**Slip 1**

**Write a C program to implement the following unix/linux command (use fork, pipe and execsystem call).**

**ls –l | wc–l**

#include<stdio.h>

#include<unistd.h>

int main()

{

int fd[2],dupFd;

char \*filename1 ="ls";

char \*filename2 ="wc";

char \*arg1 = "-l";

pipe(fd);

if(!fork())// return 0 for child process and 1 for parent process

{

close(1); // 1 for closing stdout

dup(fd[1]);

close(fd[0]);

execlp(filename1,filename1,arg1,NULL);

}else

{

close(0);

dup(fd[0]);

close(fd[1]);

execlp(filename2,filename2,arg1,NULL);

}

}

**Slip 2**

**Write a C program to create ‘n’ child processes. When all ‘n’ child processes terminates, display total cumulative time children spent in user and kernel mode.**

**#include <stdio.h>**

**#include <stdlib.h>**

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

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

**#include <unistd.h>**

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

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

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

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

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

**exit(1);**

**}**

**int num\_processes = atoi(argv[1]);**

**pid\_t pid;**

**int i;**

**struct timeval start\_time, end\_time;**

**double total\_user\_time = 0.0, total\_kernel\_time = 0.0;**

**gettimeofday(&start\_time, NULL);**

**for (i = 0; i < num\_processes; i++) {**

**pid = fork();**

**if (pid < 0) {**

**printf("Fork failed.\n");**

**exit(1);**

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

**// Child process**

**printf("Child process %d starting.\n", getpid());**

**exit(0);**

**}**

**}**

**// Parent process waits for all child processes to terminate**

**while (wait(NULL) > 0);**

**gettimeofday(&end\_time, NULL);**

**// Calculate the total user time and kernel time**

**struct rusage usage;**

**for (i = 0; i < num\_processes; i++) {**

**getrusage(RUSAGE\_CHILDREN, &usage);**

**total\_user\_time += usage.ru\_utime.tv\_sec + (usage.ru\_utime.tv\_usec / 1000000.0);**

**total\_kernel\_time += usage.ru\_stime.tv\_sec + (usage.ru\_stime.tv\_usec / 1000000.0);**

**}**

**double total\_time = (end\_time.tv\_sec - start\_time.tv\_sec) + ((end\_time.tv\_usec - start\_time.tv\_usec) / 1000000.0);**

**printf("All child processes have terminated.\n");**

**printf("Total user time: %f seconds\n", total\_user\_time);**

**printf("Total kernel time: %f seconds\n", total\_kernel\_time);**

**printf("Total elapsed time: %f seconds\n", total\_time);**

**return 0;**

**}**

**Slip 3**

**Write a C program to find properties of file such as inode number, number of hard link, File permissions, File size, of a given file using fstat() system call.**

**A)**

**#include <stdio.h>**

**#include <stdlib.h>**

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

**#include <unistd.h>**

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

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

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

**exit(1);**

**}**

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

**struct stat file\_info;**

**if (stat(filename, &file\_info) < 0) {**

**perror("Failed to stat file");**

**exit(1);**

**}**

**printf("Properties of file: %s\n", filename);**

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

**printf("Number of hard links: %ld\n", file\_info.st\_nlink);**

**printf("File permissions: %o\n", file\_info.st\_mode & 0777);**

**printf("File size: %ld bytes\n", file\_info.st\_size);**

**return 0;**

**}**

**B)**

**#include <stdio.h>**

**#include <unistd.h>**

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

**#include <time.h>**

**void printFileProperties(struct stat s);**

**int main()**

**{**

**char path[100];**

**struct stat s;**

**printf("Enter source file path: ");**

**scanf("%s", path);**

**// stat() returns 0 on successful operation,**

**// otherwise returns -1 if unable to get file properties.**

**if (stat(path, &s) == 0)**

**{**

**printFileProperties(s);**

**}**

**else**

**{**

**printf("Unable to get file properties.\n");**

**printf("Please check whether '%s' file exists.\n", path);**

**}**

**return 0;**

**}**

**/\*\***

**\* Function to print file properties.**

**\*/**

**void printFileProperties(struct stat s)**

**{**

**struct tm dt;**

**// File permissions**

**printf("\nFile access: ");**

**if (s.st\_mode & R\_OK)**

**printf("read ");**

**if (s.st\_mode & W\_OK)**

**printf("write ");**

**if (s.st\_mode & X\_OK)**

**printf("execute");**

**// File size**

**printf("\nFile size: %d", s.st\_size);**

**// Inode Number**

**printf("\n Inode Number: %d", s.st\_ino);**

**}**

**Slip 4**

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

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

int main()

{

int fd[2];

pid\_t pid;

char buf[100];

// Create the pipe

if (pipe(fd) < 0) {

fprintf(stderr, "Pipe creation failed\n");

exit(1);

}

// Fork the process

pid = fork();

if (pid < 0) {

fprintf(stderr, "Fork failed\n");

exit(1);

}

else if (pid == 0) {

// Child process

close(fd[0]); // Close unused read end

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

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

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

exit(0);

}

else {

// Parent process

close(fd[1]); // Close unused write end

printf("parent recieve msg:\n");

while (read(fd[0], buf, sizeof(buf)) > 0) {

printf("%s", buf);

}

printf("\n");

}

return 0;

}

**Slip 5**

**Write a C program to read all filenames in the current directory and display the inode number, number of links and size of each of the file.**

#include <stdio.h>

#include <dirent.h>

#include <sys/stat.h>

int main() {

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

if (!dir) {

perror("Failed to open directory");

return 1;

}

struct dirent \*entry;

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

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

char filename[100];

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

struct stat file\_info;

if (stat(filename, &file\_info) < 0) {

perror("Failed to stat file");

continue;

}

printf("%s: inode=%ld, links=%ld, size=%ld\n",

filename, file\_info.st\_ino, file\_info.st\_nlink, file\_info.st\_size);

}

}

closedir(dir);

return 0;

}

**Slip 6**

**Write a C program to create a file with hole in it. Read the file and show what data is read when the process reads at the offsets corresponding to hole.**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <unistd.h>**

**#include <fcntl.h>**

**int main()**

**{**

**int fd;**

**char buf[1024];**

**int n;**

**// Create the file with a hole**

**if ((fd = creat("file\_with\_hole.txt", 0666)) < 0) {**

**perror("creat error");**

**exit(1);**

**}**

**// Write some data to the file**

**if (write(fd, "Hello world", 11) != 11) {**

**perror("write error");**

**exit(1);**

**}**

**// Create a hole by seeking past the end of the data**

**if (lseek(fd, 16384, SEEK\_SET) == -1) {**

**perror("lseek error");**

**exit(1);**

**}**

**// Write some more data to the file**

**if (write(fd, "Goodbye world", 13) != 13) {**

**perror("write error");**

**exit(1);**

**}**

**// Close the file**

**close(fd);**

**// Open the file for reading**

**if ((fd = open("file\_with\_hole.txt", O\_RDONLY)) < 0) {**

**perror("open error");**

**exit(1);**

**}**

**// Read from the file and show what data is read at the offsets corresponding to the hole**

**while ((n = read(fd, buf, sizeof(buf))) > 0) {**

**int i;**

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

**if (buf[i] == '\0') {**

**printf("Read a null byte at offset %ld\n", lseek(fd, 0, SEEK\_CUR) - n + i);**

**}**

**}**

**if (write(STDOUT\_FILENO, buf, n) != n) {**

**perror("write error");**

**exit(1);**

**}**

**}**

**if (n < 0) {**

**perror("read error");**

**exit(1);**

**}**

**// Close the file**

**close(fd);**

**return 0;**

**}**

**Slip 7**

**Write a C program to read all filenames in the current directory and display the names of those regular files on which the user has “write” permission.**

**#include <stdio.h>**

**#include <stdlib.h>**

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

**#include <dirent.h>**

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

**#include <unistd.h>**

**int main()**

**{**

**DIR \*dir;**

**struct dirent \*ent;**

**struct stat st;**

**char \*cwd = getcwd(NULL, 0);**

**if ((dir = opendir(cwd)) == NULL) {**

**perror("opendir error");**

**exit(1);**

**}**

**while ((ent = readdir(dir)) != NULL) {**

**char \*filename = ent->d\_name;**

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

**perror("stat error");**

**exit(1);**

**}**

**if (S\_ISREG(st.st\_mode) && (st.st\_mode & S\_IWUSR)) {**

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

**}**

**}**

**closedir(dir);**

**free(cwd);**

**return 0;**

**}**

**Slip 8**

**Write a C program which receives file names as command line arguments and display the filenames in ascending order of their sizes.**

**#include<stdlib.h>**

**#include<stdio.h>**

**#include<string.h>**

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

**int main(int ac,char \*av[])**

**{**

**int i;**

**int max=0;**

**struct stat s;**

**if(ac<2)**

**{**

**printf("no commandline arg passed");**

**return 0;**

**}**

**else**

**{**

**for(i=1;i<=ac;i++)**

**{**

**stat(av[i],&s);**

**int size=s.st\_size;**

**if(size>max)**

**{**

**max=size;**

**printf("size of file %s=%ld\n",av[i],max);**

**}**

**}**

**}**

**}**

**Slip 9**

**Write a C program that will print list all subdirectories in current directory.**

#include <stdio.h>

#include <stdlib.h>

#include <dirent.h>

#include <sys/stat.h>

int main()

{

DIR \*dir;

struct dirent \*ent;

struct stat st;

char \*cwd = ".";

if ((dir = opendir(cwd)) == NULL) {

perror("opendir error");

exit(1);

}

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

char \*filename = ent->d\_name;

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

perror("stat error");

exit(1);

}

if (S\_ISDIR(st.st\_mode) && strcmp(filename, ".") != 0 && strcmp(filename, "..") != 0) {

printf("%s\n", filename);

}

}

closedir(dir);

return 0;

}

**Slip 10**

**Write a C program that redirects standard output to a file output.txt. (use dup and open system calls).**

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <unistd.h>

int main()

{

int fd, saved\_stdout;

saved\_stdout = dup(STDOUT\_FILENO); // save the current standard output

fd = open("output1.txt", O\_WRONLY | O\_CREAT | O\_TRUNC, S\_IRUSR | S\_IWUSR); // open the output file

if (fd == -1) {

perror("open error");

exit(1);

}

if (dup2(fd, STDOUT\_FILENO) == -1) { // redirect the standard output to the output file

perror("dup2 error");

exit(1);

}

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

if (dup2(saved\_stdout, STDOUT\_FILENO) == -1) { // restore the standard output

perror("dup2 error");

exit(1);

}

printf("This message will be written to the console.\n");

close(fd);

return 0;

}

**Slip 11**

**Write a C program to Identify the type (Directory, character device, Block device, Regular file, FIFO or pipe, symbolic link or socket) of files given as command line arguments.(use stat() system call).**

#include<stdlib.h>

#include<stdio.h>

#include<fcntl.h>

#include<string.h>

#include<unistd.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<dirent.h>

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

{

struct stat s;

int i;

if(argc<2)

{

printf("no command line argument passed");

}

else

{

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

{

stat(argv[i],&s);

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

printf("%s is regular file",argv[i]);

else if(S\_ISDIR(s.st\_mode))

printf("%s is directory file",argv[i]);

else if(S\_ISCHR(s.st\_mode))

printf("%s is character file",argv[i]);

else if(S\_ISFIFO(s.st\_mode))

printf("%s is pipe file",argv[i]);

else if(S\_ISLNK(s.st\_mode))

printf("%s is symbolic link file",argv[i]);

else if(S\_ISBLK(s.st\_mode))

printf("%s is BLOCK file",argv[i]);

else if(S\_ISSOCK(s.st\_mode))

printf("%s is socket file",argv[i]);

}

}

}

**Slip 12**

**Write a C program to move the content of file1.txt to file2.txt, remove the file file1.txt from directory and rename file2.txt as file1.txt.**

#include <stdio.h>

#include <stdlib.h>

int main()

{

FILE \*file1, \*file2;

int c;

// open file1 for reading

file1 = fopen("file1.txt", "r");

if (file1 == NULL) {

perror("file1.txt open error");

exit(1);

}

// open file2 for writing

file2 = fopen("file2.txt", "w");

if (file2 == NULL) {

perror("file2.txt open error");

exit(1);

}

// copy the contents of file1 to file2

while ((c = getc(file1)) != EOF) {

putc(c, file2);

}

// close the files

fclose(file1);

fclose(file2);

// remove file1

if (remove("file1.txt") == -1) {

perror("file1.txt remove error");

exit(1);

}

// rename file2 to file1

if (rename("file2.txt", "file1.txt") == -1) {

perror("file2.txt rename error");

exit(1);

}

printf("Content of file1.txt moved to file2.txt, file1.txt removed, and file2.txt renamed to file1.txt.\n");

return 0;

}

**Slip 13**

**Write a C program in which process creates a child process, which executes a linux/ unix command or any user defined program. Which is apassed as command line argument. The parent process waits till child process excutes. When child process completes its execution, parent prints pid and exit status of chils and its own pid and then terminates.**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <unistd.h>**

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

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

**pid\_t pid;**

**int status;**

**if (argc < 2) {**

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

**exit(1);**

**}**

**pid = fork();**

**if (pid < 0) {**

**perror("fork error");**

**exit(1);**

**}**

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

**// child process**

**if (execvp(argv[1], &argv[1]) < 0) {**

**perror("exec error");**

**exit(1);**

**}**

**} else {**

**// parent process**

**wait(&status);**

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

**printf("Parent process %d exited.\n", getpid());**

**exit(0);**

**}**

**return 0;**

**}**

**Slip 14**

**Write a C program to illustrate execution of atexit() function. Create two user defined functions. Register first function with atexit(), then register second function and then first again. Show how these functions will be excuted.**

#include<stdio.h>

#include<stdlib.h>

//#include<fntcl.h>

#include<sys/stat.h>

void first()

{

printf("U r in 1st\n");

}

void second()

{

printf("U r in 2nd\n");

}

int main()

{

atexit(first);

atexit(second);

atexit(first);

return 0;

}

**Slip 15**

**Write a C program to print the names of all the files along with their inode, which are created between 1 st May 2023 and 15th May 2023.**

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <dirent.h>

#include <sys/stat.h>

#include <time.h>

int main() {

DIR \*dir;

struct dirent \*entry;

struct stat file\_stat;

time\_t start\_time, end\_time;

start\_time = mktime(&(struct tm){.tm\_year=123, .tm\_mon=4, .tm\_mday=1}); // 1st May 2023

end\_time = mktime(&(struct tm){.tm\_year=123, .tm\_mon=4, .tm\_mday=15}); // 15th May 2023

dir = opendir(".");

if (dir == NULL) {

printf("Unable to open directory.");

exit(1);

}

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

stat(entry->d\_name, &file\_stat);

if (S\_ISREG(file\_stat.st\_mode) && file\_stat.st\_mtime >= start\_time && file\_stat.st\_mtime <= end\_time) {

printf("%s (inode: %ld)\n", entry->d\_name, file\_stat.st\_ino);

}

}

closedir(dir);

return 0;

}

**Slip 16**

**Write a C program which reads all files in the directory, the name of which has been passed as command line arguments and display the names of those files having file size greater than n (take n as input from user).**

**#include<stdio.h>**

**#include<stdlib.h>**

**#include<dirent.h>**

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

**#include<time.h>**

**#include<fcntl.h>**

**int main(int ac,char \*av[])**

**{**

**int n;**

**struct dirent \*d;**

**struct stat s;**

**DIR \*dr;**

**if((dr=opendir("."))!=NULL)**

**{**

**printf("Enter file size=");**

**scanf("%d",&n);**

**printf("Files whose size is greater than %d bytes are\n",n);**

**if(dr)**

**{**

**while((d=readdir(dr))!=NULL)**

**{**

**stat(d->d\_name,&s);**

**if(s.st\_size>n)**

**{**

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

**}**

**}**

**closedir(dr);**

**}**

**}**

**else**

**{**

**return EXIT\_FAILURE;**

**}**

**return EXIT\_SUCCESS;**

**}**

**Slip 17**

**Write a C program in which parent and child communicate using unnamed pipe.**

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <string.h>

int main() {

int pipefd[2];

pid\_t pid;

char message[] = "Hello, child process!";

char buffer[100];

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

printf("Error: Unable to create pipe.\n");

exit(EXIT\_FAILURE);

}

pid = fork();

if (pid == -1) {

printf("Error: Unable to create child process.\n");

exit(EXIT\_FAILURE);

}

if (pid == 0) {

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

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

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

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

exit(EXIT\_SUCCESS);

} else {

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

write(pipefd[1], message, strlen(message));

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

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

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

}

return 0;

}

**Slip 18**

**Write a C program to create two holes in a file with some data in between two holes. Read the contents of the file and display them.**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

int main() {

int fd;

char buffer1[] = "Hello";

char buffer2[] = "World";

char buffer3[] = "!!!";

char read\_buffer[20];

// Open the file for writing

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

if (fd == -1) {

printf("Error: Unable to open file for writing.\n");

exit(EXIT\_FAILURE);

}

// Write the first buffer to the file

write(fd, buffer1, sizeof(buffer1));

// Create a hole in the file by seeking past 100 bytes

lseek(fd, 100, SEEK\_CUR);

// Write the second buffer to the file

write(fd, buffer2, sizeof(buffer2));

// Create another hole in the file by seeking past another 100 bytes

lseek(fd, 100, SEEK\_CUR);

// Write the third buffer to the file

write(fd, buffer3, sizeof(buffer3));

// Close the file

close(fd);

// Open the file for reading

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

if (fd == -1) {

printf("Error: Unable to open file for reading.\n");

exit(EXIT\_FAILURE);

}

// Read the contents of the file and display them

int nbytes = read(fd, read\_buffer, sizeof(read\_buffer));

while (nbytes != 0) {

write(STDOUT\_FILENO, read\_buffer, nbytes);

nbytes = read(fd, read\_buffer, sizeof(read\_buffer));

}

// Close the file

close(fd);

return 0;

}

**Slip 19**

**Write a C program to show how the file read/write offset is shared using dup() system call.**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

int main() {

int fd1, fd2;

char buffer[20];

// Open the file for reading and writing

fd1 = open("file.txt", O\_RDWR | O\_CREAT, 0666);

if (fd1 == -1) {

printf("Error: Unable to open file for reading and writing.\n");

exit(EXIT\_FAILURE);

}

// Duplicate the file descriptor

fd2 = dup(fd1);

// Write some data to the file using the first file descriptor

write(fd1, "Hello, world!", 13);

// Read the data from the file using the second file descriptor

lseek(fd2, 0, SEEK\_SET);

int nbytes = read(fd2, buffer, sizeof(buffer));

if (nbytes == -1) {

printf("Error: Unable to read from file.\n");

exit(EXIT\_FAILURE);

}

// Print the data that was read from the file

printf("Data read from file: %.\*s\n", nbytes, buffer);

// Close the file descriptors

close(fd1);

close(fd2);

return 0;

}

**Slip 20**

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

#include<stdio.h>

#include<unistd.h>

int main()

{

int fd[2],dupFd;

char \*filename1 ="ls";

char \*filename2 ="wc";

char \*arg1 = "-l";

pipe(fd);

if(!fork())// return 0 for child process and 1 for parent process

{

close(1); // 1 for closing stdout

dup(fd[1]);

close(fd[0]);

execlp(filename1,filename1,arg1,NULL);

}else

{

close(0);

dup(fd[0]);

close(fd[1]);

execlp(filename2,filename2,arg1,NULL);

}

}