Universidad De Magallanes

Facultad De Ingeniería

Departamento De Ingeniería En Computación

Laboratorio De Introducción a Sistemas Operativos

Divide y. . . ¿No siempre vencerás?

Práctica No° 2

Estudiantes:

Milton Hernández M.- milherna@umag.cl

Ayrton Morrison R.- aymorris@umag.cl

Fecha y hora: 16 de Octubre del 2024- 8:00 AM

Profesor: Eduardo Peña J.

Asignatura: Introducción a Sistemas Operativos

Status del documento

Práctica N°2 **Divide y. . . ¿No siempre vencerás?**			
Número de referencia del documento			2
Versión	**Revisión**	**Fecha**	**Razones del cambio**
1	1	16 de Octubre del 2024	Primera revisión del documento

Registro de cambios del documento		**RCD N°:**	1
		Fecha:	16 de Octubre del 2024
		Otiginado por:	Milton Hernández M. y Ayrton Morrison R.
		Aprobado por:	Milton Hernández M. y Ayrton Morrison R.
Titulo del documento:		Práctica N°2 Divide y…¿No siempre vencerás?	
Número de referencia del documento:			002
Versión del documento	1	**Número de revisión:**	1
Página	**Párrafo**	**Razones del cambio**	

Índice

Resumen

La gestión y manejo de procesos dentro de un sistema operativo corresponden a una parte de vital importancia dentro de este. Cuando un proceso esta en ejecución este es capaz de duplicarse y crear otros procesos 'hijos' lo que permite alivianar la carga de un proceso y aprovechar de mejor manera el 'pseudoparalelismo' de los sistemas operativos. De esto surge la pregunta, **¿Es posible beneficiarse de esta capacidad para la resolución de problemas asociados a los procesos?**, pregunta a la que se le dará respuesta en este escrito.

1. Introducción

Un proceso se define como un programa en ejecución dentro de un sistema operativo. Los procesos conforman la mayoría de las acciones que se realizan dentro de un sistema operativo, desde la las entradas y salidas del mismo, la creación o eliminación de archivos y carpetas, hasta la ejecución de programas más complejos.

Sin embargo trabajar unicamente con procesos de manera independiente no siempre es lo mas optimo; es por esto que se ha desarrollado la manera de desempeñar procesos a traves de la **interconexión** entre estos.

Existen multiples maneras en las que se puede lograr lo anterior, sin embargo la manera que se desarrollara y profundizara en el presente escrito sera mediante la **duplicación y creación de procesos**. Al duplicarse un proceso, este empieza un "Árbol genealógico" de procesos, este es llamado asi debido a que el proceso creador procede a conocerse como el proceso padre y el proceso creado como el proceso hijo, y en caso de que este ultimo llegara a crear otro proceso, el proceso padre pasara a ser el proceso abuelo y asi. Piense en esto como un árbol con ramificaciones donde cada nodo representa el proceso raíz o proceso padre de los que le proceden.

Para lograr aprovechar estos procesos en la resolución de una tarea es importante mencionar las funciones de espera entre procesos padres e hijos, los cuales son demasiado influyentes, ya que como se apreciara mas adelante, harán la diferencia entre conseguir un resultado deseado o no.

Por ultimo pero no menos importante, a la hora de trabajar con múltiples procesos se producen **cambios de contexto**, lo cuál quiere decir que el sistema operativo se encarga de cambiar la ejecución de un proceso por otro, lo que implica el almacenamiento del estado actual del proceso en procesamiento para poder continuar posteriormente. Este **cambio de contexto** representa una carga en los recursos del S.O que no es menor.

Todo lo anterior de seguro genera multiples incógnitas a las que se buscará dar una respuesta en este informe.

2. Objetivo Principal

Comprender el proceso de trabajo con procesos familiares dentro de un Sistema Operativo.

3. Objetivos secundarios

1. Hacer correcto uso de la duplicación de procesos para generar programas que trabajaen de manera independiente.
2. Manejar conceptos básicos de comunicación entre procesos, permitiendo que un proceso espere au que sus hijos finalicen subsection para continuar su ejecución.
3. Comparar el comportamiento y eficiencia de programas secuenciales y programas paralelos.

4. Marco Teórico

DUrante la realización del presente informe de laboratorio se abordan diversos conceptos que es necesario tener claros a la hora de su comprensión.

4.1. Procesos

Como indicado anteriormente se le llama proceso a todo aquel programa que se encuentra en ejecución dentro de un sistema operativo.

Los procesos tienen algunas caracteristicas que vale la pena mencionar:

- **PID**: Identificador único del proceso.
- **PPID**: Identificador único del proceso padre.
- **Memoria del proceso**: La memoria que ocupa el proceso, es decir, la memoria que el proceso está usando (para almacenar variables, datos, etc.).
- **Punteros a sus hijos**: Punteros a los procesos hijos del proceso actual.

Esta información se puede encontrar en el **Bloque de control del proceso** (PCB) que es una estructura de datos que contiene toda la información necesaria para el manejo de un proceso.

4.2. Jerarquía de procesos

Es posible familiarizar a los procesos de acuerdo con el nivel de creación que tienen entre ellos conservando las siguientes reglas:

1. El proceso que crea a otro es el **padre** del creado.
2. El proceso creado es el **hijo** del proceso creador.
3. Es sólo necesario **un padre** para crear un **hijo**.
4. Cada hijo tiene **un solo padre** y un padre puede tener **varios hijos**.

Conocer la jerarquía de los procesos permite comprender de mejor manera los próximos pasos a realizar, relacionados con la creación de un proceso por parte de otro proceso.

4.2.1. Árboles genealógicos

Si un proceso *A* crea un proceso *B* y a su vez el proceso *B* crea a un proceso *C*, entonces tenemos las siguientes relaciones:

- *A* es el **padre** de *B* y *B* es el **padre** de *C*.
- *B* es el **hijo** de *A* y *C* es el **hijo** de *B*.
- *A* es el **abuelo** de *C* y *C* es el **nieto** de *A*.

4.3. Duplicación de un proceso

dentro de un sistema operativo es posible duplicar un proceso, es decir, crear un proceso que sea el mismo que el original, pero con un identificador único (PID) diferente.

Cuando un proceso es duplicado se genera un proceso Hijo de este, que tiene su información duplicada, es decir que comparte los nombres de las variables, los valores de las mismas, conoce la línea de ejecución que llevaba el padre y comienza su ejecución desde este mismo punto. Cabe entonces preguntarse ¿Compartirán estos procesos la memoria en el sistema? esta será una de las preguntas que serán analizadas posteriormente.

Este comportamiento puede ser útil a la hora de crear un programa puesto que al existir dos procesos en el sistema operativo el procesador los atenderá de forma “pseudoparalela” (como si fueran procesos diferentes) logrando aumentar la eficiencia de la CPU y trabajando en más operaciones al mismo tiempo, lo que puede aumentar el rendimiento de un proceso.

5. Procedimiento experimental

Para el presente laboratorio fueron planteadas por el profesor de asignatura las siguientes tareas:

1. Realizar un programa *hijo_padre_abuelo* que muestre a través de sus ID la creación de ellos
2. Ingresar por paso de parámetros tres valores enteros, indicar al padre que duplique el primer valor, al abuelo que eleve a la potencia 3 el segundo valor y que el hijo obtenga la raíz del tercer valor
3. Dado un conjunto de valores del tipo a, b, c obtener las raíces de una ecuación cuadrática cuyos factores son a,b,c. Resolver usando 'fork()'

Para la realización de las anteriores programas se hace uso del lenguaje `C` que con librerías como `sys/types.h` y `unistd.h` se puede interactuar con el sistema operativo. En particular nos dan acceso a las siguientes herramientas:

1. `fork()` que permite duplicar un proceso. Esta función luego de generar el proceso hijo tiene un retorno para ambos procesos, el padre recibe por retorno el PID del hijo y el hijo recibe 0 como retorno. Esto permite manejar la lógica para ambos procesos dentro del mismo código.
2. `pid_t` es el tipo de dato que representa el PID de un proceso.
3. `getpid()` que permite obtener el PID del proceso actual.
4. `getppid()` que permite obtener el PID del proceso padre.
5. `wait()` que permite esperar a que un proceso hijo termine su ejecución. Se puede pasar como parámetro el PID del proceso hijo a esperar, pero si se pasa `NULL` se esperará hasta que cualquiera de los procesos hijos termine su ejecución.
6. `exec()` que permite ejecutar un programa en el sistema operativo. Esta función aunque no fué usada en la presente práctica es llamativa de mencionar puesto que permite convertir un proceso en otro, pudiendo reverenciarlo por su PID; Esta función es una manera más adecuada de ejecutar funciones de sistema en contraposición con `system()` de la librería `stdlib.h` puesto que, de acuerdo con Tanenbaum y Woodhull (1997) puede conllevar a ciertos problemas de seguridad.

Con todas las herramientas anteriores se puede proceder a realizar las tareas que se plantean en el presente laboratorio.

6. Resultados

6.1. Programa 1: Hijo Padre Abuelo

El primer programa se codifico con la intención de observar e interactuar por primera vez con las funciones `fork()`, `getpid()` y `getppid()`. Como se puede observar en el anexo 2, el código no presenta mayor complejidad que realizar un orden correcto de duplicación de programas, al ejecutar el programa se observó como en la mayoría de los casos los datos entregados por pantalla correspondían a la salida esperada, la cual era que los programas padres estén relacionados con los programas hijos y viceversa.

Sin embargo al compilar y ejecutar este código en WSL[1] se obtuvo, en algunas ocasiones un comportamiento inesperado como vemos en el código 1.

Listing 1: Programa Hijo Padre Abuelo en WSL

```
$ ./programa1.out
Soy el padre; PID 100; PID padre 75
Soy el hijo; PID 101; PID padre 74
Soy el nieto; PID 102; PID padre 101
$ ./programa1.out
Soy el padre; PID 100; PID padre 75
Soy el hijo; PID 103; PID padre 100
Soy el nieto; PID 104; PID padre 105
```

Lo que ocurre es que el proceso hijo parece no ser creado por el proceso padre sino por un proceso que WSL denomina `Relay(75)`, que, por lo que se logró investigar es un tipo de proceso que genera WSL para obtener un mejor rendimiento al usar el entorno Linux dentro de Windows. Sin embargo este entorno no generó dificultades a la hora de realizar los demás ejercicios.

6.2. Programa 2: Uso de hijos en C

Este programa fue creado de forma no muy extensa como se muestra en el anexo 4. A la hora de realizarlo no se tuvo complicaciones con los valores de las variables. Sin embargo surgió la duda de ¿qué ocurriría si todo se trabajara con un único parámetro pasado por el usuario, idea que se explora en el anexo 3.

En este punto se observa un comportamiento interesante, ocurre que los valores de la variable modificada no se solapan, es decir que siempre se obtienen los mismos resultados, esto sugiere que al momento de ejecutar `fork()` el proceso hijo conoce los valores de las variables pero no las comparte con el proceso padre ni con sus hermanos.

[1] Windows Subsystem for Linux, es una herramienta proporcionada por Microsoft dentro de sistemas Windows que permite correr un entorno Linux sobre Windows

6.3. Programa 3: Resolución de una ecuación cuadrática con multiples procesos

Este programa es el que mas tiempo de creación tomó, no precisamente por la complejidad de lo requerido, mas bien por las dudas que surgieron durante la creación de este.

La interrogante que surgió en primera instancia fue; **¿Es posible compartir recursos entre procesos duplicados?**.

La dificultad para resolver esta duda radica principalmente en las herramientas y recursos limitados con los que se contaba en ese momento, lo que llevo a multiples acercamientos y formas de intentar resolver este problema.

El objetivo esencial era calcular las soluciones de la ecuación $ax^2 + bx + c = 0$ mediante la fórmula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Pero la implementación para este problema varió de diversas maneras.

La primera solución,como se puede observar en los anexos 5 y 8, fue utilizar archivos para comunicar entre sí a la familia de procesos. Las dos ideas puestas a cabo fueron:

1. Generar dos hijos encargados de calcular $-b$ y $2a$ respectivamente para posteriormente que el padre calculara $\sqrt{b^2 - 4ac}$ y las soluciones de la ecuación.
2. Generar tres hijos, encargados de calcular $-b$, $2a$ y $\sqrt{b^2 - 4ac}$ respectivamente, el padre calculara las soluciones de la ecuación una vez que los hijos hubieran terminado su ejecución.

Con estos programas se logro solucionar el problema de la distribución de recursos. Para comprobar que los programas funcionaban correctamente se utilizo una lista de ecuaciones entregada por el docente Eduardo Peña J.. Sin embargo, para la correcta utilización de las demás soluciones planteadas se optó por crear un programa adicional (Que puede encontrar en el anexo 11) que permite generar triadas de números aleatorios para luego ser traspasados a un archivo binario con el nombre *resorces.txt*, la creación de este archivo permite que sea más sencilla la lectura de muchas ecuaciones y el procesamiento por medio de los diferentes programas utilizados.

Posteriormente se realizó otro programa el cual era capaz de resolver ecuaciones cuadráticas secuencialmente, (el cuál se puede encontrar en el anexo 7), con el fin de comparar el tiempo de ejecución y la eficiencia de los distintos métodos utilizados. Al contrastar los datos relacionados al tiempo obtenidos, se obtuvo que los programas multiprocedurales tomaron una cantidad mucho mayor de tiempo que el programa secuencial, llegando a variar hasta en un 2200 %.

Dos soluciones más fueron llevadas a cabo, las cuales buscaban experimentar con la eficiencia de la duplicación de procesos con el fin de generar mejores conclusiones al respecto. Primeramente (como se puede observar en el anexo 9) se creó un programa que generara un hijo por cada una de las ecuaciones presentes en el archivo de recursos (de ahí la importancia que fuera binario para su fácil lectura). Es importante destacar que se encontró un límite en 30960 procesos a partir del cuál no se generan más procesos hijos. Además los tiempos de ejecución continúan siendo mayores al caso secuencial.

Finalmente se llegó a una última solución basada en la memoria compartida, sin embargo para realizar esto no es tan sencillo como compartir un puntero en el mismo programa, por lo que se tuvo que utilizar dos bibliotecas; `sys/ipc.h` y `sys/shm.h`, bibliotecas las cuales proporcionaron funciones que permitieron compartir memoria entre los diversos procesos. A pesar de que se evito el uso de archivos, el resultado obtenido también fue mas lento que el programa secuencial.

7. Análisis y discusión de los resultados

A partir de todas las pruebas realizadas anteriormente se pueden sacar en claro algunas características de la duplicación de procesos y su comportamiento.

Lo que resalta a simple vista de los resultados obtenidos es el hecho que los programas creados mediante el uso de `fork()` tuvieron considerablemente menos eficiencia que el programa secuencial (siendo la más destacable aquella versión que crea un proceso por cada una de las ecuaciones presentes en el archivo de recursos).

La razón de lo anterior es bien predecible, resulta ser que la creación de procesos duplicados mediante `fork()` requiere una importante carga para el sistema operativo, puesto que implica un cambio de contexto, la creación de un PCB, la duplicación de memoria entre otros pasos importantes, lo que en el caso de los programas secuenciales no es necesario, por lo que es más eficiente.

En el caso del programa mostrado en los anexos 5 y 8 su ineficacia, además de deberse al uso de múltiples `fork()` para cada ecuación, se debe a que el programa genera muchas aperturas y comprobaciones de archivos, lo cual es un proceso costoso en cuanto a tiempo de ejecución.

Por otro lado, la razón de la mejor funcionalidad del programa que genera un hijo por proceso se debe al poder del "paralelismo", que permite que las *n* ecuaciones ingresadas sean trabajadas "simultáneamente" generando un tiempo de respuesta un poco más rápido que otras implementaciones.

Otro análisis destacable es el hecho de que los procesos duplicados mediante `fork()` no comparten memoria entre sí; para lograr conseguir este comportamiento es necesario utilizar la memoria compartida, lo cual es un problema que se aborda en el anexo 10. Sin embargo, este programa utiliza contenidos que exceden los propósitos del presente laboratorio. Empero a pesar dle uso de memoria compartida, el programa resulta ser ineficiente, esto debido a que se crean trs `fork()` para cada ecuación, lo cual, como se menciono anteriormente es tremendamente ineficiente.

8. Conclusiones y recomendaciones

A raíz de todo lo mencionado anteriormente, se puede llegar a la conclusion de que efectivamente se logró comprender y aplicar satisfactoriamente los conceptos relacionados a la duplicación y creación de procesos, así como también se comprendió de manera parcial la comunicación entre procesos mediante el uso de la función `wait()`.

Adicionalmente, es importante destacar las múltiples limitaciones con las que se cuenta en en relación a la comunicación entre procesos, esto debido a la falta de herramientas que permitan no solamente enviar información a procesos hijos mediante la duplicación sino, además, permitir que los procesos hijos puedan enviar información a su proceso padre, logrando tener procesos que, aún en paralelo consigan trabajar en conjunto.

De lograr conseguir herramientas adecuadas para llevar a cabo dicho objetivo podría generarse un programa con real eficiencia a la hora de ejecutarse, sin embargo, esto no es el objetivo de este trabajo, por lo que esta idea se postergará para una futura experiencia de laboratorio.

9. Anexos

Listing 2: Hijo Padre Abuelo

```c
/// @author Milton Hernandez Morales
/// @brief Generacion de procesos Padre-Hijo-Nieto con fork();
///
/// Realizar un programa [padre, hijo, nieto] que muestre a traves de sus ID la creacion de ellos.

#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>

int main()
{
    if(!fork()){
        if(!fork()){ // Proceso Nieto
            printf("Soy el nieto; PID %d; PID padre %d\n", getpid(), getppid());
        }
        else{ // Proceso Hijo
            printf("Soy el hijo; PID %d; PID padre %d\n", getpid(), getppid());
        }
    }
    else{ // Proceso principal - Padre
        printf("Soy el padre; PID %d; PID padre %d\n", getpid(), getppid());
    }

    return 0;
}
```

Listing 3: Uso de hijos en C Millton

```c
/// @author Milton Hernandez Morales
/// @brief Generacion de procesos Abuelo-Padre-Hijo con fork();
///
/// Ingresar por consola un valor entero, indicar al hijo que duplique esa informacion, al padre que eleve esa informacion a la potencia 3 y que el nieto obtenga la raiz de esa informacion.

#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<math.h>

int main(int argc, char *argv[])
```

```
{
    int num;
    double resultado;
    if(argv[argc-1]){
        num = atoi(argv[argc-1]);
        printf("Numero pasado por parametro: %d\n\n", num);
    }
    else{
        printf("Numero no ingresado, se usara 2\n\n");
        num = 2;
    }

    if(!fork()){
        if(!fork()){ // Proceso nieto
            resultado = sqrt(num);
            printf("Soy el nieto; PID %d\nEl numero %d en raiz es: %f\n\n", getpid(), num,
    resultado);
        }
        else{ // Proceso Padre
            resultado = 2*num;
            printf("Soy el hijo; PID %d\nEl numero %d duplicado es: %f\n\n", getpid(), num,
    resultado);
        }
    }
    else{ // Proceso principal - Abuelo
        resultado = num*num;
        printf("Soy el padre; PID %d\nEl numero %d elevado es: %f\n\n", getpid(), num,
    resultado);
    }

    return 0;
}
```

Listing 4: Uso de hijos en C por Ayrton

```
/// @author Ayrton Morrison R.
/// @brief Generacion de procesos Padre, hijo, nieto para operaciones aritmeticas.

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <unistd.h>

int main(int argc, char **argv){
    if(argc != 4){
        printf("Error, ingrese 3 numeros\n");
```

```
        return 0;
    }
    float firstNumber = atof(argv[1]);
    float secondNumber = atof(argv[2]);
    float thirdNumber = atof(argv[3]);
    if(!fork()){
        // PROCESO HIJO (SEGUNDO PROCESO)
        printf("Soy el proceso hijo y duplicare %.1f\n", firstNumber);
        printf("El resultado es %.2f\n", firstNumber * 2);
        if(!fork()){
            // PROCESO NIETO (ULTIMO PROCESO)
            printf("Soy el proceso nieto y sacare la raiz de %.1f\n", thirdNumber);
            if(thirdNumber < 0){
                printf("Error, no se puede sacar la raiz de un numero negativo\n");
                return 0;
            }
            printf("El resultado es %.2f\n", sqrt(thirdNumber));
        }
    }
    else{
        // PROCESO PADRE (PRIMER PROCESO)
        printf("Soy el proceso padre y elevare a la potencia de 3 %.1f\n", secondNumber);
        printf("El resultado es %.2f\n", pow(secondNumber, 3));
    }
    return 0;
}
```

Listing 5: Uso de hijos en C para una ecuacion cuadratica, por Ayrton

```
    /// @author Ayrton Morrison R.
    /// @brief Generacion de procesos Padre e hijos para la resolucion de una ecuacion
    cuadratica.

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <unistd.h>
#include<sys/wait.h>
#include<sys/types.h>

typedef struct imaginary{
    float real;
    float imaginary;
}imaginary;
float getDiscriminant(float a, float b, float c);

```

```c
int main(int argc, char **argv){
    if(argc > 4 || argc < 4){
        printf("Error, ingrese 3 numeros en el formato 'a b c', siendo estos coeficientes de una ecuacion cuadratica \n");
        return 0;
    }
    float a = atof(argv[1]);
    float b = atof(argv[2]);
    float c = atof(argv[3]);
    float part_1 = 0.0, part_2 = 0.0, part_3 = 0.0;
    pid_t pid1, pid2;
    if(a == 0){
        printf("Error, el coeficiente 'a' no puede ser 0\n");
        return 0;
    }
    pid1 = fork();
    if(!pid1){
        //PROCESO HIJO(SEGUNDO PROCESO)
        part_1 = 2 * a;
        FILE *file = fopen("answers.txt", "w");
        fprintf(file, "%.3f\n", part_1);
        fclose(file);
        pid2 = fork();
        if(!pid2){
            //PROCESO NIETO(ULTIMO PROCESO)
            part_2 = -b;
            FILE *file = fopen("answers.txt", "a");
            fprintf(file, "%.3f", part_2);
            fclose(file);
            exit(0);
        }
        else{
            wait(NULL);
            exit(0);
        }
    }
    else{
        //PROCESO PADRE(PRIMER PROCESO)
        part_3 = getDiscriminant(a, b, c);
        wait(NULL);
        wait(NULL);
        FILE *file = fopen("answers.txt", "r");
        fscanf(file, "%f\n", &part_1);
        fscanf(file, "%f\n", &part_2);
        fclose(file);
        //2 RESPUESTAS DISTINTAS Y REALES
```

```
        if (part_3 > 0){
            float x1 = (part_2 + sqrt(part_3)) / part_1;
            float x2 = (part_2 - sqrt(part_3)) / part_1;
            printf("Las respuestas son %.3f y %.3f\n", x1, x2);
        }
        // 2 RESPUESTAS IGUALES Y REALES
        else if(part_3 == 0){
            float x = part_2 / part_1;
            printf("La respuesta es %.3f\n", x);
        }
        // 2 RESPUESTAS DISTINTAS Y COMPLEJAS
        else if(part_3 < 0){
            imaginary x1, x2;
            x1.real = part_2 / part_1;
            x1.imaginary = sqrt(-part_3) / part_1;
            x2.real = part_2 / part_1;
            x2.imaginary = -sqrt(-part_3) / part_1;
            printf("Las respuestas son %.3f + %.3fi y %.3f - %.3fi\n", x1.real, x1.imaginary
    , x2.real, x2.imaginary);
        }
    }
}
float getDiscriminant(float a, float b, float c){
    return ( pow(b,2) - (4*a*c) );
}
```

Listing 6: TAD para números complejos, por Milton

```
#ifndef COMPLEX
#define COMPLEX
#include<stdio.h>
#include<math.h>

/// \struct complex
/// @brief Estructura para representar un numero complejo.
typedef struct complex{
    float real; ///< Parte real
    float imaginary; ///< Parte imaginaria
} complex;

complex initComplex(float real, float imaginary);
float mod(complex x);
complex conjugate(complex x);
complex sum(complex x, complex y);
complex dif(complex x, complex y);
complex mult(complex x, complex y);
```

```
complex divition(complex x, complex y);
complex inverse(complex x);
void printComplex(FILE* stream, complex x);
#endif
```

Listing 7: Ecuación cuadrática en forma secuencial, por Milton

```
/// @author Milton Hernandez Morales
/// @brief Lectura y solucion de ecuaciones cuadraticas en forma secuencial.
///
/// Dado un conjunto de valores del tipo {a,b,c} obtener las raices de una ecuacion
    cuadratica cuyos factores son a,b,c.

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "complex.h"

/// Colores para mejor apreciacion de los resultados
#define RED "\x1b[31m"
#define GREEN "\x1b[32m"
#define YELLOW "\x1b[33m"
#define BLUE "\x1b[34m"
#define MAGENTA "\x1b[35m"
#define CYAN "\x1b[36m"
#define WHITE "\x1b[37m"
#define RESET "\x1b[0m"

int main()
{
    float coefficient[3];
    complex part1 = initComplex(0,0); // corresponde al -b
    complex part2 = initComplex(0,0); // corresponde a la raiz
    complex part3 = initComplex(0,0); // corresponde al denominador
    float discriminant;
    complex solution1, solution2; // Soluciones de la ecuacion
    clock_t start, end;
    double elapsedTime;
    FILE *resourcesF;
    unsigned int cont = 0;
    unsigned int maxCont;

    if((resourcesF = fopen("./resources.txt", "r")) == NULL){
        printf("No se pudo abrir el archivo de recursos\n");
        exit(-1);
    }
```

```
    fseek(resourcesF, 0, SEEK_END);
    maxCont = ftell(resourcesF) / sizeof(float) / 3;
    fseek(resourcesF, 0, SEEK_SET);

    start = clock();
    for(cont=1; cont<=maxCont; cont++){
        coefficient[0]=0;
        coefficient[1]=0;
        coefficient[2]=0;
        part1 = initComplex(0,0); // corresponde al -b
        part2 = initComplex(0,0); // corresponde a la raiz
        part3 = initComplex(0,0); // corresponde al denominador


        if(fread(&coefficient, sizeof(float), 3, resourcesF) < 3){
            printf("No se pudo leer el archivo de recursos\n");
            exit(-1);
        }

        printf("%d - Los coeficientes ingresados son: %.2f, %.2f, %.2f\n", cont, coefficient
    [0], coefficient[1], coefficient[2]);

        if(coefficient[0] == 0){
            printf("La ecuacion no corresponde a una ecuacion cuadratica\n");
        }
        part1.real = -coefficient[1]; // -b

        discriminant = coefficient[1]*coefficient[1] - 4*coefficient[0]*coefficient[2]; // b
    ^2 - 4ac

        if(discriminant >= 0){ // Caso real
            part2.real = sqrt(discriminant); // sqrt(b^2 - 4ac)
        }
        else{ // Caso imaginario
            part2.imaginary = -sqrt(-discriminant); // sqrt(-(b^2 - 4ac))i
        }

        part3.real = 2*coefficient[0]; // 2a

        solution1 = sum(divition(part1, part3), divition(part2,part3));
        solution2 = dif(divition(part1, part3), divition(part2,part3));


        printf("x_1 = "BLUE);
        printComplex(stdout, solution1);
        printf(RESET"   ;   x_2 = "GREEN);
```

```
        printComplex(stdout, solution2);
        printf(RESET"\n");
    }

    end = clock();
    elapsedTime = ((double)(end - start) / CLOCKS_PER_SEC) * 1000;
    printf("Tiempo de ejecucion %f\n", elapsedTime);
    fclose(resourcesF);
    return 0;
}
```

Listing 8: Ecuación cuadrática en forma paralela, por Milton

```
/// @author Milton Hernandez Morales
/// @brief Lectura y solucion de ecuaciones cuadraticas en forma paralela.
///
/// Dado un conjunto de valores del tipo {a,b,c} obtener las raices de una ecuacion
    cuadratica cuyos factores son a,b,c usando fork().

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <time.h>
#include "complex.h"

/// Colores para mejor apreciacion de los resultados
#define RED "\x1b[31m"
#define GREEN "\x1b[32m"
#define YELLOW "\x1b[33m"
#define BLUE "\x1b[34m"
#define MAGENTA "\x1b[35m"
#define CYAN "\x1b[36m"
#define WHITE "\x1b[37m"
#define RESET "\x1b[0m"

int main()
{
    float coefficient[3];
    complex part1 = initComplex(0,0); // corresponde al -b
    complex part2 = initComplex(0,0); // corresponde a la raiz
    complex part3 = initComplex(0,0); // corresponde al denominador
    complex solution1, solution2; // Soluciones de la ecuacion
    clock_t start, end;
    double elapsedTime;
```

```
33      FILE *resourcesF, *part1F, *part2F, *part3F;
34      pid_t child1, child2, child3;
35      int cont = 0;
36      int maxCont = 0;
37
38      if((resourcesF = fopen("./resources.txt", "rb")) == NULL){
39          printf("No se pudo abrir el archivo de recursos\n");
40          exit(-1);
41      }
42      fseek(resourcesF, 0, SEEK_END);
43      maxCont = ftell(resourcesF) / sizeof(float) / 3;
44      rewind(resourcesF);
45
46      start = clock();
47      for(cont = 1; cont <= maxCont; cont++){
48          part1 = initComplex(0,0); // corresponde al -b
49          part2 = initComplex(0,0); // corresponde a la raiz
50          part3 = initComplex(0,0); // corresponde al denominador
51
52          if(fread(&coefficient, sizeof(float), 3, resourcesF) < 3){
53              printf("No se pudo leer el archivo de recursos\n");
54              exit(-1);
55          }
56
57          printf("%d - Los coeficientes ingresados son: %.2f, %.2f, %.2f\n",cont , coefficient
    [0], coefficient[1], coefficient[2]);
58
59          if(!coefficient[0]){
60              printf("La ecuacion no corresponde a una ecuacion cuadratica\n");
61          }
62
63          // Primer Fork, para el -b
64          child1 = fork();
65          if(!child1){ // Es el hijo porque fork devuelve 0
66              part1.real = -coefficient[1]; // -b
67              if((part1F = fopen("./part1", "wb")) == NULL){
68                  printf("No se pudo abrir el archivo auxiliar1\n");
69                  exit(-1);
70              }
71              fwrite(&part1.real, sizeof(float), 1, part1F);
72              fwrite(&part1.imaginary, sizeof(float), 1, part1F);
73              fclose(part1F);
74              return 0;
75          }
76
77          // segundo fork, para el \sqrt{b^2-4ac}
```

```
78          child2 = fork();
79          if(!child2){ // Es el hijo porque fork devuelve 0
80              float discriminant = coefficient[1]*coefficient[1] - 4*coefficient[0]*
    coefficient[2]; // b^2 - 4ac
81              if(discriminant >= 0){ // Caso real
82                  part2.real = sqrt(discriminant); // sqrt(b^2 - 4ac)
83              }
84              else{ // Caso imaginario
85                  part2.imaginary = -sqrt(-discriminant); // sqrt(-(b^2 - 4ac))i
86              }
87
88              if((part2F = fopen("./part2", "wb")) == NULL){
89                  printf("No se pudo abrir el archivo auxiliar2\n");
90                  exit(-1);
91              }
92
93              fwrite(&part2.real, sizeof(float), 1, part2F);
94              fwrite(&part2.imaginary, sizeof(float), 1, part2F);
95              fclose(part2F);
96              return 0;
97          }
98
99          // tercer fork, para el 2a
100         child3 = fork();
101         if(!child3){ // Es el hijo porque fork devuelve 0
102             if((part3F = fopen("./part3", "wb")) == NULL){
103                 printf("No se pudo abrir el archivo auxiliar3\n");
104                 exit(-1);
105             }
106
107             part3.real = 2*coefficient[0]; // 2a
108
109             fwrite(&part3.real, sizeof(float), 1, part3F);
110             fwrite(&part3.imaginary, sizeof(float), 1, part3F);
111             fclose(part3F);
112             return 0;
113         }
114
115         // Padre
116         wait(NULL); // Espera al primer hijo
117         wait(NULL); // Espera al segundo hijo
118         wait(NULL); // Espera al tercer hijo
119
120         // Leemos las partes
121         if((part1F = fopen("./part1", "rb")) == NULL){
122             printf("No se pudo abrir el archivo auxiliar1\n");
```

```
123             exit(-1);
124         }
125         if((part2F = fopen("./part2", "rb")) == NULL){
126             printf("No se pudo abrir el archivo auxiliar2\n");
127             exit(-1);
128         }
129         if((part3F = fopen("./part3", "rb")) == NULL){
130             printf("No se pudo abrir el archivo auxiliar3\n");
131             exit(-1);
132         }
133 
134         if(fread(&part1.real, sizeof(float), 1, part1F) != 1){
135             printf("No se pudo leer el archivo auxiliar1\n");
136             exit(-1);
137         }
138         if(fread(&part1.imaginary, sizeof(float), 1, part1F) != 1){
139             printf("No se pudo leer el archivo auxiliar1\n");
140             exit(-1);
141         }
142         fclose(part1F);
143 
144         if(fread(&part2.real, sizeof(float), 1, part2F) != 1){
145             printf("No se pudo leer el archivo auxiliar2\n");
146             exit(-1);
147         }
148         if(fread(&part2.imaginary, sizeof(float), 1, part2F) != 1){
149             printf("No se pudo leer el archivo auxiliar2\n");
150             exit(-1);
151         }
152         fclose(part2F);
153 
154         if(fread(&part3.real, sizeof(float), 1, part3F) != 1){
155             printf("No se pudo leer el archivo auxiliar3\n");
156             exit(-1);
157         }
158         if(fread(&part3.imaginary, sizeof(float), 1, part3F) != 1){
159             printf("No se pudo leer el archivo auxiliar3\n");
160             exit(-1);
161         }
162         fclose(part3F);
163 
164         solution1 = sum(divition(part1, part3), divition(part2,part3));
165         solution2 = dif(divition(part1, part3), divition(part2,part3));
166 
167         // Imprimir por pantalla
168         printf("x_1 = "BLUE);
```

```
169         printComplex(stdout, solution1);
170         printf(RESET"   ;   x_2 = "GREEN);
171         printComplex(stdout, solution2);
172         printf(RESET"\n");
173     }
174 
175     end = clock();
176     elapsedTime = ((double)(end - start) / CLOCKS_PER_SEC) * 1000;
177     printf("Tiempo de ejecucion %f\n", elapsedTime);
178     fclose(resourcesF);
179 
180     return 0;
181 }
```

Listing 9: Ecuación cuadrática con un hijo por proceso, por Milton

```
1  /// @author Milton Hernandez Morales
2  /// @brief Lectura y solucion de n ecuaciones cuadraticas en forma paralela con n hijos.
3  ///
4  /// En este programa se crea un hijo mediante fork() para cada una de las ecuaciones
       cuadraticas.
5  
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <time.h>
9  #include "complex.h"
10 #include <unistd.h>
11 #include "sys/types.h"
12 #include "sys/wait.h"
13 
14 /// Colores para mejor apreciacion de los resultados
15 #define RED "\x1b[31m"
16 #define GREEN "\x1b[32m"
17 #define YELLOW "\x1b[33m"
18 #define BLUE "\x1b[34m"
19 #define MAGENTA "\x1b[35m"
20 #define CYAN "\x1b[36m"
21 #define WHITE "\x1b[37m"
22 #define RESET "\x1b[0m"
23 
24 int main()
25 {
26     float coefficient[3];
27     complex part1 = initComplex(0,0); // corresponde al -b
28     complex part2 = initComplex(0,0); // corresponde a la raiz
29     complex part3 = initComplex(0,0); // corresponde al denominador
```

```
30      float discriminant;
31      complex solution1, solution2; // Soluciones de la ecuacion
32      clock_t start, end;
33      double elapsedTime;
34      FILE *resourcesF;
35      long int MAXHIJOS = 0;
36
37      if((resourcesF = fopen("./resources.txt", "rb")) == NULL){
38          printf("No se pudo abrir el archivo de recursos\n");
39          exit(-1);
40      }
41
42      part1 = initComplex(0,0); // corresponde al -b
43      part2 = initComplex(0,0); // corresponde a la raiz
44      part3 = initComplex(0,0); // corresponde al denominador
45
46      // Calcular la cantidad de hijos maxima
47      if((resourcesF = fopen("./resources.txt", "rb")) == NULL){
48          printf("No se pudo abrir el archivo de recursos\n");
49          exit(-1);
50      }
51      fseek(resourcesF, 0, SEEK_END);
52      MAXHIJOS = ftell(resourcesF) / sizeof(float) / 3;
53      fclose(resourcesF);
54
55      printf("Hijos maximos: %ld\n", MAXHIJOS);
56
57      start = clock();
58
59      for(long int i = 1; i <= MAXHIJOS; i++){
60          if(!fork()){ // Crear un hijo
61              if((resourcesF = fopen("./resources.txt", "rb")) == NULL){
62                  printf("No se pudo abrir el archivo de recursos\n");
63                  exit(-1);
64              }
65              fseek(resourcesF, (i-1) * sizeof(float) * 3, SEEK_SET); // Ir a la linea
66              if(fread(&coefficient, sizeof(float), 3, resourcesF)<3){ // Leer los
    coeficientes
67                  printf("No se pudo leer los coeficientes de la linea %ld\n", i);
68                  exit(-1);
69              }
70              fclose(resourcesF);
71
72              if(coefficient[0] == 0){
73                  printf("La ecuacion no corresponde a una ecuacion cuadratica\n");
74              }
```

```
            part1.real = -coefficient[1]; // -b

            discriminant = coefficient[1]*coefficient[1] - 4*coefficient[0]*coefficient[2];
    // b^2 - 4ac

            if(discriminant >= 0){ // Caso real
                part2.real = sqrt(discriminant); // sqrt(b^2 - 4ac)
            }
            else{ // Caso imaginario
                part2.imaginary = -sqrt(-discriminant); // sqrt(-(b^2 - 4ac))i
            }

            part3.real = 2*coefficient[0]; // 2a

            solution1 = sum(divition(part1, part3), divition(part2,part3));
            solution2 = dif(divition(part1, part3), divition(part2,part3));

            // Imprimir soluciones por pantalla
            printf("%ld - Coef: %.2f, %.2f, %.2f\t\t ", i, coefficient[0], coefficient[1],
    coefficient[2]);
            printf("x_1 = "BLUE);
            printComplex(stdout, solution1);
            printf(RESET"   ;   x_2 = "GREEN);
            printComplex(stdout, solution2);
            printf(RESET"\n");
            exit(0);
        }
    }

    printf("For terminado\n");
    for(long int i = 0; i < MAXHIJOS; i++){ // Esperamos a todos los hijos
        wait(NULL);
    }
    end = clock();
    elapsedTime = ((double)(end - start) / CLOCKS_PER_SEC) * 1000;
    printf("Tiempo de ejecucion %f\n", elapsedTime);

    return 0;
}
```

Listing 10: Ecuación cuadrática con memoria compartida, por Milton

```
/// @author Milton Hernandez Morales
/// @brief Lectura y solucion de n ecuaciones cuadraticas en forma paralela con memoria
    compartida.
///
```

```c
/// Dado un conjunto de valores del tipo {a,b,c} obtener las raices de una ecuacion cuadratica cuyos factores son a,b,c usando fork(), se usa <sys/shm.h> y <sys/ipc.h> para memoria compartida.

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <time.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdbool.h>
#include "complex.h"

/// Colores para mejor apreciacion de los resultados
#define RED "\x1b[31m"
#define GREEN "\x1b[32m"
#define YELLOW "\x1b[33m"
#define BLUE "\x1b[34m"
#define MAGENTA "\x1b[35m"
#define CYAN "\x1b[36m"
#define WHITE "\x1b[37m"
#define RESET "\x1b[0m"

int main()
{
    float coefficient[3];
    FILE *resourcesF;
    int cont = 0;
    int maxCont;
    clock_t start, end;
    double elapsedTime;

    // Crear un segmento de memoria compartida
    int shmid;
    complex *parts;
    complex solution1, solution2;
    key_t key = 1234; // Llave unica para la memoria compartida

    // Crear un segmento de memoria compartida de tamanio para 2 enteros
    if ((shmid = shmget(key, 3 * sizeof(complex), IPC_CREAT | 0666)) < 0)
    {
        perror("Error al crear la memoria compartida");
        exit(1);
    }
```

```
48
49      // Adjuntar la memoria compartida
50      if ((parts = (complex *)shmat(shmid, NULL, 0)) == (complex *)-1)
51      {
52          perror("Error al adjuntar la memoria compartida");
53          exit(1);
54      }
55
56      // Inicializamos memoria compartida
57      parts[0] = initComplex(0,0);
58      parts[1] = initComplex(0,0);
59      parts[2] = initComplex(0,0);
60
61      if((resourcesF = fopen("./resources.txt", "rb")) == NULL){
62          printf("No se pudo abrir el archivo de recursos\n");
63          exit(-1);
64      }
65
66      fseek(resourcesF, 0, SEEK_END);
67      maxCont = ftell(resourcesF) / sizeof(float) / 3;
68      fseek(resourcesF, 0, SEEK_SET);
69      printf("Cantidad de numeros: %d\n\n", maxCont);
70
71      start = clock();
72
73      for(cont = 1; cont <= maxCont; cont++)
74      {
75          if(fread(&coefficient, sizeof(float), 3, resourcesF)<3){ // Leer los coeficientes
76              printf("No se pudo leer los coeficientes de la ecuacion %d\n", cont);
77              exit(-1);
78          }
79          printf("%d - Los coeficientes ingresados son: %.2f, %.2f, %.2f\n",cont , coefficient
    [0], coefficient[1], coefficient[2]);
80
81          if(!fork()){ // Primer hijo, encargado de -b
82              parts[0].real = -coefficient[1]; // -b
83              exit(0);
84          }
85
86          if(!fork()){ // Segundo hijo, encargado de \sqrt{b^2-4ac}
87              float discriminant = coefficient[1]*coefficient[1] - 4*coefficient[0]*
    coefficient[2]; // b^2 - 4ac
88              if(discriminant >= 0){ // Caso real
89                  parts[1].real = sqrt(discriminant); // sqrt(b^2 - 4ac)
90              }
91              else{ // Caso imaginario
```

```
                parts[1].imaginary = -sqrt(-discriminant); // sqrt(-(b^2 - 4ac))i
            }
            exit(0);
        }

        if(!fork()){ // Tercer hijo, encargado de 2a
            parts[2].real = 2*coefficient[0]; // 2a
            exit(0);
        }

        wait(NULL); // Espera al primer hijo
        wait(NULL); // Espera al segundo hijo
        wait(NULL); // Espera al tercer hijo

        solution1 = sum(divition(parts[0], parts[2]), divition(parts[1],parts[2]));
        solution2 = dif(divition(parts[0], parts[2]), divition(parts[1],parts[2]));

        // Imprimir por pantalla
        printf("x_1 = "BLUE);
        printComplex(stdout, solution1);
        printf(RESET"   ;   x_2 = "GREEN);
        printComplex(stdout, solution2);
        printf(RESET"\n");
    }

    shmdt(parts); // Desconectar la memoria compartida
    shmctl(shmid, IPC_RMID, NULL); // Eliminar la memoria compartida

    end = clock();
    elapsedTime = ((double)(end - start) / CLOCKS_PER_SEC) * 1000;
    printf("Tiempo de ejecucion %f\n", elapsedTime);
    fclose(resourcesF);

    return 0;
}
```

Listing 11: Generador de números en formato binario

```
#include<stdlib.h>
#include<stdio.h>
#include<time.h>

int main(int argc, char* argv[]){
    srand(time(NULL));

    int triadas;
```

```
    float coeficientes[3];
    if(argc < 2){
        triadas = 10;
    }
    else{
        triadas = atoi(argv[1]);
    }
    printf("Se generaran %d triadas de numeros aleatorios:\n", triadas);

    FILE *res = fopen("./resources.txt", "wb");
    for(int i = 0; i < triadas; i++){
        for(int j = 0; j < 3; j++){
            coeficientes[j] = (float)((rand()%11)+1);
        }
        printf("%d %.2f %.2f %.2f\n", i+1, coeficientes[0], coeficientes[1], coeficientes
    [2]);
        fwrite(coeficientes, sizeof(float), 3, res);
    }
    fclose(res);
    return 0;
}
```

Referencias

Group, T. O. (2004). *fork() - create a new process*. Consultado el 12 de octubre de 2024, desde https://pubs.opengroup.org/onlinepubs/009696799/functions/fork.html

Tanenbaum, A. S., & Woodhull, A. S. (1997). *Sistemas Operativos: Diseño e implementación* (R. Escalona, Trad.; Segunda edición). Prentice Hall.