**Assignment 1**

**Set A**

**1. Create a child process using fork(), display parent and child process id. Child**

**process will display the message “Hello World” and the parent process should display**

**“Hi”.**

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

int main() {

pid\_t pid;

pid = fork();

if (pid < 0) {

perror("fork failed");

return 1;

}

if (pid == 0) {

printf("Child Process: PID = %d, Parent PID = %d\n", getpid(), getppid());

printf("Hello World\n");

} else {

printf("Parent Process: PID = %d, Child PID = %d\n", getpid(), pid);

printf("Hi\n");

}

return 0;

}

**2. Creating a child process using the command exec(). Note down process ids of the**

**parent and the child processes, check whether the control is given back to the parent**

**after the child process terminates. Write a similar program using execv() and execvp()**

**and observe the differences in behaviours of the commands.**

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

int main() {

pid\_t pid;

pid = fork(); // Create a new process

if (pid < 0) {

// Fork failed

perror("fork");

return 1;

}

if (pid == 0) {

// Child process

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

printf("Parent process ID of child: %d\n", getppid());

execlp("ls", "ls", NULL); // Replace child process image with 'ls'

// execlp only returns on failure

perror("execlp");

return 1;

} else {

// Parent process

printf("Parent process ID: %d\n", getpid());

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

wait(NULL); // Wait for child process to terminate

printf("Child process terminated\n");

}

return 0;

}

**3. Creating a child process without terminating the parent process Write a program to**

**create a child process using fork().The parent should goto sleep state and child**

**process should begin its execution. In the child process, use execl() to execute the “ls”**

**Command.**

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

int main() {

pid\_t pid;

pid = fork(); // Create a new process

if (pid < 0) {

// Fork failed

perror("fork");

return 1;

}

if (pid == 0) {

// Child process

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

printf("Parent process ID of child: %d\n", getppid());

// Replace child process image with 'ls' command

execl("/bin/ls", "ls", NULL);

// If execl() fails, it returns -1

perror("execl");

return 1;

} else {

// Parent process

printf("Parent process ID: %d\n", getpid());

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

// Sleep for 10 seconds

printf("Parent going to sleep for 10 seconds...\n");

sleep(10);

// Optionally wait for child process to complete

// wait(NULL);

printf("Parent process waking up after sleep.\n");

}

return 0;

}

**Set B**

**1. Write a program to illustrate the concept of orphan process ( Using fork() and**

**sleep())( Refer Program 5).**

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <stdlib.h>

#include <sys/wait.h>

int main() {

pid\_t pid;

pid = fork(); // Create a new process

if (pid < 0) {

// Fork failed

perror("fork");

return 1;

}

if (pid == 0) {

// Child process

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

printf("Parent process ID of child (before parent exit): %d\n", getppid());

// Sleep for 20 seconds

sleep(20);

// After parent exits, the child process will be adopted by init

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

printf("Parent process ID of child (after parent exit): %d\n", getppid());

} else {

// Parent process

printf("Parent process ID: %d\n", getpid());

// Parent process exits immediately

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

exit(0);

}

return 0;

}

**2. Write a program that demonstrates the use of nice() system call. After a child**

**process is started using fork(), assign higher priority to the child using nice() system**

**Call.**

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

int main() {

pid\_t pid;

// Fork to create a new child process

pid = fork();

if (pid < 0) {

// Fork failed

perror("fork");

return 1;

}

if (pid == 0) {

// Child process

printf("Child process (PID: %d) before nice(): %d\n", getpid(), nice(0));

// Set a higher priority (lower nice value) for the child process

nice(-10); // Decrease nice value by 10 (higher priority)

// Check new nice value

printf("Child process (PID: %d) after nice(): %d\n", getpid(), nice(0));

// Child process sleeps for 10 seconds

sleep(10);

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

} else {

// Parent process

printf("Parent process (PID: %d) before nice(): %d\n", getpid(), nice(0));

// Set a higher nice value (lower priority) for the parent process

nice(10); // Increase nice value by 10 (lower priority)

// Check new nice value

printf("Parent process (PID: %d) after nice(): %d\n", getpid(), nice(0));

// Parent process waits for the child process to complete

wait(NULL);

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

}

return 0;

}

**3. Write a program to find the execution time taken for execution of a given set of**

**instructions (Hint: use clock() function. This function clock() is called at the**

**beginning of program and again at the end of the program and the difference between**

**the values returned gives the time spent by processor on the program.)**

#include <stdio.h>

#include <time.h>

// Function to simulate some work

void perform\_task() {

long long sum = 0;

for (long long i = 0; i < 1000000000; i++) {

sum += i;

}

printf("Task complete. Sum = %lld\n", sum);

}

int main() {

// Record the start time

clock\_t start\_time = clock();

// Perform the task

perform\_task();

// Record the end time

clock\_t end\_time = clock();

// Calculate the elapsed time

double elapsed\_time = (double)(end\_time - start\_time) / CLOCKS\_PER\_SEC;

// Print the elapsed time

printf("Time taken for execution: %f seconds\n", elapsed\_time);

return 0;

}

**Set C(Shell Command)**

**1.Implement the shell program that accepts the command at $ prompt displayed by**

**your shell (myshell$). Implement the „count‟ command by creating child process**

**which works as follows:**

**myshell$ count c filename: To display the number of characters in given file**

**myshell$ count w filename: To display the number of words in given file**

**myshell$ count l filename: To display the number of lines in given file**

**Myshell.c**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <string.h>**

**#include <unistd.h>**

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

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

**#define MAX\_INPUT\_SIZE 1024**

**#define MAX\_ARG\_SIZE 256**

**void count\_characters(const char \*filename) {**

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

**if (!file) {**

**perror("fopen");**

**exit(EXIT\_FAILURE);**

**}**

**int count = 0;**

**int ch;**

**while ((ch = fgetc(file)) != EOF) {**

**count++;**

**}**

**fclose(file);**

**printf("Character count: %d\n", count);**

**}**

**void count\_words(const char \*filename) {**

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

**if (!file) {**

**perror("fopen");**

**exit(EXIT\_FAILURE);**

**}**

**int count = 0;**

**int in\_word = 0;**

**int ch;**

**while ((ch = fgetc(file)) != EOF) {**

**if (ch == ' ' || ch == '\n' || ch == '\t') {**

**in\_word = 0;**

**} else {**

**if (!in\_word) {**

**count++;**

**in\_word = 1;**

**}**

**}**

**}**

**fclose(file);**

**printf("Word count: %d\n", count);**

**}**

**void count\_lines(const char \*filename) {**

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

**if (!file) {**

**perror("fopen");**

**exit(EXIT\_FAILURE);**

**}**

**int count = 0;**

**int ch;**

**while ((ch = fgetc(file)) != EOF) {**

**if (ch == '\n') {**

**count++;**

**}**

**}**

**fclose(file);**

**printf("Line count: %d\n", count);**

**}**

**int main() {**

**char input[MAX\_INPUT\_SIZE];**

**char \*args[MAX\_ARG\_SIZE];**

**int status;**

**while (1) {**

**printf("myshell$ ");**

**fflush(stdout);**

**if (!fgets(input, MAX\_INPUT\_SIZE, stdin)) {**

**perror("fgets");**

**exit(EXIT\_FAILURE);**

**}**

**// Remove newline character if present**

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

**// Tokenize the input string**

**int i = 0;**

**args[i] = strtok(input, " ");**

**while (args[i] != NULL) {**

**i++;**

**args[i] = strtok(NULL, " ");**

**}**

**// Check if 'count' command is used**

**if (args[0] != NULL && strcmp(args[0], "count") == 0) {**

**if (args[1] == NULL || args[2] == NULL) {**

**fprintf(stderr, "Usage: count [c|w|l] filename\n");**

**continue;**

**}**

**pid\_t pid = fork();**

**if (pid < 0) {**

**perror("fork");**

**exit(EXIT\_FAILURE);**

**}**

**if (pid == 0) {**

**// Child process**

**if (strcmp(args[1], "c") == 0) {**

**count\_characters(args[2]);**

**} else if (strcmp(args[1], "w") == 0) {**

**count\_words(args[2]);**

**} else if (strcmp(args[1], "l") == 0) {**

**count\_lines(args[2]);**

**} else {**

**fprintf(stderr, "Invalid option. Use 'c', 'w', or 'l'.\n");**

**}**

**exit(EXIT\_SUCCESS);**

**} else {**

**// Parent process**

**waitpid(pid, &status, 0);**

**}**

**} else {**

**fprintf(stderr, "Unknown command: %s\n", args[0]);**

**}**

**}**

**return 0;**

**}**

**gcc -o myshell myshell.c**

**./myshell**

**2. Extend the shell to implement the commands „list‟ which works as follows:**

**myshell$ list f dirname: It will display filenames in a given directory.**

**myshell$ list ndirname: It will count the number of entries in a given directory.**

**myshell$ list i dirname: It will display filenames and their inode number for**

**the files in a given directory.**

**myshell.c**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <dirent.h>

#include <sys/stat.h>

#include <errno.h>

#define MAX\_INPUT\_SIZE 1024

#define MAX\_ARG\_SIZE 256

void count\_characters(const char \*filename) {

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

if (!file) {

perror("fopen");

exit(EXIT\_FAILURE);

}

int count = 0;

int ch;

while ((ch = fgetc(file)) != EOF) {

count++;

}

fclose(file);

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

}

void count\_words(const char \*filename) {

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

if (!file) {

perror("fopen");

exit(EXIT\_FAILURE);

}

int count = 0;

int in\_word = 0;

int ch;

while ((ch = fgetc(file)) != EOF) {

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

in\_word = 0;

} else {

if (!in\_word) {

count++;

in\_word = 1;

}

}

}

fclose(file);

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

}

void count\_lines(const char \*filename) {

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

if (!file) {

perror("fopen");

exit(EXIT\_FAILURE);

}

int count = 0;

int ch;

while ((ch = fgetc(file)) != EOF) {

if (ch == '\n') {

count++;

}

}

fclose(file);

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

}

void list\_files(const char \*dirname) {

DIR \*dir = opendir(dirname);

if (!dir) {

perror("opendir");

exit(EXIT\_FAILURE);

}

struct dirent \*entry;

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

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

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

}

}

closedir(dir);

}

void count\_entries(const char \*dirname) {

DIR \*dir = opendir(dirname);

if (!dir) {

perror("opendir");

exit(EXIT\_FAILURE);

}

int count = 0;

struct dirent \*entry;

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

count++;

}

closedir(dir);

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

}

void list\_files\_inodes(const char \*dirname) {

DIR \*dir = opendir(dirname);

if (!dir) {

perror("opendir");

exit(EXIT\_FAILURE);

}

struct dirent \*entry;

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

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

struct stat file\_stat;

char path[PATH\_MAX];

snprintf(path, sizeof(path), "%s/%s", dirname, entry->d\_name);

if (stat(path, &file\_stat) == 0) {

printf("%s - Inode: %lu\n", entry->d\_name, (unsigned long)file\_stat.st\_ino);

} else {

perror("stat");

}

}

}

closedir(dir);

}

int main() {

char input[MAX\_INPUT\_SIZE];

char \*args[MAX\_ARG\_SIZE];

int status;

while (1) {

printf("myshell$ ");

fflush(stdout);

if (!fgets(input, MAX\_INPUT\_SIZE, stdin)) {

perror("fgets");

exit(EXIT\_FAILURE);

}

// Remove newline character if present

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

// Tokenize the input string

int i = 0;

args[i] = strtok(input, " ");

while (args[i] != NULL) {

i++;

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

}

// Check if 'count' command is used

if (args[0] != NULL && strcmp(args[0], "count") == 0) {

if (args[1] == NULL || args[2] == NULL) {

fprintf(stderr, "Usage: count [c|w|l] filename\n");

continue;

}

pid\_t pid = fork();

if (pid < 0) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (pid == 0) {

// Child process

if (strcmp(args[1], "c") == 0) {

count\_characters(args[2]);

} else if (strcmp(args[1], "w") == 0) {

count\_words(args[2]);

} else if (strcmp(args[1], "l") == 0) {

count\_lines(args[2]);

} else {

fprintf(stderr, "Invalid option. Use 'c', 'w', or 'l'.\n");

}

exit(EXIT\_SUCCESS);

} else {

// Parent process

waitpid(pid, &status, 0);

}

}

// Check if 'list' command is used

else if (args[0] != NULL && strcmp(args[0], "list") == 0) {

if (args[1] == NULL || args[2] == NULL) {

fprintf(stderr, "Usage: list [f|ndir|i] dirname\n");

continue;

}

pid\_t pid = fork();

if (pid < 0) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (pid == 0) {

// Child process

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

list\_files(args[2]);

} else if (strcmp(args[1], "ndir") == 0) {

count\_entries(args[2]);

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

list\_files\_inodes(args[2]);

} else {

fprintf(stderr, "Invalid option. Use 'f', 'ndir', or 'i'.\n");

}

exit(EXIT\_SUCCESS);

} else {

// Parent process

waitpid(pid, &status, 0);

}

} else {

fprintf(stderr, "Unknown command: %s\n", args[0]);

}

}

return 0;

}

**gcc -o myshell myshell.c**

**./myshell**