1. **Write a program to create a named pipe where child sends message to parent.**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/stat.h>

#include <string.h>

#define FIFO\_NAME "/tmp/myfifo"

int main() {

mkfifo(FIFO\_NAME, 0666);

if (fork() == 0) {

int fd = open(FIFO\_NAME, O\_WRONLY);

write(fd, "Hello, parent!", 15);

close(fd);

exit(EXIT\_SUCCESS);

} else {

int fd = open(FIFO\_NAME, O\_RDONLY);

char buf[100];

read(fd, buf, sizeof(buf));

printf("Received message from child: %s\n", buf);

close(fd);

unlink(FIFO\_NAME);

exit(EXIT\_SUCCESS);

}

return 0;

}

--------------------------------------------------------------------------------------------------------------------

1. **Write a program in C to print the file names of a specified directory.**

#include <stdio.h>

#include <dirent.h>

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

if (argc != 2) {

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

return 1;

}

DIR \*dir = opendir(argv[1]);

if (dir == NULL) {

perror("opendir");

return 1;

}

struct dirent \*entry;

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

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

}

closedir(dir);

return 0;

}

1. **Write a program to catch SIGINT signal five times and print message ‘SIGINT signal occurred’ every time and exit at sixth occurrence. Also ignore every occurrence of SIGTSTP signal.**

#include <stdio.h>

#include <stdlib.h>

#include <signal.h>

volatile sig\_atomic\_t sigint\_count = 0;

void sigint\_handler(int signum) {

sigint\_count++;

printf("SIGINT signal occurred\n");

if (sigint\_count >= 5) {

printf("Exiting after catching SIGINT signal 5 times.\n");

exit(EXIT\_SUCCESS);

}

}

void sigtstp\_handler(int signum) {

printf("SIGTSTP signal occurred but ignored.\n");

}

int main() {

signal(SIGINT, sigint\_handler);

signal(SIGTSTP, sigtstp\_handler);

printf("Press Ctrl+C (SIGINT) 5 times to exit.\n");

while (1) {

// Wait for signals

}

return 0;

}

---------------------------------------------------------------------------------------------------------------------

1. **WriteaprogramtoblockSIGINTsignalfor5seconds.**

#include <stdio.h>

#include <stdlib.h>

#include <signal.h>

#include <unistd.h>

int main() {

sigset\_t set;

sigemptyset(&set);

sigaddset(&set, SIGINT);

// Block SIGINT

sigprocmask(SIG\_BLOCK, &set, NULL);

printf("SIGINT signal blocked for 5 seconds...\n");

// Sleep for 5 seconds

sleep(5);

// Unblock SIGINT

sigprocmask(SIG\_UNBLOCK, &set, NULL);

printf("SIGINT signal unblocked.\n");

return 0;

}

---------------------------------------------------------------------------------------------------------------------

1. **Write a program in LINUX to simulate extended shell. Show the prompt and accept standard shell command, which will be executed by child process using one of the exec family system calls. Parent process waits until child finishes execution The command may consist of at the most 5 parameters . The process should be repeated till user types “exit”.**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/wait.h>

#define MAX\_PARAMS 5

#define MAX\_COMMAND\_LEN 100

int main() {

char command[MAX\_COMMAND\_LEN];

char \*args[MAX\_PARAMS + 2]; // +1 for command, +1 for NULL terminator

while (1) {

printf("Extended Shell > ");

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

if (strcmp(command, "exit\n") == 0) {

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

break;

}

int num\_params = 0;

args[num\_params++] = strtok(command, " \n");

while ((args[num\_params++] = strtok(NULL, " \n")) != NULL && num\_params < MAX\_PARAMS + 1);

pid\_t child\_pid = fork();

if (child\_pid < 0) {

perror("fork");

exit(EXIT\_FAILURE);

} else if (child\_pid == 0) {

execvp(args[0], args);

perror("execvp");

exit(EXIT\_FAILURE);

}

else {

int status;

waitpid(child\_pid, &status, 0);

if (WIFEXITED(status)) {

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

}

}

}

return 0;

}

1. **Write a program to count the no. of files of a specified directory.**

#include <stdio.h>

#include <stdlib.h>

#include <dirent.h>

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

if (argc != 2) {

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

return 1;

}

DIR \*dir = opendir(argv[1]);

if (dir == NULL) {

perror("opendir");

return 1;

}

int file\_count = 0;

struct dirent \*entry;

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

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

file\_count++;

}

}

closedir(dir);

printf("Number of files in directory '%s': %d\n", argv[1], file\_count);

return 0;

}

1. **Use ofdupanddup2systemcall. Create one file namedtemp.txt. Write some contents in the file. Make use of dup and dup2 system calls to write following contents in the file. Make use of four descriptors and channel zero. ‘M.Sc. Computer Science SemesterII Class Advanced Operating Systems Practical Examination’**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

int main() {

int fd, fd2, fd3, fd4;

// Open the file

if ((fd = open("temp.txt", O\_WRONLY | O\_CREAT | O\_TRUNC, 0666)) == -1) {

perror("open");

exit(EXIT\_FAILURE);

}

// Duplicate file descriptor using dup

fd2 = dup(fd);

fd3 = dup(fd);

fd4 = dup(fd);

// Write to the file using different descriptors

dprintf(fd, "M.Sc. Computer Science Semester");

dprintf(fd2, "II Class Advanced Operating Systems");

dprintf(fd3, "Practical Examination");

dprintf(fd4, "\n");

// Close the file descriptors

close(fd);

close(fd2);

close(fd3);

close(fd4);

printf("Contents written to file successfully.\n");

return 0;

}

1. **Check whether the specified file is regular file or directory and print user permissions of the file.**

#include <stdio.h>

#include <stdlib.h>

#include <sys/stat.h>

#include <sys/types.h>

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

if (argc != 2) {

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

return 1;

}

struct stat st;

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

perror("stat");

return 1;

}

printf("%s is a %s.\n", argv[1], S\_ISREG(st.st\_mode) ? "regular file" : S\_ISDIR(st.st\_mode) ? "directory" : "neither");

printf("Permissions: %c%c%c\n", (st.st\_mode & S\_IRUSR) ? 'r' : '-', (st.st\_mode & S\_IWUSR) ? 'w' : '-', (st.st\_mode & S\_IXUSR) ? 'x' : '-');

return 0;

}

1. **Write a program in Linux to block SIGQUIT signal for 5 seconds.**

#include <stdio.h>

#include <stdlib.h>

#include <signal.h>

#include <unistd.h>

void sigquit\_handler(int signum) {

printf("SIGQUIT signal received.\n");

}

int main() {

// Register signal handler for SIGQUIT

signal(SIGQUIT, sigquit\_handler);

printf("SIGQUIT signal blocked for 5 seconds...\n");

// Block SIGQUIT

sigset\_t set, oldset;

sigemptyset(&set);

sigaddset(&set, SIGQUIT);

sigprocmask(SIG\_BLOCK, &set, &oldset);

// Sleep for 5 seconds

sleep(5);

// Unblock SIGQUIT

sigprocmask(SIG\_SETMASK, &oldset, NULL);

printf("SIGQUIT signal unblocked.\n");

return 0;

}

1. **Write a program to create a named pipe where parent sends message to child.**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <string.h>

#define FIFO\_NAME "/tmp/myfifo"

#define BUFFER\_SIZE 100

int main() {

mkfifo(FIFO\_NAME, 0666);

pid\_t child\_pid = fork();

if (child\_pid < 0) {

perror("Fork failed");

exit(EXIT\_FAILURE);

}

int fd = open(FIFO\_NAME, O\_WRONLY | O\_CREAT | O\_TRUNC);

if (child\_pid == 0) {

// Child process

dup2(fd, STDOUT\_FILENO);

execlp("cat", "cat", NULL);

perror("exec failed");

exit(EXIT\_FAILURE);

} else {

// Parent process

char message[BUFFER\_SIZE] = "Hello, child!\n";

write(fd, message, strlen(message));

close(fd);

wait(NULL);

unlink(FIFO\_NAME);

exit(EXIT\_SUCCESS);

}

}

1. **Write program to handle SIGINT, SIGSEGV, SIGQUIT and SIFTSTP signals.**

#include <stdio.h>

#include <stdlib.h>

#include <signal.h>

void sig\_handler(int signum) {

switch(signum) {

case SIGINT:

printf("Caught SIGINT signal\n");

break;

case SIGSEGV:

printf("Caught SIGSEGV signal\n");

break;

case SIGQUIT:

printf("Caught SIGQUIT signal\n");

break;

case SIGTSTP:

printf("Caught SIGTSTP signal\n");

break;

}

}

int main() {

signal(SIGINT, sig\_handler);

signal(SIGSEGV, sig\_handler);

signal(SIGQUIT, sig\_handler);

signal(SIGTSTP, sig\_handler);

printf("Signal handlers registered. Waiting for signals...\n");

while (1) {

// This loop allows the program to continue running and handle signals

}

return 0;

}

1. **Write program to handle SIGINT, SIGALRM and SIFTSTP signals.**

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

void sig\_handler(int signum) {

switch(signum) {

case SIGINT:

printf("Caught SIGINT signal\n");

break;

case SIGALRM:

printf("Caught SIGALRM signal\n");

break;

case SIGTSTP:

printf("Caught SIGTSTP signal\n");

break;

}

}

int main() {

signal(SIGINT, sig\_handler);

signal(SIGALRM, sig\_handler);

signal(SIGTSTP, sig\_handler);

printf("Signal handlers registered. Waiting for signals...\n");

alarm(5);

while (1) {

// This loop allows the program to continue running and handle signals

}

return 0;

}

1. **Write a program to create hole in it.(Use L seek system call) .Write “Hello World !”five times after every 100 blank characters.**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

#include <string.h>

#define FILENAME "holed\_file.txt"

#define BUFFER\_SIZE 100

int main() {

// Create or truncate the file

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

if (fd == -1) {

perror("open");

exit(EXIT\_FAILURE);

}

// Write blank characters to create a hole

lseek(fd, 500, SEEK\_SET); // Move the file offset 500 bytes from the beginning

write(fd, "", 1); // Write one byte to create a hole

// Write "Hello World !" five times after every 100 blank characters

char message[] = "Hello World !\n";

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

lseek(fd, 100, SEEK\_CUR); // Move the file offset 100 bytes from the current position

write(fd, message, strlen(message)); // Write "Hello World !"

}

// Close the file

close(fd);

printf("File with hole created: %s\n", FILENAME);

return 0;

}

1. **Write program that illustrates suspending and resuming processes using signals.**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <signal.h>

volatile sig\_atomic\_t suspend\_flag = 0;

void sig\_handler(int signum) {

if (signum == SIGTSTP) {

suspend\_flag = !suspend\_flag;

printf(suspend\_flag ? "Process suspended. Press Enter to resume...\n" : "Resuming process...\n");

} else {

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

exit(EXIT\_SUCCESS);

}

}

int main() {

signal(SIGINT, sig\_handler);

signal(SIGTSTP, sig\_handler);

printf("Running. Press Ctrl+Z to suspend and Ctrl+C to exit.\n");

while (1) {

if (suspend\_flag) getchar();

sleep(1); // Simulate some work

}

return 0;

}

1. **Write a program to create an unnamed pipe where child sends message to parent.**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main() {

int pipefd[2];

char buffer[100];

pipe(pipefd);

if (fork() == 0) {

close(pipefd[0]);

write(pipefd[1], "Hello, parent!", 14);

close(pipefd[1]);

} else {

close(pipefd[1]);

read(pipefd[0], buffer, 100);

printf("Message from child: %s\n", buffer);

close(pipefd[0]);

}

return 0;

}

1. **Print the type of file and group permissions of it where file name is accepted thru command line arguments.**

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/stat.h>

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

if (argc != 2) {

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

return 1;

}

struct stat st;

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

perror("stat");

return 1;

}

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

printf("Type: Regular File\n");

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

printf("Type: Directory\n");

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

printf("Type: Symbolic Link\n");

} else {

printf("Type: Other\n");

}

printf("Group Permissions: %c%c%c\n",

(st.st\_mode & S\_IRGRP) ? 'r' : '-',

(st.st\_mode & S\_IWGRP) ? 'w' : '-',

(st.st\_mode & S\_IXGRP) ? 'x' : '-');

return 0;

}

---------------------------------------------------------------------------------------------------------------------17) **Write a program to catch death of child signal by parent process after 5 seconds. Use alarm system call**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <signal.h>

void child\_handler(int signum) {

printf("Child process has died.\n");

exit(EXIT\_SUCCESS);

}

int main() {

pid\_t child\_pid;

signal(SIGCHLD, child\_handler);

child\_pid = fork();

if (child\_pid < 0) {

perror("Fork failed");

exit(EXIT\_FAILURE);

}

if (child\_pid == 0) {

sleep(2);

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

exit(EXIT\_SUCCESS);

}

printf("Parent process waiting for child to die.\n");

alarm(5);

pause();

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

exit(EXIT\_SUCCESS);

}

1. **Write a program where parent and child share file access.**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <fcntl.h>

int main() {

int fd;

fd = open("shared\_file.txt", O\_WRONLY | O\_CREAT | O\_TRUNC, 0666);

if (fd == -1) {

perror("open");

exit(EXIT\_FAILURE);

}

if (fork() == 0) {

printf("Child process writing to file...\n");

dprintf(fd, "Hello from child!\n");

} else {

printf("Parent process writing to file...\n");

dprintf(fd, "Hello from parent!\n");

wait(NULL);

close(fd);

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

}

return 0;

}

1. **Write the program that accepts directory name as input and print its contents**

#include <stdio.h>

#include <stdlib.h>

#include <dirent.h>

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

if (argc != 2) {

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

return 1;

}

DIR \*dir = opendir(argv[1]);

if (!dir) {

perror("opendir");

return 1;

}

printf("Contents of directory '%s':\n", argv[1]);

struct dirent \*entry;

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

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

}

closedir(dir);

return 0;

}

1. **Write a program to create an unnamed pipe where parent sends message to child.**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#define BUFFER\_SIZE 100

int main() {

int pipefd[2];

char buffer[BUFFER\_SIZE];

pipe(pipefd);

if (fork() == 0) {

close(pipefd[1]);

read(pipefd[0], buffer, BUFFER\_SIZE);

printf("Child received message from parent: %s\n", buffer);

close(pipefd[0]);

} else {

close(pipefd[0]);

write(pipefd[1], "Hello, child!", 14);

close(pipefd[1]);

wait(NULL);

}

return 0;

}

---------------------------------------------------------------------------------------------------------------------21) **Write a program to implement following commands in linux. Typelines+5<filename>print first 5linesof a file. Type lines 8<filename>print last 20linesof a file.**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_LINE\_LENGTH 256

#define MAX\_LINES 20

void print\_lines(char \*filename, int start, int num\_lines) {

FILE \*file = fopen(filename, "r");

if (file == NULL) {

perror("fopen");

exit(EXIT\_FAILURE);

}

char lines[MAX\_LINES][MAX\_LINE\_LENGTH];

int line\_count = 0;

while (fgets(lines[line\_count % MAX\_LINES], MAX\_LINE\_LENGTH, file) != NULL) {

line\_count++;

}

start = line\_count > MAX\_LINES ? (line\_count % MAX\_LINES) : 0;

for (int i = 0; i < num\_lines && i < line\_count; i++) {

printf("%s", lines[(start + i) % MAX\_LINES]);

}

fclose(file);

}

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

if (argc != 3) {

fprintf(stderr, "Usage: %s <lines+5 or lines8> <filename>\n", argv[0]);

return 1;

}

int start = 0, num\_lines;

sscanf(argv[1], "lines%\*c%d", &num\_lines);

if (strncmp(argv[1], "lines+", 6) == 0) {

start = 5;

}

print\_lines(argv[2], start, num\_lines);

return 0;

}

22) **Write a program to count the no.of files of a directory.**

#include <stdio.h>

#include <stdlib.h>

#include <dirent.h>

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

if (argc != 2) {

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

return 1;

}

DIR \*dir = opendir(argv[1]);

if (!dir) {

perror("opendir");

return 1;

}

int file\_count = 0;

struct dirent \*entry;

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

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

file\_count++;

}

}

closedir(dir);

printf("Number of files in directory '%s': %d\n", argv[1], file\_count);

return 0;

}

23) **Write a program to catch SIGQUIT signal five times and print message ‘SIGQUIT signal occurred’ every time and exit at sixth occurrence. Also ignore every occurrence of SIGTSTP signal.**

#include <stdio.h>

#include <stdlib.h>

#include <signal.h>

volatile sig\_atomic\_t quit\_count = 0;

void sigquit\_handler(int signum) {

if (signum == SIGQUIT) {

printf("SIGQUIT signal occurred %d time(s).\n", ++quit\_count);

if (quit\_count >= 5) {

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

exit(EXIT\_SUCCESS);

}

}

}

void sigtstp\_handler(int signum) {

// Ignore SIGTSTP signals

}

int main() {

signal(SIGQUIT, sigquit\_handler);

signal(SIGTSTP, sigtstp\_handler);

printf("Waiting for SIGQUIT signal...\n");

while (1) {

pause();

}

return 0;

}

24) **Write a program in LINUX to simulate extended shell. Show the prompt and accept standard shell command, which will be executed by child process using one of the exec family system calls. Parent process waits until child finishes execution The comm. And may consist of at the most 5 parameters . The process should be repeated till user types “exit”.**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/wait.h>

#define MAX\_COMMAND\_LENGTH 256

#define MAX\_PARAMETERS 5

int main() {

char command[MAX\_COMMAND\_LENGTH];

char \*parameters[MAX\_PARAMETERS + 1]; // Additional slot for NULL

while (1) {

printf("Shell> ");

if (!fgets(command, MAX\_COMMAND\_LENGTH, stdin)) break;

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

if (strcmp(command, "exit") == 0) break;

int pid = fork();

if (pid == -1) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (pid == 0) {

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

int i = 0;

while (token != NULL && i < MAX\_PARAMETERS) {

parameters[i++] = token;

token = strtok(NULL, " ");

}

parameters[i] = NULL;

execvp(parameters[0], parameters);

perror("execvp");

exit(EXIT\_FAILURE);

} else {

wait(NULL);

}

}

return 0;

}

25) **Print the type of file where file name is accepted thru command line arguments.**

#include <stdio.h>

#include <sys/stat.h>

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

if (argc != 2) {

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

return 1;

}

struct stat st;

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

perror("stat");

return 1;

}

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

if (S\_ISREG(st.st\_mode)) printf("regular file.\n");

else if (S\_ISDIR(st.st\_mode)) printf("directory.\n");

else if (S\_ISLNK(st.st\_mode)) printf("symbolic link.\n");

else printf("file of unknown type.\n");

return 0;

}

--------------------------------------------------------------------------------------------------------------------------