INFORME DE LABORATORIO

INFORMACIÓN BÁSICA					
ASIGNATURA:	Laboratorio de Programación de Sistemas				
TÍTULO DE LA PRÁCTICA:	Procesos y Señales				
NÚMERO DE PRÁCTICA:	Guía 8	**AÑO LECTIVO:**	2023	**NRO. SEMESTRE:**	**V**
FECHA DE PRESENTACIÓN	**21/07/2023**	**HORA DE PRESENTACIÓN**	*14*		
INTEGRANTE (s) *Calcina Puma Esteven Antonio* *Galvez Quilla Henry Isaias* *Paredes Quispe Jose Andre*				**NOTA (0-20)**	
DOCENTE(s): **Mg. Edith Giovanna Cano Mamani**					

RESULTADOS Y PRUEBAS

I. EJERCICIOS RESUELTOS:

EJERCICIO 1.

Crear un programa en C++ donde utilizando una llamada a sistema fork() se puedan crear tres hijos.

```
1   #include <stdio.h>
  1 #include<sys/types.h>
  2 #include<unistd.h>
  3 int main(){
  4   fork();
  5   fork();
  6   printf("Hola  Mundo\n");
  7   return 0;
  8 }
```

Aquí tenemos un proceso padre y tres procesos hijos.

```
stv@stv-Satellite-P55t-A:~/PS/lab08$ gcc ej1.c -o ej1c
stv@stv-Satellite-P55t-A:~/PS/lab08$ ./ej1c
Hola  Mundo
Hola  Mundo
Hola  Mundo
Hola  Mundo
stv@stv-Satellite-P55t-A:~/PS/lab08$
```

EJERCICIO 2.

Crear un programa en C++ donde utilizando la biblioteca csignal se pueda imprimir un mensaje de “hola” utilizando la señal SIGABRT

```
17 #include <iostream>
16 #include <csignal>
15
14 void signalHandler(int signal) {
13     if (signal == SIGABRT) {
12         std::cout << "Hola desde la señal SIGABRT" << std::endl;
11     }
10 }
 9
 8 int main() {
 7     // Asignar el manejador de señales
 6     std::signal(SIGABRT, signalHandler);
 5
 4     // Generar una señal SIGABRT
 3     std::abort();
 2
 1     return 0;
 }
```

```
usuario@usuario-ThinkCentre-M93p:~$ ./ej
Hola desde la señal SIGABRT
Abortado (`core' generado)
usuario@usuario-ThinkCentre-M93p:~$
```

EJERCICIO 3.

Modifica el código10.c para que mediante la utilización de la función “sleep()”, la frecuencia a la que el proceso hijo escribe en los ficheros sea menor que la del proceso padre. Es decir que realice menos escrituras por unidad de tiempo.

```c
// codigo10.c modificadoS
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>

int main() {
        int i;
        int fd1, fd2;
        const char string1[10]= "*********";
        const char string2[10]= "---------";
        pid_t rf;
        fd1 = creat("ficheroA", 0666);
        fd2 = creat("ficheroB", 0666);
        rf = fork();    /* Crea el proceso hijo */
        switch (rf) {
                case -1:
                        printf("\nNo he podido crear el proceso hijo");
                        break;
                case 0: /* Ejecuta el proceso hijo */
                        for (i = 0; i < 10; i++) { /* Escribe ---------  */
                                write(fd1, string2, sizeof(string2));
                                write(fd2, string2, sizeof(string2));
                                sleep(5); /* Abandonamos voluntariamente el procesador */
                        }
                        break;
                default: /* Ejecuta el proceso padre */
                         for (i = 0; i < 10; i++) { /* Escribe *********  */
                                 write(fd1, string1, sizeof(string1));
                                 write(fd2, string1, sizeof(string1));
                                 sleep(1); /* Abandonamos voluntariamente el procesador */
                         }
        }
        printf("\nFinal de ejecucion de %d \n", getpid());
        exit(0);
}
```

```
┌──(andr3㉿kali)-[~/Documents/PS]
└─$ gcc -Wall -o codigo10 codigo10.c

┌──(andr3㉿kali)-[~/Documents/PS]
└─$ ./codigo10

Final de ejecucion de 1505

┌──(andr3㉿kali)-[~/Documents/PS]
└─$ ps
    PID TTY          TIME CMD
   1066 pts/0    00:00:00 bash
   1506 pts/0    00:00:00 codigo10
   1514 pts/0    00:00:00 ps

┌──(andr3㉿kali)-[~/Documents/PS]
└─$
Final de ejecucion de 1506
```

```
┌──(andr3㉿kali)-[~/Documents/PS]
└─$ cat ficheroA
*********----------**********
********
┌──(andr3㉿kali)-[~/Documents/PS]
└─$ cat ficheroA
*********----------**********
********----------**********
**********----------
┌──(andr3㉿kali)-[~/Documents/PS]
└─$ cat ficheroA
*********----------**********
********----------**********
**********----------
----------
┌──(andr3㉿kali)-[~/Documents/PS]
└─$ cat ficheroA
*********----------**********
********----------**********
**********----------
----------
```

EJERCICIO 4.

Crear un programa en C++ o C que permita comprobar que el proceso nieto se ejecuta antes que el proceso hijo y el proceso hijo se ejecuta antes que el proceso padre

```cpp
#include<iostream>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
using namespace std;

int main(){
  pid_t hijo =fork();
  if(hijo==0){
    pid_t nieto = fork();
    if(nieto==0){
      cout<<"Proceso nieto"<<endl;
    }
    else{
      wait(NULL);
      cout<<"termino Proceso nieto"<<endl;
      cout<<"Proceso hijo"<<endl;
    }
  }
  else{
    wait(NULL);
    cout<<"termino proceso hijo"<<endl;
    cout<<"El proceso padre"<<endl;
  }
  return 0;
}
```

```
stv@stv--pc:[~/PS/lab08]$./ej4
Proceso nieto
termino Proceso nieto
Proceso hijo
termino proceso hijo
El proceso padre
```

II. PRUEBAS

¿Con que valores comprobaste que tu práctica estuviera correcta? ¿Qué resultado esperabas obtener para cada valor de entrada? ¿Qué valor o comportamiento obtuviste para cada valor de entrada?

Cada ejercicio se probó con los ficheros generados y se comprobaba con las indicaciones de cada uno de los ejercicios

III. CUESTIONARIO:

1) ¿Cómo usted definiría la bomba fork?

La "bomba fork" es una técnica utilizada en programación para generar una gran cantidad de procesos o bifurcaciones de procesos en un corto período de tiempo. Esta técnica puede ser utilizada para crear una carga excesiva en el sistema, consumir recursos o causar una denegación de servicio.

2) ¿Para qué sirve la señal SIGXCPU?

La señal SIGXCPU es una señal de tiempo de CPU (CPU time limit exceeded) que se envía a un proceso cuando ha excedido su límite de tiempo de CPU asignado. En sistemas Unix, los procesos pueden estar sujetos a límites de tiempo de CPU para evitar que utilicen recursos del sistema de manera excesiva o maliciosa.

CONCLUSIONES

Procesos:

Los procesos son entidades fundamentales en sistemas operativos que permiten la ejecución de programas de manera concurrente e independiente.
En C y C++, la creación de procesos se puede lograr mediante llamadas al sistema como fork() en sistemas Unix.

Los procesos creados mediante fork() son copias exactas del proceso original, pero tienen diferentes identificadores de proceso (PID).
Cada proceso tiene su propio espacio de memoria y recursos, lo que permite la ejecución paralela de programas y evita conflictos de memoria.
La comunicación entre procesos puede lograrse a través de mecanismos como pipes, colas de mensajes o memoria compartida.
Señales:

Las señales son mecanismos que permiten a los procesos recibir notificaciones asíncronas sobre eventos específicos.
En C y C++, se pueden manejar señales usando la función signal() o estableciendo un manejador de señales.
Algunas señales comunes incluyen SIGINT (generada por Ctrl+C en la terminal) y SIGTERM (señal de terminación).
La recepción de una señal puede alterar el flujo de ejecución del programa, como finalizarlo o realizar ciertas acciones antes de finalizar.
El manejo adecuado de señales es importante para evitar comportamientos inesperados o para liberar recursos adecuadamente antes de finalizar un programa.
Consideraciones adicionales:

Al trabajar con procesos y señales, es fundamental garantizar la sincronización y la gestión adecuada de los recursos compartidos para evitar condiciones de carrera y bloqueos.
Es importante manejar errores y excepciones correctamente al trabajar con procesos y señales para asegurar un comportamiento robusto y predecible del programa.
Al programar en entornos multiplataforma, ten en cuenta que las llamadas al sistema y el manejo de señales pueden variar entre sistemas operativos. Utiliza bibliotecas y abstracciones adecuadas para garantizar la portabilidad del código.

METODOLOGÍA DE TRABAJO

Se siguieron los siguientes pasos:

1. Lectura del enunciado.
2. Análisis del ejercicio.
3. Propuesta de solución.
4. Diseño de la solución.
5. Propuesta de casos de prueba.
6. Validación del diseño con los casos de prueba.
7. Implementación de la solución.
8. Pruebas unitarias.
9. Validación de la solución.
10.Refactorización en funciones atómicas.

REFERENCIAS Y BIBLIOGRAFÍA

[1] P.J. Deitel and H.M. Deitel, "Cómo Programar en C++", México, Ed. Pearson Educación, 2009
[2] B. Stroustrup, "El Lenguaje de Programación C++", Madrid, Adisson Pearson Educación, 2002
[3] B. Eckel, "Thinking in C++", Prentice Hall, 2000