process to write unnamed pipe and will read from it.** |  |
| **3** | **To create a file with hole in it.** |  |
| **4** | **Takes multiple files as Command Line Arguments and print their inode number.** |  |
| **5** | **To handle the two-way communication between parent and child using pipe.** |  |
| **6** | **Print the type of file where file name accepted through Command Line.** |  |
| **7** | **To demonstrate the use of atexit() function.** |  |
| **8** | **Open a file goes to sleep for 15 seconds before terminating.** |  |
| **9** | **To print the size of the file.** |  |
| **10** | **Read the current directory and display the name of the files, no of files in current directory.** |  |
| **11** | **Write a C program to implement the following unix/linux command (use fork, pipe and exec system call) ls –l | wc –l** |  |
| **12** | **Write a C program to display all the files from current directory which are created in particular month.** |  |
| **13** | **Write a C program to display all the files from current directory whose size is greater that n Bytes Where n is accept from user.** |  |
| **14** | **Write a C program to implement the following unix/linux command**  **i. ls –l > output.txt** |  |
| **15** | **Write a C program which display the information of a given file similar to given by the unix / linux command ls –l** |  |
| **16** | **Write a C program that behaves like a shell (command interpreter). It has its own prompt say “NewShell$”. Any normal shell command is executed from your shell by starting a child process to execute the system program corresponding to the command. It should additionally interpret the following command.**  **i) count c - print number of characters in file**  **ii) count w - print number of words in file**  **iii) count l - print number of lines in file** |  |
| **17** | **Write a C program that behaves like a shell (command interpreter). It has its own prompt say “NewShell$”. Any normal shell command is executed from your shell by starting a child process to execute the system program corresponding to the command. It should additionally interpret the following command.**  **i) list f - print name of all files in directory**  **ii) list n - print number of all entries**  **iii) list i - print name and inode of all files** |  |
| **18** | **Write a C program that behaves like a shell (command interpreter). It has its own prompt say “NewShell$”. Any normal shell command is executed from your shell by starting a child process to execute the system program corresponding to the command. It should additionally interpret the following command.**  **i) typeline +10 - print first 10 lines of file**  **ii) typeline -20 - print last 20 lines of file**  **iii) typeline a - print all lines of file** |  |
| **19** | **Write a C program that behaves like a shell (command interpreter). It has its own prompt say “NewShell$”.Any normal shell command is executed from your shell by starting a child process to execute the system program corresponding to the command. It should**  **i) additionally interpret the following command.**  **ii) search f - search first occurrence of pattern in filename**  **iii) search c - count no. of occurrences of pattern in filename**  **iv) search a - search all occurrences of pattern in filename** |  |
| **20** | **Write a C program which receives file names as command line arguments and display those filenames in ascending order according to their sizes. i)**  **(e.g $ a.out a.txt b.txt c.txt, …)** |  |
| **21** | **Write a C program which create a child process which catch a signal sighup, sigint and sigquit. The Parent process send a sighup or sigint signal after every 3 seconds, at the end of 30 second parent send sigquit signal to child and child terminates my displaying message "My DADDY has Killed me!!!”.** |  |
| **22** | **Write a C program to implement the following unix/linux command (use fork, pipe and exec system call). Your program should block the signal Ctrl-C and Ctrl-\ signal during the execution. i. ls –l | wc –l** |  |
| **23** | **Write a C Program that demonstrates redirection of standard output to a file** |  |
| **24** | **Write a C program that illustrates how to execute two commands concurrently with a pipe.** |  |
| **25** | **Write a C program that illustrates suspending and resuming processes using signals.** |  |
| **26** | **Write a C program that illustrates inters process communication using shared memory** |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

**1. To create ‘n’ children. When the children will terminate, display total cumulative time children spent in user and kernel mode.**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

#include <sys/times.h>

int main(int argc, char \*argv[]) {

if (argc != 2) {

printf("Usage: %s n\n", argv[0]);

return 1;

}

int n = atoi(argv[1]);

pid\_t pid;

struct tms start, end;

clock\_t user\_time = 0, kernel\_time = 0;

for (int i = 0; i < n; i++) {

pid = fork();

if (pid < 0) {

printf("Error: fork() failed\n");

return 1;

} else if (pid == 0) { // child process

printf("Child %d started\n", i);

exit(0);

}

}

// parent process

int status;

while ((pid = wait(&status)) > 0) {

if (WIFEXITED(status)) {

printf("Child %d exited normally with status %d\n", pid, WEXITSTATUS(status));

} else {

printf("Child %d exited abnormally\n", pid);

}

times(&end);

user\_time += end.tms\_cutime - start.tms\_cutime;

kernel\_time += end.tms\_cstime - start.tms\_cstime;

}

printf("Total user time: %ld\n", user\_time);

printf("Total kernel time: %ld\n", kernel\_time);

return 0;

}

**2. To generate parent process to write unnamed pipe and will read from it.**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#define BUFFER\_SIZE 1024

int main() {

int fd[2]; // file descriptors for pipe

char buffer[BUFFER\_SIZE];

pid\_t pid;

if (pipe(fd) < 0) {

printf("Error: pipe() failed\n");

return 1;

}

pid = fork();

if (pid < 0) {

printf("Error: fork() failed\n");

return 1;

} else if (pid == 0) { // child process

close(fd[1]); // close write end of pipe

while (read(fd[0], buffer, BUFFER\_SIZE) > 0) {

printf("Child process received: %s", buffer);

}

close(fd[0]); // close read end of pipe

exit(0);

} else { // parent process

close(fd[0]); // close read end of pipe

char \*msg = "Hello, child process!\n";

write(fd[1], msg, BUFFER\_SIZE);

close(fd[1]); // close write end of pipe

wait(NULL); // wait for child to exit

}

return 0;

}

**3. To create a file with hole in it.**

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <unistd.h>

#define FILENAME "file\_with\_hole.txt"

int main() {

int fd;

char buffer1[10] = "abcdefghi";

char buffer2[10] = "123456789";

fd = open(FILENAME, O\_WRONLY | O\_CREAT | O\_TRUNC, 0666);

if (fd < 0) {

printf("Error: open() failed\n");

return 1;

}

write(fd, buffer1, 9); // write first buffer to file

lseek(fd, 1000, SEEK\_CUR); // create hole of 1000 bytes

write(fd, buffer2, 9); // write second buffer to file

close(fd);

return 0;

}

**4. Takes multiple files as Command Line Arguments and print their inode number.**

#include <stdio.h>

#include <stdlib.h>

#include <sys/stat.h>

int main(int argc, char \*argv[]) {

int i;

struct stat file\_stat;

for (i = 1; i < argc; i++) {

if (stat(argv[i], &file\_stat) < 0) {

printf("Error: stat() failed for file %s\n", argv[i]);

continue;

}

printf("Inode number of file %s: %ld\n", argv[i], file\_stat.st\_ino);

}

return 0;

}

**5. To handle the two-way communication between parent and child using pipe.**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#define BUFFER\_SIZE 25

#define READ\_END 0

#define WRITE\_END 1

int main() {

char write\_msg[BUFFER\_SIZE] = "Hello, child!";

char read\_msg[BUFFER\_SIZE];

int fd[2];

pid\_t pid;

if (pipe(fd) == -1) {

fprintf(stderr, "Pipe failed");

return 1;

}

pid = fork();

if (pid < 0) {

fprintf(stderr, "Fork failed");

return 1;

}

if (pid > 0) { // parent process

close(fd[READ\_END]);

write(fd[WRITE\_END], write\_msg, BUFFER\_SIZE);

printf("Parent sent message: %s\n", write\_msg);

close(fd[WRITE\_END]);

} else { // child process

close(fd[WRITE\_END]);

read(fd[READ\_END], read\_msg, BUFFER\_SIZE);

printf("Child received message: %s\n", read\_msg);

close(fd[READ\_END]);

}

return 0;

}

**6. Print the type of file where file name accepted through Command Line.**

#include <stdio.h>

#include <stdlib.h>

#include <sys/stat.h>

int main(int argc, char \*argv[]) {

struct stat file\_stat;

if (argc != 2) {

printf("Usage: %s <filename>\n", argv[0]);

return 1;

}

if (stat(argv[1], &file\_stat) < 0) {

printf("Error: stat() failed for file %s\n", argv[1]);

return 1;

}

if (S\_ISREG(file\_stat.st\_mode)) {

printf("%s is a regular file\n", argv[1]);

} else if (S\_ISDIR(file\_stat.st\_mode)) {

printf("%s is a directory\n", argv[1]);

} else if (S\_ISCHR(file\_stat.st\_mode)) {

printf("%s is a character device\n", argv[1]);

} else if (S\_ISBLK(file\_stat.st\_mode)) {

printf("%s is a block device\n", argv[1]);

} else if (S\_ISFIFO(file\_stat.st\_mode)) {

printf("%s is a FIFO/pipe\n", argv[1]);

} else if (S\_ISSOCK(file\_stat.st\_mode)) {

printf("%s is a socket\n", argv[1]);

} else if (S\_ISLNK(file\_stat.st\_mode)) {

printf("%s is a symbolic link\n", argv[1]);

} else {

printf("%s is an unknown file type\n", argv[1]);

}

return 0;

}

**7. To demonstrate the use of atexit() function.**

#include <stdio.h>

#include <stdlib.h>

void cleanup() {

printf("Cleaning up...\n");

}

int main() {

int i;

for (i = 0; i < 5; i++) {

printf("Loop iteration %d\n", i);

}

atexit(cleanup);

printf("Exiting...\n");

return 0;

}

**8. Open a file goes to sleep for 15 seconds before terminating.**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

int main() {

int fd;

fd = open("myfile.txt", O\_RDONLY);

if (fd < 0) {

perror("open");

exit(1);

}

printf("File opened successfully\n");

sleep(15);

printf("Terminating...\n");

close(fd);

return 0;

}

**9. To print the size of the file.**

#include <stdio.h>

#include <stdlib.h>

#include <sys/stat.h>

int main(int argc, char \*argv[]) {

if (argc != 2) {

fprintf(stderr, "Usage: %s <filename>\n", argv[0]);

exit(1);

}

struct stat st;

if (stat(argv[1], &st) == -1) {

perror("stat");

exit(1);

}

printf("Size of %s is %lld bytes\n", argv[1], (long long)st.st\_size);

return 0;

}

**10. Read the current directory and display the name of the files, no of files in current directory.**

#include <stdio.h>

#include <stdlib.h>

#include <dirent.h>

int main() {

DIR \*dir;

struct dirent \*entry;

int count = 0;

dir = opendir(".");

if (!dir) {

perror("opendir");

exit(1);

}

while ((entry = readdir(dir)) != NULL) {

if (entry->d\_type == DT\_REG) { // check if the entry is a regular file

printf("%s\n", entry->d\_name);

count++;

}

}

printf("Number of files in current directory: %d\n", count);

closedir(dir);

return 0;

}

**11. Write a C program to implement the following unix/linux command (use fork, pipe and exec system call) ls –l | wc –l**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

int main() {

int fd[2];

pid\_t pid;

int status;

if (pipe(fd) == -1) {

perror("pipe");

exit(1);

}

pid = fork();

if (pid == -1) {

perror("fork");

exit(1);

} else if (pid == 0) {

// Child process

close(fd[0]); // close the read end of the pipe

if (dup2(fd[1], STDOUT\_FILENO) == -1) {

perror("dup2");

exit(1);

}

if (execlp("ls", "ls", "-l", NULL) == -1) {

perror("execlp");

exit(1);

}

} else {

// Parent process

close(fd[1]); // close the write end of the pipe

if (dup2(fd[0], STDIN\_FILENO) == -1) {

perror("dup2");

exit(1);

}

if (execlp("wc", "wc", "-l", NULL) == -1) {

perror("execlp");

exit(1);

}

}

return 0;

}

**12. Write a C program to display all the files from current directory which are created in particular month.**

#include <stdio.h>

#include <dirent.h>

#include <sys/stat.h>

#include <time.h>

int main(int argc, char \*argv[]) {

DIR \*dir;

struct dirent \*entry;

struct stat info;

char \*month\_str = argv[1];

int month;

time\_t now;

struct tm \*timeinfo;

// Convert month string to integer

if (sscanf(month\_str, "%d", &month) != 1) {

printf("Invalid month: %s\n", month\_str);

return 1;

}

// Open current directory

dir = opendir(".");

if (!dir) {

perror("opendir");

return 1;

}

// Get current time

time(&now);

timeinfo = localtime(&now);

// Iterate over directory entries

while ((entry = readdir(dir)) != NULL) {

// Get file info

if (stat(entry->d\_name, &info) == -1) {

perror("stat");

continue;

}

// Check if file was created in specified month

if (timeinfo->tm\_year == info.st\_mtime / 31536000 &&

timeinfo->tm\_mon - 1 == month &&

S\_ISREG(info.st\_mode)) {

printf("%s\n", entry->d\_name);

}

} // Close directory

closedir(dir);

return 0;

}

**13. Write a C program to display all the files from current directory whose size is greater that n Bytes Where n is accept from user.**

#include <stdio.h>

#include <dirent.h>

#include <sys/stat.h>

int main(int argc, char \*argv[]) {

DIR \*dir;

struct dirent \*entry;

struct stat info;

long size\_threshold;

char \*size\_str;

// Check command-line arguments

if (argc != 2) {

printf("Usage: %s <size in bytes>\n", argv[0]);

return 1;

}

size\_str = argv[1];

if (sscanf(size\_str, "%ld", &size\_threshold) != 1) {

printf("Invalid size: %s\n", size\_str);

return 1;

}

// Open current directory

dir = opendir(".");

if (!dir) {

perror("opendir");

return 1;

}

// Iterate over directory entries

while ((entry = readdir(dir)) != NULL) {

// Get file info

if (stat(entry->d\_name, &info) == -1) {

perror("stat");

continue;

}

// Check if file size is greater than threshold

if (info.st\_size > size\_threshold) {

printf("%s\n", entry->d\_name);

}

}

// Close directory

closedir(dir);

return 0;

}

**14. Write a C program to implement the following unix/linux command**

**i. ls –l > output.txt**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

int main() {

int fd, saved\_stdout;

// Open the output file

fd = open("output.txt", O\_CREAT | O\_TRUNC | O\_WRONLY, 0644);

if (fd == -1) {

perror("open");

exit(EXIT\_FAILURE);

}

// Save the standard output file descriptor

saved\_stdout = dup(STDOUT\_FILENO);

if (saved\_stdout == -1) {

perror("dup");

exit(EXIT\_FAILURE);

}

// Redirect standard output to the output file

if (dup2(fd, STDOUT\_FILENO) == -1) {

perror("dup2");

exit(EXIT\_FAILURE);

}

// Execute the ls command with the -l option

execlp("ls", "ls", "-l", NULL);

perror("execlp");

exit(EXIT\_FAILURE);

// Restore standard output

if (dup2(saved\_stdout, STDOUT\_FILENO) == -1) {

perror("dup2");

exit(EXIT\_FAILURE);

}

// Close the output file

if (close(fd) == -1) {

perror("close");

exit(EXIT\_FAILURE);

}

return 0;

}

**15. Write a C program which display the information of a given file similar to given by the unix / linux command ls –l**

#include <stdio.h>

#include <sys/stat.h>

#include <stdlib.h>

#include <unistd.h>

#include <time.h>

int main(int argc, char \*argv[]) {

struct stat file\_stat;

if (argc != 2) {

printf("Usage: %s <filename>\n", argv[0]);

exit(EXIT\_FAILURE);

}

if (stat(argv[1], &file\_stat) == -1) {

perror("stat");

exit(EXIT\_FAILURE);

}

printf("File: %s\n", argv[1]);

printf("Size: %ld bytes\n", file\_stat.st\_size);

printf("Mode: %o\n", file\_stat.st\_mode & 07777);

printf("User ID: %d\n", file\_stat.st\_uid);

printf("Group ID: %d\n", file\_stat.st\_gid);

printf("Access time: %s", ctime(&file\_stat.st\_atime));

printf("Modification time: %s", ctime(&file\_stat.st\_mtime));

printf("Status change time: %s", ctime(&file\_stat.st\_ctime));

return 0;

}

**16. Write a C program that behaves like a shell (command interpreter). It has its own prompt say “NewShell$”. Any normal shell command is executed from your shell by starting a child process to execute the system program corresponding to the command. It should additionally interpret the following command.**

**i) count c - print number of characters in file**

**ii) count w - print number of words in file**

**iii) count l - print number of lines in file**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/wait.h>

#include <sys/stat.h>

#define MAX\_COMMAND\_LENGTH 100

#define MAX\_ARGUMENTS 10

int main() {

char command[MAX\_COMMAND\_LENGTH];

char \*args[MAX\_ARGUMENTS];

int status;

while (1) {

printf("NewShell$ ");

fgets(command, MAX\_COMMAND\_LENGTH, stdin);

// Remove the newline character at the end of the command

command[strcspn(command, "\n")] = 0;

// Tokenize the command into arguments

char \*arg = strtok(command, " ");

int i = 0;

while (arg != NULL && i < MAX\_ARGUMENTS - 1) {

args[i++] = arg;

arg = strtok(NULL, " ");

}

args[i] = NULL;

// Check if the command is "count c", "count w", or "count l"

if (i == 3 && strcmp(args[0], "count") == 0) {

struct stat st;

if (stat(args[2], &st) != 0) {

perror("stat");

continue;

}

int count = 0;

switch (args[1][0]) {

case 'c':

count = st.st\_size;

break;

case 'w':

// Count the number of words in the file

FILE \*fp = fopen(args[2], "r");

if (fp == NULL) {

perror("fopen");

continue;

}

int in\_word = 0;

int c;

while ((c = fgetc(fp)) != EOF) {

if (c == ' ' || c == '\n' || c == '\t') {

if (in\_word) {

count++;

in\_word = 0;

}

} else {

in\_word = 1;

}

}

if (in\_word) {

count++;

}

fclose(fp);

break;

case 'l':

// Count the number of lines in the file

fp = fopen(args[2], "r");

if (fp == NULL) {

perror("fopen");

continue;

}

in\_word = 0;

while ((c = fgetc(fp)) != EOF) {

if (c == '\n') {

count++;

}

}

fclose(fp);

break;

default:

printf("Invalid count command\n");

continue;

}

printf("%d\n", count);

} else {

// Create a child process to execute the command

pid\_t pid = fork();

if (pid == 0) {

execvp(args[0], args);

perror("execvp");

exit(1);

} else if (pid > 0) {

waitpid(pid, &status, 0);

} else {

perror("fork");

}}}

Return 0; }

**17. Write a C program that behaves like a shell (command interpreter). It has its own prompt say “NewShell$”. Any normal shell command is executed from your shell by starting a child process to execute the system program corresponding to the command. It should additionally interpret the following command.**

**i) list f - print name of all files in directory**

**ii) list n - print number of all entries**

**iii) list i - print name and inode of all files**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

#include <string.h>

#include <dirent.h>

#include <sys/stat.h>

#define MAX\_COMMAND\_LENGTH 1024

#define MAX\_TOKENS 100

void print\_error(char \*msg) {

perror(msg);

exit(1);

}

void execute\_command(char \*\*args) {

pid\_t pid = fork();

if (pid < 0) {

print\_error("fork() failed");

} else if (pid == 0) {

if (execvp(args[0], args) < 0) {

print\_error("execvp() failed");

}

} else {

wait(NULL);

}

}

void list\_files() {

DIR \*dir = opendir(".");

if (!dir) {

print\_error("opendir() failed");

}

struct dirent \*entry;

while ((entry = readdir(dir)) != NULL) {

if (entry->d\_type == DT\_REG) {

printf("%s\n", entry->d\_name);

}

}

closedir(dir);

}

void list\_entries() {

DIR \*dir = opendir(".");

if (!dir) {

print\_error("opendir() failed");

}

int count = 0;

struct dirent \*entry;

while ((entry = readdir(dir)) != NULL) {

count++;

}

printf("Number of entries: %d\n", count);

closedir(dir);

}

void list\_inodes() {

DIR \*dir = opendir(".");

if (!dir) {

print\_error("opendir() failed");

}

struct dirent \*entry;

while ((entry = readdir(dir)) != NULL) {

if (entry->d\_type == DT\_REG) {

struct stat st;

char filename[MAX\_COMMAND\_LENGTH];

sprintf(filename, "%s/%s", ".", entry->d\_name);

if (stat(filename, &st) == 0) {

printf("%s %lu\n", entry->d\_name, st.st\_ino);

} else {

print\_error("stat() failed");

}

}

}

closedir(dir);

}

int main() {

char command[MAX\_COMMAND\_LENGTH];

char \*tokens[MAX\_TOKENS];

char \*delim = " \t\n";

while (1) {

printf("NewShell$ ");

if (fgets(command, MAX\_COMMAND\_LENGTH, stdin) == NULL) {

printf("\n");

exit(0);

}

int num\_tokens = 0;

tokens[num\_tokens] = strtok(command, delim);

while (tokens[num\_tokens] != NULL) {

num\_tokens++;

tokens[num\_tokens] = strtok(NULL, delim);

}

if (num\_tokens == 0) {

continue;

}

if (strcmp(tokens[0], "list") == 0) {

if (num\_tokens < 2) {

printf("Usage: list f|n|i\n");

continue;

}

if (strcmp(tokens[1], "f") == 0) {

list\_files();

} else if (strcmp(tokens[1], "n") == 0) {

list\_entries();

} else if (strcmp(tokens[1], "i") == 0) {

list\_inodes();

} else {

printf("Invalid option: %s\n", tokens[1]);

}

} else {

execute\_command(tokens);

}

}

return 0;

}

**18. Write a C program that behaves like a shell (command interpreter). It has its own prompt say “NewShell$”. Any normal shell command is executed from your shell by starting a child process to execute the system program corresponding to the command. It should additionally interpret the following command.**

**i) typeline +10 - print first 10 lines of file**

**ii) typeline -20 - print last 20 lines of file**

**iii) typeline a - print all lines of file**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <fcntl.h>

#define MAX\_ARGS 10

#define BUFFER\_SIZE 1024

void execute\_typeline(char \*filename, char \*mode, int n) {

FILE \*fp;

char buffer[BUFFER\_SIZE];

int line\_count = 0;

int i;

fp = fopen(filename, "r");

if (fp == NULL) {

perror("Cannot open file");

return;

}

if (strcmp(mode, "+") == 0) {

for (i = 0; i < n; i++) {

if (fgets(buffer, BUFFER\_SIZE, fp) == NULL) {

break;

}

printf("%s", buffer);

}

} else if (strcmp(mode, "-") == 0) {

fseek(fp, -n, SEEK\_END);

while (fgets(buffer, BUFFER\_SIZE, fp) != NULL) {

printf("%s", buffer);

}

} else if (strcmp(mode, "a") == 0) {

while (fgets(buffer, BUFFER\_SIZE, fp) != NULL) {

printf("%s", buffer);

}

} else {

printf("Invalid mode\n");

}

fclose(fp);

}

int main() {

char buffer[BUFFER\_SIZE];

char \*args[MAX\_ARGS];

int i, n;

pid\_t pid;

int status;

while (1) {

printf("NewShell$ ");

fflush(stdout);

fgets(buffer, BUFFER\_SIZE, stdin);

buffer[strlen(buffer) - 1] = '\0';

// Parse the command line

n = 0;

args[n] = strtok(buffer, " ");

while (args[n] != NULL) {

n++;

args[n] = strtok(NULL, " ");

}

if (n == 0) {

continue;

}

// Check if the command is typeline

if (strcmp(args[0], "typeline") == 0) {

if (n < 3) {

printf("Usage: typeline [+|-|a] <file> <n>\n");

continue;

}

if (strcmp(args[1], "+") != 0 && strcmp(args[1], "-") != 0 && strcmp(args[1], "a") != 0) {

printf("Invalid mode\n");

continue;

}

execute\_typeline(args[2], args[1], atoi(args[3]));

continue;

}

// Fork a child process

pid = fork();

if (pid == -1) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (pid == 0) {

// Child process

// Block Ctrl-C and Ctrl-\

signal(SIGINT, SIG\_IGN);

signal(SIGQUIT, SIG\_IGN);

// Redirect stdout if necessary

for (i = 1; i < n - 1; i++) {

if (strcmp(args[i], ">") == 0) {

int fd = open(args[i+1], O\_WRONLY | O\_CREAT | O\_TRUNC, 0644);

if (fd == -1) {

perror("open");

exit(EXIT\_FAILURE);

}

dup2(fd, STDOUT\_FILENO);

close(fd);

args[i] = NULL;

n = i;

break;

}

}// Execute the command

**19. Write a C program that behaves like a shell (command interpreter). It has its own prompt say “NewShell$”.Any normal shell command is executed from your shell by starting a child process to execute the system program corresponding to the command. It should**

**i) additionally interpret the following command.**

**ii) search f - search first occurrence of pattern in filename**

**iii) search c - count no. of occurrences of pattern in filename**

**iv) search a - search all occurrences of pattern in filename**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/wait.h>

#define MAX\_ARGS 10

#define MAX\_COMMAND\_LENGTH 100

#define MAX\_FILENAME\_LENGTH 50

#define MAX\_PATTERN\_LENGTH 50

int main() {

char command[MAX\_COMMAND\_LENGTH];

char \*args[MAX\_ARGS];

char filename[MAX\_FILENAME\_LENGTH];

char pattern[MAX\_PATTERN\_LENGTH];

while (1) {

// Display prompt

printf("NewShell$ ");

fflush(stdout);

// Read command from user

fgets(command, MAX\_COMMAND\_LENGTH, stdin);

command[strcspn(command, "\n")] = 0; // Remove trailing newline

// Tokenize command into arguments

int num\_args = 0;

args[num\_args] = strtok(command, " ");

while (args[num\_args] != NULL && num\_args < MAX\_ARGS - 1) {

num\_args++;

args[num\_args] = strtok(NULL, " ");

}

args[num\_args] = NULL;

// Check if command is a built-in command

if (strcmp(args[0], "search") == 0) {

// Check if filename and pattern are provided

if (num\_args != 3) {

printf("Usage: search <f|c|a> <filename> <pattern>\n");

continue;

}

char \*mode = args[1];

strncpy(filename, args[2], MAX\_FILENAME\_LENGTH);

strncpy(pattern, args[3], MAX\_PATTERN\_LENGTH);

// Fork a child process to execute the search command

pid\_t pid = fork();

if (pid == -1) {

printf("Error: Failed to fork process\n");

continue;

} else if (pid == 0) {

// Child process

if (strcmp(mode, "f") == 0) {

// Search for first occurrence of pattern

execlp("grep", "grep", "-m", "1", pattern, filename, NULL);

} else if (strcmp(mode, "c") == 0) {

// Count number of occurrences of pattern

execlp("grep", "grep", "-c", pattern, filename, NULL);

} else if (strcmp(mode, "a") == 0) {

// Search for all occurrences of pattern

execlp("grep", "grep", pattern, filename, NULL);

} else {

printf("Error: Invalid search mode\n");

exit(1);

}

} else {

// Parent process

wait(NULL);

}

} else {

// Fork a child process to execute the command

pid\_t pid = fork();

if (pid == -1) {

printf("Error: Failed to fork process\n");

continue;

} else if (pid == 0) {

// Child process

execvp(args[0], args);

printf("Error: Failed to execute command\n");

exit(1);

} else {

// Parent process

wait(NULL);

}

}

}

return 0;

}

**20. Write a C program which receives file names as command line arguments and display those filenames in ascending order according to their sizes. i)**

**(e.g $ a.out a.txt b.txt c.txt, …)**

#include <stdio.h>

#include <stdlib.h>

#include <sys/stat.h>

int compare(const void \*a, const void \*b) {

struct stat s1, s2;

stat(\*(const char\*\*)a, &s1);

stat(\*(const char\*\*)b, &s2);

return s1.st\_size - s2.st\_size;

}

int main(int argc, char \*argv[]) {

if(argc < 2) {

printf("Usage: %s <file1> <file2> ... <fileN>\n", argv[0]);

exit(EXIT\_FAILURE);

}

qsort(argv+1, argc-1, sizeof(char\*), compare);

for(int i=1; i<argc; i++) {

printf("%s\n", argv[i]);

}

return 0;

}

**21. Write a C program which create a child process which catch a signal sighup, sigint and sigquit. The Parent process send a sighup or sigint signal after every 3 seconds, at the end of 30 second parent send sigquit signal to child and child terminates my displaying message "My DADDY has Killed me!!!”.**

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

#include <sys/types.h>

#include <stdlib.h>

void sighup(); /\* routine for handling SIGHUP signal \*/

void sigint(); /\* routine for handling SIGINT signal \*/

void sigquit(); /\* routine for handling SIGQUIT signal \*/

int main()

{

pid\_t pid;

/\* create a child process \*/

if ((pid = fork()) < 0) {

perror("fork");

exit(1);

}

if (pid == 0) { /\* child \*/

signal(SIGHUP, sighup); /\* catch SIGHUP \*/

signal(SIGINT, sigint); /\* catch SIGINT \*/

signal(SIGQUIT, sigquit); /\* catch SIGQUIT \*/

for (;;) ; /\* loop for child \*/

}

else { /\* parent \*/

printf("\nParent sleeping for 30 seconds\n");

sleep(30);

/\* send SIGHUP signal to child \*/

printf("\nSending SIGHUP signal to child\n");

kill(pid, SIGHUP);

sleep(3);

/\* send SIGINT signal to child \*/

printf("\nSending SIGINT signal to child\n");

kill(pid, SIGINT);

sleep(3);

/\* send SIGQUIT signal to child \*/

printf("\nSending SIGQUIT signal to child\n");

kill(pid, SIGQUIT);

sleep(3);

}

}

void sighup()

{

signal(SIGHUP, sighup); /\* reset signal \*/

printf("Child: I have received a SIGHUP signal\n");

}

void sigint()

{

signal(SIGINT, sigint); /\* reset signal \*/

printf("Child: I have received a SIGINT signal\n");

}

void sigquit()

{

printf("Child: My DADDY has Killed me!!!\n");

exit(0);

}

**22. Write a C program to implement the following unix/linux command (use fork, pipe and exec system call). Your program should block the signal Ctrl-C and Ctrl-\ signal during the execution. i. ls –l | wc –l**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <signal.h>

#include <sys/wait.h>

void sig\_handler(int sig) {}

int main() {

// Block SIGINT and SIGQUIT signals

struct sigaction sa;

sa.sa\_handler = sig\_handler;

sigemptyset(&sa.sa\_mask);

sa.sa\_flags = 0;

sigaction(SIGINT, &sa, NULL);

sigaction(SIGQUIT, &sa, NULL);

// Create pipe

int fd[2];

if (pipe(fd) == -1) {

perror("pipe");

exit(EXIT\_FAILURE);

}

// Fork process

pid\_t pid = fork();

if (pid == -1) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (pid == 0) { // Child process

// Close read end of pipe

close(fd[0]);

// Duplicate write end of pipe to stdout

dup2(fd[1], STDOUT\_FILENO);

// Close write end of pipe

close(fd[1]);

// Execute 'ls -l' command

execlp("ls", "ls", "-l", NULL);

// If execlp returns, an error has occurred

perror("execlp");

exit(EXIT\_FAILURE);

} else { // Parent process

// Close write end of pipe

close(fd[1]);

// Duplicate read end of pipe to stdin

dup2(fd[0], STDIN\_FILENO);

// Close read end of pipe

close(fd[0]);

// Execute 'wc -l' command

execlp("wc", "wc", "-l", NULL);

// If execlp returns, an error has occurred

perror("execlp");

exit(EXIT\_FAILURE);

}

return 0;

}

**23. Write a C Program that demonstrates redirection of standard output to a file.**

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <unistd.h>

int main() {

int file\_descriptor = open("output.txt", O\_CREAT | O\_WRONLY, 0644);

if (file\_descriptor < 0) {

perror("Failed to open output file");

exit(1);

}

// Redirect stdout to the output file

dup2(file\_descriptor, STDOUT\_FILENO);

// Print some output

printf("This is a test of output redirection.\n");

// Close the file descriptor

close(file\_descriptor);

return 0;

}

**24. Write a C program that illustrates how to execute two commands concurrently with a pipe.**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main() {

int fd[2];

pid\_t pid;

if (pipe(fd) == -1) {

perror("pipe");

exit(EXIT\_FAILURE);

}

pid = fork();

if (pid < 0) {

perror("fork");

exit(EXIT\_FAILURE);

} else if (pid == 0) { // Child process

close(fd[0]);

dup2(fd[1], STDOUT\_FILENO);

execlp("ls", "ls", NULL);

} else { // Parent process

close(fd[1]);

dup2(fd[0], STDIN\_FILENO);

execlp("wc", "wc", "-l", NULL);

}

Return 0; }

**25. Write a C program that illustrates suspending and resuming processes using signals.**

#include <stdio.h>

#include <stdlib.h>

#include <signal.h>

#include <unistd.h>

pid\_t child\_pid;

void sigint\_handler(int sig) {

if (child\_pid > 0) {

printf("Child process is being suspended...\n");

kill(child\_pid, SIGSTOP);

}

}

void sigtstp\_handler(int sig) {

if (child\_pid > 0) {

printf("Child process is being resumed...\n");

kill(child\_pid, SIGCONT);

}

}

int main() {

signal(SIGINT, sigint\_handler);

signal(SIGTSTP, sigtstp\_handler);

printf("Starting child process...\n");

child\_pid = fork();

if (child\_pid == 0) {

// Child process

printf("Child process started. PID: %d\n", getpid());

for (int i = 1; i <= 10; i++) {

printf("Child process: %d\n", i);

sleep(1);

}

printf("Child process finished.\n");

exit(0);

} else if (child\_pid > 0) {

// Parent process

while (1) {

printf("Parent process running...\n");

sleep(1);

}

} else {

// Fork error

printf("Error creating child process.\n");

exit(1);

}

return 0;

}

**26. Write a C program that illustrates inters process communication using shared memory**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <string.h>

#define SHM\_SIZE 1024

int main()

{

key\_t key = 1234;

int shmid;

char \*shm\_ptr;

// Create a shared memory segment

shmid = shmget(key, SHM\_SIZE, IPC\_CREAT | 0666);

if (shmid == -1) {

perror("shmget");

exit(1);

}

// Attach the shared memory segment to our process's address space

shm\_ptr = shmat(shmid, NULL, 0);

if (shm\_ptr == (char \*) -1) {

perror("shmat");

exit(1);

}

// Write some data to the shared memory segment

strcpy(shm\_ptr, "Hello from the parent process!");

// Fork a child process

pid\_t pid = fork();

if (pid == -1) {

perror("fork");

exit(1);

}

if (pid == 0) {

// Child process

// Attach the shared memory segment to the child process's address space

shm\_ptr = shmat(shmid, NULL, 0);

if (shm\_ptr == (char \*) -1) {

perror("shmat");

exit(1);

}

// Read the data from the shared memory segment

printf("Message from parent: %s\n", shm\_ptr);

// Detach the shared memory segment from the child process's address space

if (shmdt(shm\_ptr) == -1) {

perror("shmdt");

exit(1);

}

exit(0);

} else {

// Parent process

// Wait for the child process to finish

wait(NULL);

// Detach the shared memory segment from the parent process's address space

if (shmdt(shm\_ptr) == -1) {

perror("shmdt");

exit(1);

}

// Remove the shared memory segment

if (shmctl(shmid, IPC\_RMID, NULL) == -1) {

perror("shmctl");

exit(1);

}

}

return 0;

}