***Slip 6***

**Q.1) 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>

#include <string.h>

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

DIR \*dir;

struct dirent \*entry;

struct stat file\_stat;

struct tm \*timeinfo;

int target\_month;

// Get target month from command line (1-12)

if (argc != 2) {

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

return 1;

}

target\_month = atoi(argv[1]);

if (target\_month < 1 || target\_month > 12) {

printf("Month must be between 1 and 12\n");

return 1;

}

// Open current directory

dir = opendir(".");

if (dir == NULL) {

printf("Unable to open directory\n");

return 1;

}

// Read directory entries

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

// Skip . and ..

if (strcmp(entry->d\_name, ".") == 0 || strcmp(entry->d\_name, "..") == 0)

continue;

// Get file stats

if (stat(entry->d\_name, &file\_stat) == 0) {

timeinfo = localtime(&file\_stat.st\_ctime);

// Check if file was created in target month

if ((timeinfo->tm\_mon + 1) == target\_month) {

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

}

}

}

closedir(dir);

return 0;

}

**Q.2) 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,for,signal and**

**sleep system call1)**

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

#include <sys/types.h>

#include <stdlib.h>

// Signal handler function

void alarm\_handler(int signum) {

printf("Alarm fired! Signal %d received by parent process\n", signum);

}

int main() {

pid\_t pid;

// Set up signal handler in parent before fork

if (signal(SIGALRM, alarm\_handler) == SIG\_ERR) {

perror("signal");

exit(1);

}

// Create child process

pid = fork();

if (pid < 0) {

perror("fork");

exit(1);

}

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

printf("Child process (PID: %d) sending SIGALRM to parent (PID: %d)\n",

getpid(), getppid());

sleep(2); // Wait for 2 seconds before sending signal

// Send SIGALRM to parent

if (kill(getppid(), SIGALRM) == -1) {

perror("kill");

exit(1);

}

exit(0);

}

else { // Parent process

printf("Parent process (PID: %d) waiting for signal...\n", getpid());

// Wait for 5 seconds to receive signal

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

sleep(1);

}

}

return 0;

}

***Slip 11***

**Q.1) 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>

#include <unistd.h>

// Function to print the resource limit

void print\_limit(int resource, const char \*name) {

struct rlimit limit;

if (getrlimit(resource, &limit) == 0) {

printf("%s limits:\n", name);

printf(" Soft limit: %ld\n", (long)limit.rlim\_cur);

printf(" Hard limit: %ld\n\n", (long)limit.rlim\_max);

} else {

perror("getrlimit");

exit(1);

}

}

// Function to set a new soft limit for a resource

void set\_soft\_limit(int resource, rlim\_t new\_soft\_limit) {

struct rlimit limit;

if (getrlimit(resource, &limit) != 0) {

perror("getrlimit");

exit(1);

}

limit.rlim\_cur = new\_soft\_limit; // Set the new soft limit

if (setrlimit(resource, &limit) != 0) {

perror("setrlimit");

exit(1);

}

}

int main() {

// Display current limits for open files

print\_limit(RLIMIT\_NOFILE, "Open files");

// Set a new soft limit for open files (e.g., 1024)

printf("Setting new soft limit for open files to 1024...\n");

set\_soft\_limit(RLIMIT\_NOFILE, 1024);

print\_limit(RLIMIT\_NOFILE, "Open files");

// Display current limits for virtual memory

print\_limit(RLIMIT\_AS, "Virtual memory");

// Set a new soft limit for virtual memory (e.g., 512 MB)

printf("Setting new soft limit for virtual memory to 512 MB...\n");

set\_soft\_limit(RLIMIT\_AS, 512 \* 1024 \* 1024); // 512 MB in bytes

print\_limit(RLIMIT\_AS, "Virtual memory");

return 0;

}

***Slip 12***

**Q.1) Write a C program that print the exit status of a terminated child process**

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

int main() {

pid\_t pid = fork();

if (pid < 0) {

// Error in fork

perror("Fork failed");

exit(1);

} else if (pid == 0) {

// Child process

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

// Exit with a specific status, e.g., 5

exit(5);

} else {

// Parent process

int status;

pid\_t child\_pid = wait(&status); // Wait for the child process to terminate

if (child\_pid == -1) {

perror("wait failed");

exit(1);

}

printf("Child process (PID: %d) terminated. ", child\_pid);

// Check if the child exited normally

if (WIFEXITED(status)) {

int exit\_status = WEXITSTATUS(status);

printf("Exit status: %d\n", exit\_status);

} else {

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

}

}

return 0;

}

***Slip 13***

**Q.1) Write a C program that illustrates suspending and resuming processes using signals**

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

#include <stdlib.h>

#include <sys/wait.h>

// Signal handler for SIGCONT

void cont\_handler(int signum) {

printf("Process %d resumed\n", getpid());

}

int main() {

pid\_t pid;

// Register SIGCONT handler

if (signal(SIGCONT, cont\_handler) == SIG\_ERR) {

perror("signal");

exit(1);

}

// Create child process

pid = fork();

if (pid < 0) {

perror("fork");

exit(1);

}

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

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

// Simulate continuous work

while(1) {

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

sleep(1);

}

}

else { // Parent process

printf("Parent process (PID: %d) controlling child (PID: %d)\n",

getpid(), pid);

// Give child time to start

sleep(3);

// Suspend child

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

if (kill(pid, SIGSTOP) == -1) {

perror("kill (SIGSTOP)");

exit(1);

}

printf("Child process suspended. Waiting 3 seconds...\n");

sleep(3);

// Resume child

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

if (kill(pid, SIGCONT) == -1) {

perror("kill (SIGCONT)");

exit(1);

}

sleep(3);

// Terminate child

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

if (kill(pid, SIGTERM) == -1) {

perror("kill (SIGTERM)");

exit(1);

}

// Wait for child to terminate

wait(NULL);

printf("Child process terminated\n");

}

return 0;

}

***Slip 14***

**Q.1) 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>

#include <stdlib.h>

#include <string.h>

#include <errno.h>

int main() {

DIR \*dir;

struct dirent \*entry;

struct stat file\_stat;

long min\_size;

int found = 0;

// Get minimum file size from user

printf("Enter minimum file size in bytes: ");

if (scanf("%ld", &min\_size) != 1) {

fprintf(stderr, "Invalid input\n");

return 1;

}

// Open current directory

dir = opendir(".");

if (dir == NULL) {

perror("Failed to open directory");

return 1;

}

printf("\nFiles larger than %ld bytes:\n", min\_size);

printf("--------------------------------\n");

// Read directory entries

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

// Skip . and ..

if (strcmp(entry->d\_name, ".") == 0 || strcmp(entry->d\_name, "..") == 0) {

continue;

}

// Get file statistics

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

fprintf(stderr, "Cannot stat %s: %s\n",

entry->d\_name, strerror(errno));

continue;

}

// Check if it's a regular file and size condition

if (S\_ISREG(file\_stat.st\_mode) && file\_stat.st\_size > min\_size) {

printf("%-30s %10ld bytes\n", entry->d\_name, file\_stat.st\_size);

found++;

}

}

// Print summary

printf("--------------------------------\n");

printf("Total files found: %d\n", found);

closedir(dir);

return 0;

}

***Slip 15***

**Q.2) 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**

#include <stdio.h>

#include <stdlib.h>

#include <signal.h>

#include <unistd.h>

#include <sys/wait.h>

pid\_t child\_pid = 0;

// Signal handler for SIGCHLD (child termination)

void handle\_sigchld(int sig) {

int status;

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

if (WIFEXITED(status)) {

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

} else {

printf("Child terminated abnormally.\n");

}

exit(0);

}

// Signal handler for SIGALRM (timeout)

void handle\_sigalrm(int sig) {

printf("Child did not finish in time. Terminating child...\n");

kill(child\_pid, SIGKILL);

}

int main() {

signal(SIGCHLD, handle\_sigchld);

signal(SIGALRM, handle\_sigalrm);

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

// Child process: Replace with command (e.g., `sleep 10`)

execlp("sleep", "sleep", "10", NULL);

perror("execlp failed");

exit(1);

} else if (child\_pid > 0) {

// Parent process: Set a 5-second alarm

alarm(5);

pause(); // Wait for signals

} else {

perror("fork failed");

exit(1);

}

return 0;

}

***Slip 16***

**Q.2) 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 seconparent send sigquit signal to child and child terminates my displaying message “My DADDY hasKilled me!!!”.**

#include <stdio.h>

#include <stdlib.h>

#include <signal.h>

#include <unistd.h>

void handle\_sighup(int sig) {

printf("Child received SIGHUP\n");

}

void handle\_sigint(int sig) {

printf("Child received SIGINT\n");

}

void handle\_sigquit(int sig) {

printf("My DADDY has killed me!!!\n");

exit(0);

}

int main() {

pid\_t pid = fork();

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

// Set up signal handlers

signal(SIGHUP, handle\_sighup);

signal(SIGINT, handle\_sigint);

signal(SIGQUIT, handle\_sigquit);

// Child process waits indefinitely for signals

while (1) {

pause(); // Wait for signals

}

} else if (pid > 0) { // Parent process

for (int i = 0; i < 10; i++) {

// Alternate sending SIGHUP and SIGINT every 3 seconds

sleep(3);

if (i % 2 == 0) {

kill(pid, SIGHUP);

} else {

kill(pid, SIGINT);

}

}

// After 30 seconds, send SIGQUIT to terminate the child

kill(pid, SIGQUIT);

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

} else {

perror("fork failed");

exit(1);

}

return 0;

}

***Slip 17***

**Q.1) 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 file\_count = 0;

// Open the current directory

dir = opendir(".");

if (dir == NULL) {

perror("Unable to open current directory");

return 1;

}

printf("Files in the current directory:\n");

// Read and display each entry in the directory

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

// Only consider regular files (ignore directories and hidden files)

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

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

file\_count++;

}

}

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

// Close the directory

closedir(dir);

return 0;

}

***Slip18***

**Q.1) Write a C program to find whether a given file is present in current directory or not**

#include <stdio.h>

#include <stdlib.h>

#include <dirent.h>

#include <string.h>

int main() {

DIR \*dir;

struct dirent \*entry;

char file\_name[100];

int file\_found = 0;

// Ask the user to input the file name to search

printf("Enter the file name to search: ");

scanf("%s", file\_name);

// Open the current directory

dir = opendir(".");

if (dir == NULL) {

perror("Unable to open directory");

return 1;

}

// Search for the file in the directory

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

if (strcmp(entry->d\_name, file\_name) == 0) {

file\_found = 1;

break;

}

}

// Display the result

if (file\_found) {

printf("File '%s' is present in the current directory.\n", file\_name);

} else {

printf("File '%s' is not found in the current directory.\n", file\_name);

}

// Close the directory

closedir(dir);

return 0;

}

**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”**

#include <stdio.h>

#include <unistd.h>

#include <string.h>

#include <stdlib.h>

int main() {

int pipe\_fd[2]; // Pipe file descriptors

pid\_t pid;

char buffer[100];

// Create the pipe

if (pipe(pipe\_fd) == -1) {

perror("Pipe creation failed");

return 1;

}

// Create a child process

pid = fork();

if (pid == -1) {

perror("Fork failed");

return 1;

}

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

// Close the read end of the pipe

close(pipe\_fd[0]);

// Write messages to the pipe

write(pipe\_fd[1], "Hello World\n", strlen("Hello World\n"));

write(pipe\_fd[1], "Hello SPPU\n", strlen("Hello SPPU\n"));

write(pipe\_fd[1], "Linux is Funny\n", strlen("Linux is Funny\n"));

// Close the write end of the pipe

close(pipe\_fd[1]);

} else { // Parent process

// Close the write end of the pipe

close(pipe\_fd[1]);

// Read and display messages from the pipe

while (read(pipe\_fd[0], buffer, sizeof(buffer)) > 0) {

printf("Parent received: %s", buffer);

}

// Close the read end of the pipe

close(pipe\_fd[0]);

// Wait for the child process to finish

wait(NULL);

}

return 0;

}

***Slip 21***

**Q.2) 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 <string.h>

#include <sys/stat.h>

#define MAX\_FILES 100

// Function to compare file sizes

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

struct stat stat\_a, stat\_b;

// Get the file size of the first file

if (stat(\*(const char \*\*)a, &stat\_a) == -1) {

perror("Error getting file size");

return 0;

}

// Get the file size of the second file

if (stat(\*(const char \*\*)b, &stat\_b) == -1) {

perror("Error getting file size");

return 0;

}

// Compare the file sizes

return stat\_a.st\_size - stat\_b.st\_size;

}

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

if (argc < 2) {

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

return 1;

}

// Sort the file names based on their size

qsort(&argv[1], argc - 1, sizeof(char \*), compare\_file\_sizes);

printf("Files sorted by size in ascending order:\n");

// Display the sorted filenames

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

struct stat stat\_buf;

if (stat(argv[i], &stat\_buf) == -1) {

perror("Error getting file size");

continue;

}

printf("%s (Size: %ld bytes)\n", argv[i], stat\_buf.st\_size);

}

return 0;

}

***Slip 22***

**Q.1) Write a C Program that demonstrates redirection of standard output to a file**

#include <stdio.h>

int main() {

// Open a file for writing (the file will be created if it doesn't exist)

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

if (file == NULL) {

perror("Error redirecting stdout to file");

return 1;

}

// Now all the printf statements will be written to output.txt

printf("This is a test message written to the file.\n");

printf("Redirection of standard output is successful!\n");

// Close the file after writing

fclose(file);

return 0;

}

***Slip 23***

**Q.2) 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/stat.h>

#include <unistd.h>

// Function to print the file type

void identify\_file\_type(const char \*path) {

struct stat statbuf;

// Get the file status

if (stat(path, &statbuf) == -1) {

perror("stat");

return;

}

// Check the file type

if (S\_ISDIR(statbuf.st\_mode)) {

printf("The file is a Directory.\n");

}

else if (S\_ISCHR(statbuf.st\_mode)) {

printf("The file is a Character device.\n");

}

else if (S\_ISBLK(statbuf.st\_mode)) {

printf("The file is a Block device.\n");

}

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

printf("The file is a Regular file.\n");

}

else if (S\_ISFIFO(statbuf.st\_mode)) {

printf("The file is a FIFO (Named pipe).\n");

}

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

printf("The file is a Symbolic link.\n");

}

else if (S\_ISSOCK(statbuf.st\_mode)) {

printf("The file is a Socket.\n");

}

else {

printf("The file type is unknown.\n");

}

}

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

if (argc != 2) {

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

return 1;

}

// Call the function to identify the file type

identify\_file\_type(argv[1]);

return 0;

}

***Slip 24***

**Q1. Print the type of file and inode number where file name accepted through Command Line**

#include <stdio.h>

#include <stdlib.h>

#include <sys/stat.h>

#include <unistd.h>

// Function to print the file type

void print\_file\_info(const char \*path) {

struct stat statbuf;

// Get the file status

if (stat(path, &statbuf) == -1) {

perror("stat");

return;

}

// Print the inode number

printf("Inode number: %ld\n", statbuf.st\_ino);

// Check the file type and print it

if (S\_ISDIR(statbuf.st\_mode)) {

printf("The file is a Directory.\n");

}

else if (S\_ISCHR(statbuf.st\_mode)) {

printf("The file is a Character device.\n");

}

else if (S\_ISBLK(statbuf.st\_mode)) {

printf("The file is a Block device.\n");

}

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

printf("The file is a Regular file.\n");

}

else if (S\_ISFIFO(statbuf.st\_mode)) {

printf("The file is a FIFO (Named pipe).\n");

}

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

printf("The file is a Symbolic link.\n");

}

else if (S\_ISSOCK(statbuf.st\_mode)) {

printf("The file is a Socket.\n");

}

else {

printf("The file type is unknown.\n");

}

}

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

if (argc != 2) {

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

return 1;

}

// Call the function to print the file type and inode number

print\_file\_info(argv[1]);

return 0;

}