| **Fr. Conceicao Rodrigues College of Engineering**  **Department of Computer Engineering** | | | |
| --- | --- | --- | --- |
| **Student’s Roll No** | **9914** | **Students Name** | **Vivian Ludrick** |
| **Date of Performance** | **08/02/2024** | **SE Computer – Div** | **A** |

**Aim:** To study Process and File Management System Calls

**Lab Outcome:**

**CSL403.1: Demonstrate basic Operating system Commands, Shell scripts, System Calls and API wrt Linux.**

**Problem Statements:**

(1.) Process related System Calls.

a) Create a child process in Linux using the fork system call. From the child process obtain the process ID of both child and parent by using getpid and getppid system call.

b) Explore wait and waitpid before termination of process.

c) Explain ps command and output in detail. What is Zombie and Orphan Process? Show the output.

d) Explain fork(), getpid(), getppid(),wait() and waitpid() with syntax.

(2) File related system calls

a) Program to copy contents of one file (source) to another file (destination). Finally displaying contents of destination file.

b) 2. Explain creat(), open(), close(), read() and write() with syntax.

**CODE:**

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

#include<sys/wait.h>

#include <stdlib.h>

int main()

{

int pid, status;

pid = fork();

if (pid > 0){

printf("\npid of the current process is %d", getpid());

printf("\npid of the parent process is %d", getppid());

printf("\npid of the child process is %d", pid);

wait(&status);

printf("\nExiting the parent process");

}else{

printf("\npid of the current process is %d", getpid());

printf("\npid of the child parent is %d", getppid());

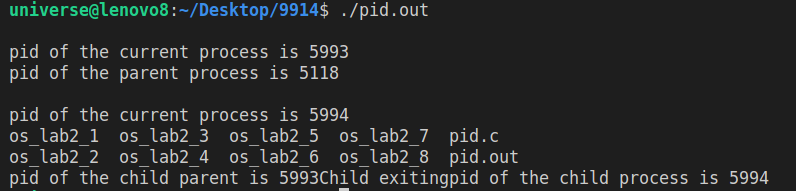
system("ls");

printf("Child exiting");

}

}

**OUTPUT:**

****

**CODE:**

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

#include <sys/wait.h>

#include <stdlib.h>

int main()

{

int pid, status;

pid = fork();

if (pid > 0){

printf("\npid of the current process is %d", getpid());

printf("\npid of the parent process is %d", getppid());

printf("\npid of the child process is %d", pid);

wait(&status);

printf("\nExiting the parent process");

}else{

printf("\npid of the current process is %d", getpid());

printf("\npid of the child parent is %d", getppid());

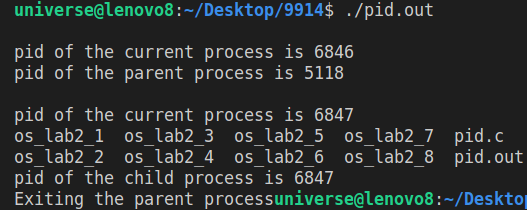
execl("/bin/ls", "ls", NULL );

printf("Child exiting");

}

}

**OUTPUT:**

****

**FILE OPERATIONS:**

**CODE:**

*// C program to illustrate*

*// open system call*

#include <fcntl.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

int main() {

int choice;

char og\_file[50], cp\_file[50], name[50], buff[1000], buff1[1000];

up:

*// get user choice*

printf("Enter the choice: \n1. Create\n2. Read and write\n3. Copy\n4.Exit");

scanf("%d", &choice);

switch (choice) {

case 1:

*// write in a file*

printf("Enter the name of the file:\t");

scanf("%49s", name);

*// open a new empty file with the default permissions*

int fd = open(name, O\_CREAT | O\_EXCL, 0644);

if (fd == -1) {

perror("Error: the file was not created");

exit(1);

} else {

printf("The file was create successfully");

close(fd); *// close the opened file buffer*

goto up;

}

case 2:

*// write in an already existing file*

printf("Enter the name of the file:\t");

scanf("%49s", name);

*// open the file with the read and write permissions*

fd = open(name, O\_RDWR);

if (fd == -1) {

perror("Error: the file couldn't be opened");

exit(2);

}

*// get the contents of the file*

printf("Enter the content to write in the file:\n");

*// some newline character filtering so that we can include the spaces*

*// i hate regex*

scanf(" %[^\n]", buff);

write(fd, buff, strlen(buff));

lseek(fd, 0, SEEK\_SET); *// set the file descriptor to the start of the file*

read(fd, buff1, strlen(buff));

printf("The contents of the file are:\n%s", buff1);

close(fd);

goto up;

case 3:

printf("Enter the name of source file:\t");

scanf("%49s", og\_file);

printf("Enter the name of the destination file:\t");

scanf("%49s", cp\_file);

*// open the source file*

fd = open(og\_file, O\_RDWR);

if (fd == -1) {

perror("Error: the file couldn't be opened");

exit(2);

}

*// save the content of the source file in a variable*

read(fd, buff, strlen(buff));

close(fd);

*// open the destination file*

fd = open(cp\_file, O\_RDWR);

if (fd == -1) {

perror("Error: the file couldn't be opened");

exit(2);

}

*// write the contents of the variable in the destination file*

write(fd, buff, strlen(buff));

lseek(fd, 0, SEEK\_SET); *// set the file descriptor to the start of the file*

read(fd, buff1, strlen(buff)); *// read the content of the destination file to verify a successful copy*

printf("The contents of the file are:\n%s", buff1);

close(fd);

goto up;

case 4:

printf("Exiting program\n");

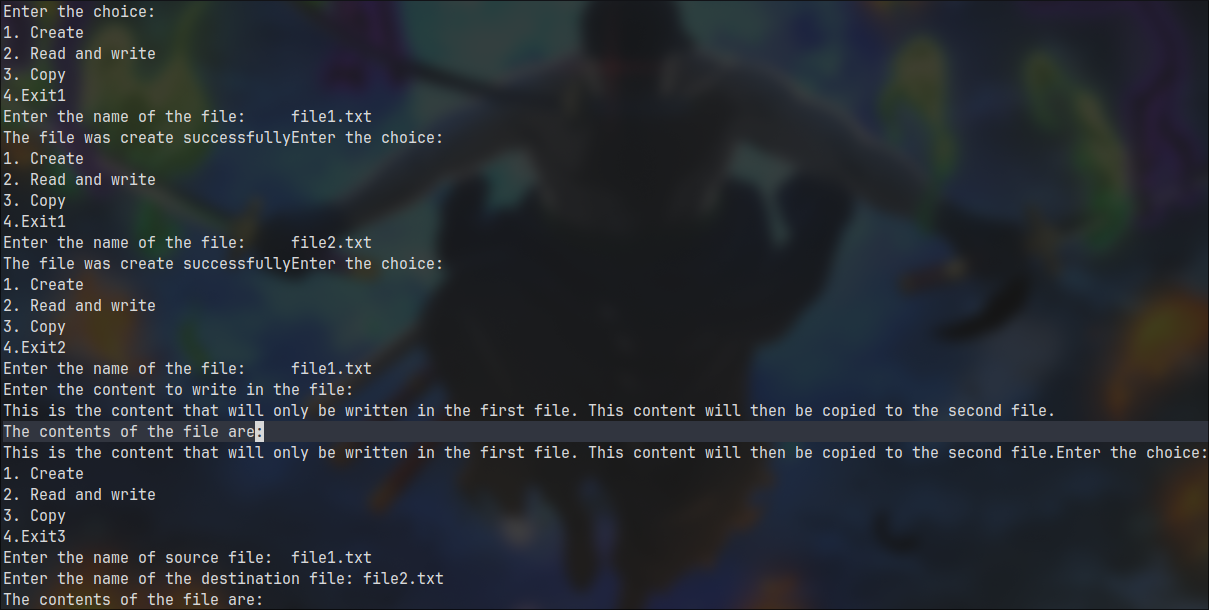
exit(EXIT\_SUCCESS);

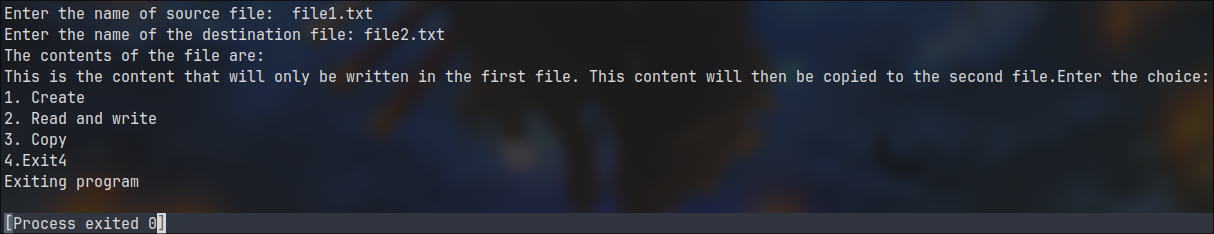
}

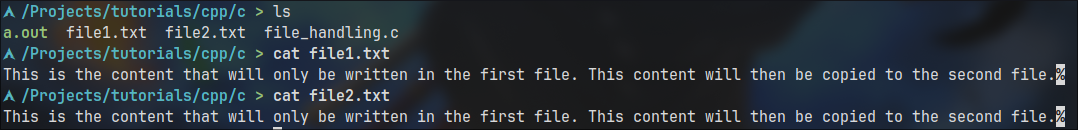
return 0;

}

**OUTPUT:**

****

****

****

**What is ps command?**

**⇒** ps displays information about a selection of the active processes. If you want a repetitive update of the selection and the displayed information, use top instead. By default, ps selects all processes with the same effective user ID (euid=EUID) as the current user and associated with the same terminal as the invoker. It displays the process ID (pid=PID), the terminal associated with the process (tname=TTY), the cumulated CPU time in [DD-]hh:mm:ss format (time=TIME), and the executable name (ucmd=CMD). Output is unsorted by default.

**What is zombie and orphan process? Show the output.**

**⇒**

**ZOMBIE PROCESS**

A process which has finished the execution but still has entry in the process table to report to its parent process is known as a zombie process. A child process always first becomes a zombie before being removed from the process table.

**CODE:**

*// A C program to demonstrate Zombie Process.*

*// Child becomes Zombie as parent is sleeping*

*// when child process exits.*

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <unistd.h>

int main() {

*// Fork returns process id*

*// in parent process*

pid\_t pid = fork();

*// Parent process*

if (pid > 0) {

sleep(10);

printf("The parent process has exited");

}

*// Child process*

else {

printf("The child process has exited");

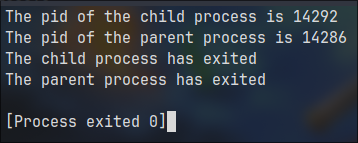
exit(0);

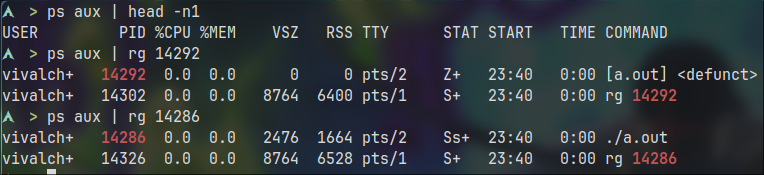
}

return 0;

}

**OUTPUT:**

****

****

**ORPHAN PROCESS:**

A process whose parent process no more exists i.e. either finished or terminated without waiting for its child process to terminate is called an orphan process.

**CODE:**

*// A C program to demonstrate Orphan Process.*

*// Parent process finishes execution while the*

*// child process is running. The child process*

*// becomes orphan.*

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

int main() {

*// Create a child process*

int pid = fork();

if (pid > 0) {

printf("The pid of the child process is %d\n", pid);

printf("The pid of the parent process is %d\n", getpid());

printf("in parent process");

} else if (pid == 0) {

sleep(60);

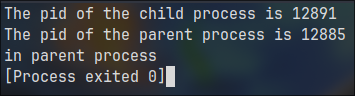
printf("in child process");

}

return 0;

}

**OUTPUT:**

****

**References:**

<https://www.geeksforgeeks.org/fork-system-call/>

<https://www.geeksforgeeks.org/getppid-getpid-linux/>

<https://www.geeksforgeeks.org/wait-system-call-c/>

<https://www.geeksforgeeks.org/zombie-and-orphan-processes-in-c/>

<https://www.geeksforgeeks.org/input-output-system-calls-c-create-open-close-read-write/>

| **On time Submission(2)** | **Knowledge of Topic(4)** | **Implementation and Demonstraion(4)** | **Total (10)** |
| --- | --- | --- | --- |
|  |  |  |  |
| **Signature of Faculty** |  | **Date of Submission** |  |