S1

Q1)Take multiple files as Command Line Arguments and print their inode numbers and file types

Ans->

import os

import sys

def get\_file\_info(file\_path):

try:

# Get the inode number and file type

inode\_number = os.stat(file\_path).st\_ino

file\_type = 'Directory' if os.path.isdir(file\_path) else 'File'

return inode\_number, file\_type

except FileNotFoundError:

return None, None

def main():

# Check if there are command line arguments

if len(sys.argv) < 2:

print("Usage: python script.py file1 file2 file3 ...")

sys.exit(1)

# Process each command line argument

for file\_path in sys.argv[1:]:

inode\_number, file\_type = get\_file\_info(file\_path)

if inode\_number is not None and file\_type is not None:

print(f"File: {file\_path}\nInode Number: {inode\_number}\nFile Type: {file\_type}\n")

if \_\_name\_\_ == "\_\_main\_\_":

main()

Q2 Write a C program to send SIGALRM signal by child process to parent process and parent process make a provision to catch the signal and display alarm is fired.(Use Kill, fork, signal and sleep system call)

Ans ->

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <signal.h>

#include <sys/types.h>

void alarm\_handler(int signum) {

if (signum == SIGALRM) {

printf("Alarm is fired!\n");

}

}

int main() {

pid\_t child\_pid;

// Set up the signal handler for SIGALRM

signal(SIGALRM, alarm\_handler);

// Create a child process

if ((child\_pid = fork()) < 0) {

perror("Fork failed");

exit(EXIT\_FAILURE);

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

// Child process

sleep(2); // Sleep for 2 seconds

// Send SIGALRM signal to the parent process

kill(getppid(), SIGALRM);

exit(EXIT\_SUCCESS);

} else {

// Parent process

printf("Waiting for the alarm...\n");

// Sleep to allow the child process to send the signal

sleep(5);

// The parent process will continue here after receiving the signal

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

}

return 0;

}

S2

Q1) Write a C program to find file properties such as inode number, number of hard link, File permissions, File size, File access and modification time and so on of a given file using stat() system call

Ans ->

#include <stdio.h>

#include <sys/stat.h>

#include <unistd.h>

#include <stdlib.h>

#include <sys/types.h>

#include <pwd.h>

#include <grp.h>

#include <time.h>

void printFileProperties(const char \*filename);

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

if (argc != 2) {

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

exit(EXIT\_FAILURE);

}

const char \*filename = argv[1];

printFileProperties(filename);

return 0;

}

void printFileProperties(const char \*filename) {

struct stat fileInfo;

if (stat(filename, &fileInfo) == -1) {

perror("stat");

exit(EXIT\_FAILURE);

}

printf("File Information for: %s\n", filename);

printf("=======================================\n");

printf("Inode Number: %lu\n", (unsigned long)fileInfo.st\_ino);

printf("Number of Hard Links: %lu\n", (unsigned long)fileInfo.st\_nlink);

printf("File Size: %ld bytes\n", (long)fileInfo.st\_size);

printf("File Permissions: ");

printf((S\_ISDIR(fileInfo.st\_mode)) ? "d" : "-");

printf((fileInfo.st\_mode & S\_IRUSR) ? "r" : "-");

printf((fileInfo.st\_mode & S\_IWUSR) ? "w" : "-");

printf((fileInfo.st\_mode & S\_IXUSR) ? "x" : "-");

printf((fileInfo.st\_mode & S\_IRGRP) ? "r" : "-");

printf((fileInfo.st\_mode & S\_IWGRP) ? "w" : "-");

printf((fileInfo.st\_mode & S\_IXGRP) ? "x" : "-");

printf((fileInfo.st\_mode & S\_IROTH) ? "r" : "-");

printf((fileInfo.st\_mode & S\_IWOTH) ? "w" : "-");

printf((fileInfo.st\_mode & S\_IXOTH) ? "x" : "-");

printf("\n");

printf("Owner UID: %d (%s)\n", (int)fileInfo.st\_uid, getpwuid(fileInfo.st\_uid)->pw\_name);

printf("Group GID: %d (%s)\n", (int)fileInfo.st\_gid, getgrgid(fileInfo.st\_gid)->gr\_name);

// Convert time to a string representation

char accessTime[20];

strftime(accessTime, sizeof(accessTime), "%Y-%m-%d %H:%M:%S", localtime(&fileInfo.st\_atime));

char modifyTime[20];

strftime(modifyTime, sizeof(modifyTime), "%Y-%m-%d %H:%M:%S", localtime(&fileInfo.st\_mtime));

printf("Last Access Time: %s\n", accessTime);

printf("Last Modification Time: %s\n", modifyTime);

}

Q2) Write a C program that catches the ctrl-c (SIGINT) signal for the first time and display the appropriate message and exits on pressing ctrl-c again.

Ans ->

#include <stdio.h>

#include <signal.h>

#include <stdbool.h>

// Global variable to track whether SIGINT has been caught once

bool sigintCaught = false;

// Signal handler function for SIGINT

void sigintHandler(int signo) {

if (!sigintCaught) {

printf("\nCtrl-C (SIGINT) caught. Press again to exit.\n");

sigintCaught = true;

} else {

printf("\nCtrl-C pressed again. Exiting.\n");

exit(0);

}

}

int main() {

// Set up the signal handler for SIGINT

if (signal(SIGINT, sigintHandler) == SIG\_ERR) {

perror("Unable to set up signal handler");

return 1;

}

// Infinite loop to keep the program running

while (1) {

// Do some work or just sleep to keep the program alive

// Replace this with your actual program logic

sleep(1);

}

return 0;

}

S3

Q1) Print the type of file and inode number where file name accepted through Command Lin

Ans ->

#include <stdio.h>

#include <stdlib.h>

#include <sys/stat.h>

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

// Check if a filename is provided as a command line argument

if (argc != 2) {

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

return 1;

}

const char \*filename = argv[1];

struct stat fileStat;

// Use stat to retrieve file information

if (stat(filename, &fileStat) == -1) {

perror("Error getting file information");

return 1;

}

// Print file type and inode number

printf("File Type: ");

switch (fileStat.st\_mode & S\_IFMT) {

case S\_IFREG:

printf("Regular File\n");

break;

case S\_IFDIR:

printf("Directory\n");

break;

case S\_IFLNK:

printf("Symbolic Link\n");

break;

case S\_IFIFO:

printf("FIFO/pipe\n");

break;

case S\_IFCHR:

printf("Character Device\n");

break;

case S\_IFBLK:

printf("Block Device\n");

break;

case S\_IFSOCK:

printf("Socket\n");

break;

default:

printf("Unknown\n");

}

printf("Inode Number: %lu\n", (unsigned long)fileStat.st\_ino);

return 0;

}

Q2) Write a C program which creates a child process to run linux/ unix command or any user defined program. The parent process set the signal handler for death of child signal and Alarm signal. If a child process does not complete its execution in 5 second then parent process kills child process.

Ans

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <signal.h>

#include <sys/wait.h>

// Global variable to track whether the child process has terminated

volatile sig\_atomic\_t childTerminated = 0;

// Signal handler for SIGCHLD

void sigchldHandler(int signo) {

(void)signo; // Unused parameter

childTerminated = 1;

}

// Signal handler for SIGALRM

void sigalrmHandler(int signo) {

(void)signo; // Unused parameter

fprintf(stderr, "Child process took too long. Killing it.\n");

exit(EXIT\_FAILURE);

}

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

// Check if a command is provided as a command line argument

if (argc < 2) {

fprintf(stderr, "Usage: %s <command> [arg1] [arg2] ...\n", argv[0]);

return EXIT\_FAILURE;

}

// Set up the signal handler for SIGCHLD

if (signal(SIGCHLD, sigchldHandler) == SIG\_ERR) {

perror("Unable to set up signal handler for SIGCHLD");

return EXIT\_FAILURE;

}

// Set up the signal handler for SIGALRM

if (signal(SIGALRM, sigalrmHandler) == SIG\_ERR) {

perror("Unable to set up signal handler for SIGALRM");

return EXIT\_FAILURE;

}

pid\_t childPid = fork();

if (childPid == -1) {

perror("Failed to fork");

return EXIT\_FAILURE;

}

if (childPid == 0) {

// Child process

execvp(argv[1], &argv[1]);

// If execvp fails

perror("execvp");

exit(EXIT\_FAILURE);

} else {

// Parent process

// Set an alarm for 5 seconds

alarm(5);

// Wait for the child process to terminate

while (!childTerminated) {

// Sleep for a short duration to avoid busy waiting

usleep(100000); // 100 milliseconds

}

// Cancel the alarm

alarm(0);

// Wait for the child process to get its exit status

int status;

waitpid(childPid, &status, 0);

if (WIFEXITED(status)) {

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

} else {

printf("Child process did not exit normally\n");

}

}

return EXIT\_SUCCESS;

}

S4

Q1) Write a C program to find whether a given files passed through command line arguments are present in current directory or not.

Ans

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

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

// Check if file names are provided as command line arguments

if (argc < 2) {

fprintf(stderr, "Usage: %s <file1> [file2] [file3] ...\n", argv[0]);

return EXIT\_FAILURE;

}

// Iterate through the command line arguments starting from index 1

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

// Check if the file exists in the current directory

if (access(argv[i], F\_OK) != -1) {

printf("%s: File exists in the current directory.\n", argv[i]);

} else {

printf("%s: File does not exist in the current directory.\n", argv[i]);

}

}

return EXIT\_SUCCESS;

}

Q2) Write a C program which creates a child process and child process catches a signal SIGHUP, SIGINT and SIGQUIT. The Parent process send a SIGHUP or SIGINT signal after every 3 seconds, at the end of 15 second parent send SIGQUIT signal to child and child terminates by displaying message "My Papa has Killed me!!!”.

ANS

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <signal.h>

#include <sys/types.h>

// Signal handler function for SIGHUP, SIGINT, and SIGQUIT

void signalHandler(int signo) {

if (signo == SIGHUP) {

printf("Child: Received SIGHUP signal from Papa.\n");

} else if (signo == SIGINT) {

printf("Child: Received SIGINT signal from Papa.\n");

} else if (signo == SIGQUIT) {

printf("Child: Received SIGQUIT signal from Papa. My Papa has Killed me!!!\n");

exit(EXIT\_SUCCESS);

}

}

int main() {

pid\_t childPid = fork();

if (childPid == -1) {

perror("Failed to fork");

return EXIT\_FAILURE;

}

if (childPid == 0) {

// Child process

// Set up signal handlers for SIGHUP, SIGINT, and SIGQUIT

signal(SIGHUP, signalHandler);

signal(SIGINT, signalHandler);

signal(SIGQUIT, signalHandler);

// Infinite loop to keep the child process running

while (1) {

// Do some work or just sleep to keep the child process alive

// Replace this with your actual program logic

sleep(1);

}

} else {

// Parent process

// Sleep for 3 seconds and send SIGHUP signal to the child

sleep(3);

kill(childPid, SIGHUP);

// Sleep for another 3 seconds and send SIGINT signal to the child

sleep(3);

kill(childPid, SIGINT);

// Sleep for another 3 seconds and send SIGHUP signal to the child

sleep(3);

kill(childPid, SIGHUP);

// Sleep for another 6 seconds (total of 15 seconds) and send SIGQUIT signal to the child

sleep(6);

kill(childPid, SIGQUIT);

// Wait for the child process to terminate

wait(NULL);

}

return EXIT\_SUCCESS;

}

S5

Q1) Read the current directory and display the name of the files, no of files in current directory

ANS

#include <stdio.h>

#include <stdlib.h>

#include <dirent.h>

int main() {

DIR \*dir;

struct dirent \*entry;

// Open the current directory

dir = opendir(".");

if (dir == NULL) {

perror("Unable to open directory");

return EXIT\_FAILURE;

}

// Initialize file count

int fileCount = 0;

// Read directory entries

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

// Skip entries for current directory (.) and parent directory (..)

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

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

fileCount++;

}

}

// Close the directory

closedir(dir);

// Print the total number of files

printf("\nTotal number of files: %d\n", fileCount);

return EXIT\_SUCCESS;

}

Q2) Write a C program to create an unnamed pipe. The child process will write following three messages to pipe and parent process display it. Message1 = “Hello World” Message2 = “Hello SPPU” Message3 = “Linux is Funny”

ANS

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#define MESSAGE1 "Hello World"

#define MESSAGE2 "Hello SPPU"

#define MESSAGE3 "Linux is Funny"

int main() {

int pipefd[2];

pid\_t childPid;

// Create a pipe

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

perror("Failed to create pipe");

return EXIT\_FAILURE;

}

// Fork a child process

childPid = fork();

if (childPid == -1) {

perror("Failed to fork");

return EXIT\_FAILURE;

}

if (childPid == 0) {

// Child process

// Close the read end of the pipe

close(pipefd[0]);

// Write messages to the pipe

write(pipefd[1], MESSAGE1, strlen(MESSAGE1) + 1);

write(pipefd[1], MESSAGE2, strlen(MESSAGE2) + 1);

write(pipefd[1], MESSAGE3, strlen(MESSAGE3) + 1);

// Close the write end of the pipe

close(pipefd[1]);

exit(EXIT\_SUCCESS);

} else {

// Parent process

// Close the write end of the pipe

close(pipefd[1]);

// Read and display messages from the pipe

char buffer[256];

while (read(pipefd[0], buffer, sizeof(buffer)) > 0) {

printf("Parent: Received message: %s\n", buffer);

}

// Close the read end of the pipe

close(pipefd[0]);

// Wait for the child process to terminate

wait(NULL);

}

return EXIT\_SUCCESS;

}

S6

Q1) ) Display all the files from current directory which are created in particular month

ANS

find . -type f -newermt "$(date -d '2022-12-01' +%Y-%m-%d)" ! -newermt "$(date -d '2023-01-01' +%Y-%m-%d)" -exec ls -l {} \;

(SLIP-7)

**SUBJECT: CS-504-MJP: Lab Course on CS-501-MJ (Advanced Operating System)**

Q1)Write a C Program that demonstrates redirection of standard output to a file

#include <stdio.h>

int main() {

// Open a file for writing

FILE \*file = freopen("output.txt", "w", stdout);

if (file == NULL) {

perror("Error opening file");

return 1;

}

// Print to the file instead of the console

printf("This text will be written to output.txt\n");

// Close the file (optional, as fclose(stdout) will be called when the program exits)

fclose(file);

return 0;

}

Q2) 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>

int main() {

int pipefd[2];

pid\_t childPid;

// Create a pipe

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

perror("Failed to create pipe");

exit(EXIT\_FAILURE);

}

// Fork a child process

childPid = fork();

if (childPid == -1) {

perror("Failed to fork");

exit(EXIT\_FAILURE);

}

if (childPid == 0) {

// Child process

// Close the write end of the pipe

close(pipefd[1]);

// Redirect the standard input to read from the pipe

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

// Close the remaining file descriptor

close(pipefd[0]);

// Execute wc -l using exec

execlp("wc", "wc", "-l", (char \*)NULL);

// If exec fails

perror("execlp");

exit(EXIT\_FAILURE);

} else {

// Parent process

// Close the read end of the pipe

close(pipefd[0]);

// Redirect the standard output to write to the pipe

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

// Close the remaining file descriptor

close(pipefd[1]);

// Execute ls -l using exec

execlp("ls", "ls", "-l", (char \*)NULL);

// If exec fails

perror("execlp");

exit(EXIT\_FAILURE);

}

return EXIT\_SUCCESS;

}

**(SLIP 8)**

**SUBJECT: CS-504-MJP: Lab Course on CS-501-MJ (Advanced Operating System)**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <unistd.h>**

**#include <fcntl.h>**

**int main() {**

**// Open the file for writing (creating if it doesn't exist, truncating to zero length)**

**int fileDescriptor = open("output.txt", O\_WRONLY | O\_CREAT | O\_TRUNC, 0666);**

**if (fileDescriptor == -1) {**

**perror("Error opening file");**

**exit(EXIT\_FAILURE);**

**}**

**// Duplicate the file descriptor to replace stdout (file descriptor 1)**

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

**perror("Error duplicating file descriptor");**

**close(fileDescriptor);**

**exit(EXIT\_FAILURE);**

**}**

**// Close the original file descriptor**

**close(fileDescriptor);**

**// Now, stdout is redirected to the file "output.txt"**

**// Print to stdout (which is now the file)**

**printf("This text will be written to output.txt\n");**

**// It's good practice to restore stdout to the original state when done**

**// Redirect stdout back to the terminal**

**dup2(STDOUT\_FILENO, 1);**

**// Print to the terminal**

**printf("This text goes back to the terminal.\n");**

**return 0;**

**}**

**Q2) 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 pipefd[2];**

**pid\_t ls\_pid, wc\_pid;**

**// Create a pipe**

**if (pipe(pipefd) == -1) {**

**perror("Failed to create pipe");**

**exit(EXIT\_FAILURE);**

**}**

**// Fork the first child process for ls -l**

**ls\_pid = fork();**

**if (ls\_pid == -1) {**

**perror("Failed to fork");**

**exit(EXIT\_FAILURE);**

**}**

**if (ls\_pid == 0) {**

**// Child process (ls -l)**

**// Close the read end of the pipe**

**close(pipefd[0]);**

**// Redirect standard output to the pipe**

**dup2(pipefd[1], STDOUT\_FILENO);**

**// Close the remaining file descriptor**

**close(pipefd[1]);**

**// Execute ls -l using exec**

**execlp("ls", "ls", "-l", (char \*)NULL);**

**// If exec fails**

**perror("execlp");**

**exit(EXIT\_FAILURE);**

**}**

**// Fork the second child process for wc -l**

**wc\_pid = fork();**

**if (wc\_pid == -1) {**

**perror("Failed to fork");**

**exit(EXIT\_FAILURE);**

**}**

**if (wc\_pid == 0) {**

**// Child process (wc -l)**

**// Close the write end of the pipe**

**close(pipefd[1]);**

**// Redirect standard input to read from the pipe**

**dup2(pipefd[0], STDIN\_FILENO);**

**// Close the remaining file descriptor**

**close(pipefd[0]);**

**// Execute wc -l using exec**

**execlp("wc", "wc", "-l", (char \*)NULL);**

**// If exec fails**

**perror("execlp");**

**exit(EXIT\_FAILURE);**

**}**

**// Close both ends of the pipe in the parent process**

**close(pipefd[0]);**

**close(pipefd[1]);**

**// Wait for both child processes to finish**

**waitpid(ls\_pid, NULL, 0);**

**waitpid(wc\_pid, NULL, 0);**

**return 0;**

**}**

**(SLIP-9)**

**SUBJECT: CS-504-MJP: Lab Course on CS-501-MJ (Advanced Operating System)**

**Q1) Generate parent process to write unnamed pipe and will read from it**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <unistd.h>**

**#include <sys/wait.h>**

**int main() {**

**int pipefd[2]; // 0 for reading, 1 for writing**

**pid\_t child\_pid;**

**// Create a pipe**

**if (pipe(pipefd) == -1) {**

**perror("Failed to create pipe");**

**exit(EXIT\_FAILURE);**

**}**

**// Fork a child process**

**child\_pid = fork();**

**if (child\_pid == -1) {**

**perror("Failed to fork");**

**exit(EXIT\_FAILURE);**

**}**

**if (child\_pid == 0) {**

**// Child process**

**close(pipefd[1]); // Close the write end of the pipe**

**char buffer[1024];**

**ssize\_t bytesRead;**

**// Read from the pipe**

**bytesRead = read(pipefd[0], buffer, sizeof(buffer));**

**// Check if read was successful**

**if (bytesRead > 0) {**

**printf("Child**

**Q2)** **Write a C program to Identify the type (Directory, character device, Block device, Regular file, FIFO or pipe, symbolic link or socket) of given file using stat() system call**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <sys/types.h>**

**#include <sys/stat.h>**

**#include <unistd.h>**

**void identifyFileType(const char \*filename) {**

**struct stat fileInfo;**

**// Use stat to get information about the file**

**if (stat(filename, &fileInfo) == -1) {**

**perror("Error getting file information");**

**exit(EXIT\_FAILURE);**

**}**

**// Check the file type**

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

**printf("%s: Regular File\n", filename);**

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

**printf("%s: Directory\n", filename);**

**} else if (S\_ISCHR(fileInfo.st\_mode)) {**

**printf("%s: Character Device\n", filename);**

**} else if (S\_ISBLK(fileInfo.st\_mode)) {**

**printf("%s: Block Device\n", filename);**

**} else if (S\_ISFIFO(fileInfo.st\_mode)) {**

**printf("%s: FIFO or Pipe\n", filename);**

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

**printf("%s: Symbolic Link\n", filename);**

**} else if (S\_ISSOCK(fileInfo.st\_mode)) {**

**printf("%s: Socket\n", filename);**

**} else {**

**printf("%s: Unknown File Type\n", filename);**

**}**

**}**

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

**if (argc != 2) {**

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

**return EXIT\_FAILURE;**

**}**

**const char \*filename = argv[1];**

**identifyFileType(filename);**

**return EXIT\_SUCCESS;**

**}**

**./file\_type\_identifier some\_file**

**(SLIP-10)**

**SUBJECT: CS-504-MJP: Lab Course on CS-501-MJ (Advanced Operating System)**

**Q1) Write a program that illustrates how to execute two commands concurrently with a pipe.**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <unistd.h>**

**#include <sys/wait.h>**

**int main() {**

**int pipefd[2];**

**pid\_t child1\_pid, child2\_pid;**

**// Create a pipe**

**if (pipe(pipefd) == -1) {**

**perror("Failed to create pipe");**

**exit(EXIT\_FAILURE);**

**}**

**// Fork the first child process**

**child1\_pid = fork();**

**if (child1\_pid == -1) {**

**perror("Failed to fork");**

**exit(EXIT\_FAILURE);**

**}**

**if (child1\_pid == 0) {**

**// Child process 1**

**close(pipefd[0]); // Close the read end of the pipe**

**// Redirect standard output to write to the pipe**

**dup2(pipefd[1], STDOUT\_FILENO);**

**// Close the remaining file descriptor**

**close(pipefd[1]);**

**// Execute the first command (e.g., ls)**

**execlp("ls", "ls", "-l", (char \*)NULL);**

**// If exec fails**

**perror("execlp");**

**exit(EXIT\_FAILURE);**

**}**

**// Fork the second child process**

**child2\_pid = fork();**

**if (child2\_pid == -1) {**

**perror("Failed to fork");**

**exit(EXIT\_FAILURE);**

**}**

**if (child2\_pid == 0) {**

**// Child process 2**

**close(pipefd[1]); // Close the write end of the pipe**

**// Redirect standard input to read from the pipe**

**dup2(pipefd[0], STDIN\_FILENO);**

**// Close the remaining file descriptor**

**close(pipefd[0]);**

**// Execute the second command (e.g., wc -l)**

**execlp("wc", "wc", "-l", (char \*)NULL);**

**// If exec fails**

**perror("execlp");**

**exit(EXIT\_FAILURE);**

**}**

**// Close both ends of the pipe in the parent process**

**close(pipefd[0]);**

**close(pipefd[1]);**

**// Wait for both child processes to finish**

**waitpid(child1\_pid, NULL, 0);**

**waitpid(child2\_pid, NULL, 0);**

**return EXIT\_SUCCESS;**

**}**

**Q2)** **Generate parent process to write unnamed pipe and will write into it. Also generate child process which will read from pipe**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <unistd.h>**

**#define BUFFER\_SIZE 1024**

**int main() {**

**int pipefd[2]; // 0 for reading, 1 for writing**

**pid\_t child\_pid;**

**// Create a pipe**

**if (pipe(pipefd) == -1) {**

**perror("Failed to create pipe");**

**exit(EXIT\_FAILURE);**

**}**

**// Fork a child process**

**child\_pid = fork();**

**if (child\_pid == -1) {**

**perror("Failed to fork");**

**exit(EXIT\_FAILURE);**

**}**

**if (child\_pid == 0) {**

**// Child process**

**close(pipefd[1]); // Close the write end of the pipe**

**char buffer[BUFFER\_SIZE];**

**ssize\_t bytesRead;**

**// Read from the pipe**

**bytesRead = read(pipefd[0], buffer, sizeof(buffer));**

**// Check if read was successful**

**if (bytesRead > 0) {**

**printf("Child received from parent: %.\*s", (int)bytesRead, buffer);**

**} else {**

**perror("Error reading from pipe");**

**}**

**// Close the read end of the pipe**

**close(pipefd[0]);**

**exit(EXIT\_SUCCESS);**

**} else {**

**// Parent process**

**close(pipefd[0]); // Close the read end of the pipe**

**const char \*message = "Hello, Child!"; // Message to be sent to child**

**// Write to the pipe**

**if (write(pipefd[1], message, strlen(message)) == -1) {**

**perror("Error writing to pipe");**

**}**

**// Close the write end of the pipe**

**close(pipefd[1]);**

**// Wait for the child process to finish**

**waitpid(child\_pid, NULL, 0);**

**}**

**return EXIT\_SUCCESS;**

**}**

**(SLIP-10)**

**SUBJECT: CS-504-MJP: Lab Course on CS-501-MJ (Advanced Operating System)**

**Q1) Write a C program to get and set the resource limits such as files, memory associated with a process**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <sys/resource.h>**

**void printResourceLimits() {**

**struct rlimit limit;**

**// Get the current resource limit for the maximum number of file descriptors**

**if (getrlimit(RLIMIT\_NOFILE, &limit) == -1) {**

**perror("getrlimit");**

**exit(EXIT\_FAILURE);**

**}**

**printf("Current RLIMIT\_NOFILE (max file descriptors): %ld\n", limit.rlim\_cur);**

**// Get the current resource limit for the virtual memory size**

**if (getrlimit(RLIMIT\_AS, &limit) == -1) {**

**perror("getrlimit");**

**exit(EXIT\_FAILURE);**

**}**

**printf("Current RLIMIT\_AS (address space): %ld bytes\n", limit.rlim\_cur);**

**}**

**void setResourceLimits() {**

**struct rlimit newLimit;**

**// Set a new resource limit for the maximum number of file descriptors**

**newLimit.rlim\_cur = 1000;**

**newLimit.rlim\_max = 2000;**

**if (setrlimit(RLIMIT\_NOFILE, &newLimit) == -1) {**

**perror("setrlimit");**

**exit(EXIT\_FAILURE);**

**}**

**printf("New RLIMIT\_NOFILE (max file descriptors) set to: %ld\n", newLimit.rlim\_cur);**

**// Set a new resource limit for the virtual memory size (e.g., 1 GB)**

**newLimit.rlim\_cur = 1 \* 1024 \* 1024 \* 1024; // 1 GB**

**newLimit.rlim\_max = 2 \* 1024 \* 1024 \* 1024; // 2 GB**

**if (setrlimit(RLIMIT\_AS, &newLimit) == -1) {**

**perror("setrlimit");**

**exit(EXIT\_FAILURE);**

**}**

**printf("New RLIMIT\_AS (address space) set to: %ld bytes\n", newLimit.rlim\_cur);**

**}**

**int main() {**

**printf("Before setting resource limits:\n");**

**printResourceLimits();**

**setResourceLimits();**

**printf("\nAfter setting resource limits:\n");**

**printResourceLimits();**

**return EXIT\_SUCCESS;**

**}**