Taller Inicial Fork

Sistemas Operativos

Integrantes:

Daniel Felipe Castro Moreno

Daniel Horacio Gonzalez Orduz

Eliana Katherine Cepeda Gonzalez

Maria Paula Rodríguez Ruiz

Profesor:

John Corredor Franco

Dpto. de Ingeniería de Sistemas

Pontificia Universidad Javeriana

Bogotá D.C.

16 de Octubre del 2024

El objetivo de este taller es aplicar conceptos respecto a los procesos y la comunicación entre ellos, utilizando llamadas a sistema como fork() y pipe() en el lenguaje de programación C. Es así como el programa a desarrollar se encargará de recibir dos archivos que contienen arreglos de enteros, leerlos, realizar sumas parciales de los valores en procesos hijos y enviar estos resultados al proceso padre para su consolidación.

Ahora bien, es preciso conocer previo a su desarrollo un pequeño contexto acerca de los conceptos de los cuales se hace uso, es así como a continuación daremos un abrebocas respecto a la función fork(), pipe() y malloc():

- fork(): Es una llamada al sistema en Unix/Linux que se utiliza para crear un nuevo proceso, conocido como "hijo", que es una copia del proceso "padre" que aunque copia, gracias al bloque de control, cada proceso adquiere su propio id. La ejecución del proceso hijo comienza exactamente donde se llama a fork() en el código, pero con un espacio de memoria independiente y su uso esta principalmente dado en búsqueda de la ejecución en paralelo pudiendo el hijo ejecutar tareas diferentes a las de su padre. Es importante recalcar que el valor de retorno de fork() es crucial para identificar a los procesos:
    - Retorna 0 en el proceso hijo.
    - Retorna un PID positivo (el identificador del proceso hijo) en el proceso padre.
- pipe(): Se utiliza para crear un canal de comunicación unidireccional entre procesos. Permite que un proceso escriba en el extremo de escritura del pipe y otro proceso lea desde el extremo de lectura, lo que facilita la transferencia de datos entre procesos. A continuación, en la figura 1, un ejemplo respecto al uso de la función pipe() en una sola ejecución de la función fork():

```c
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int main() {
    int fd[2]; // Array para los descriptores del pipe (fd[0]: lectura, fd[1]: escritura)
    pipe(fd);  // Crear el pipe

    pid_t pid = fork();  // Crear un proceso hijo

    if (pid == -1) {
        perror("Error al crear el proceso hijo");
        return 1;
    }

    if (pid == 0) {
        // Proceso hijo: leer del pipe
        close(fd[1]);  // Cerrar el extremo de escritura
        char buffer[100];
        read(fd[0], buffer, sizeof(buffer));  // Leer datos del pipe
        printf("Proceso hijo leyó: %s\n", buffer);
        close(fd[0]);  // Cerrar el extremo de lectura
    } else {
        // Proceso padre: escribir en el pipe
        close(fd[0]);  // Cerrar el extremo de lectura
        char mensaje[] = "Hola desde el proceso padre!";
        write(fd[1], mensaje, strlen(mensaje) + 1);  // Escribir datos en el pipe
        close(fd[1]);  // Cerrar el extremo de escritura
        wait(NULL);    // Esperar a que el hijo termine
    }

    return 0;
}

```

Figura 1. Función pipe() en un llamado de la función fork(). Elaboración propia

- malloc(): Es una función de la biblioteca estándar de C que se utiliza para asignar memoria dinámica. En este programa, se usa para crear arreglos que almacenan los valores leídos de los archivos, adaptándose al tamaño necesario según los datos de entrada. A continuación, en la figura 2, un ejemplo de su uso donde n corresponde al tamaño total:

```c
// Asignación de memoria dinámica para el arreglo
*arreglo = (int*)malloc(*n * sizeof(int));
```

Figura 2. Función malloc(). Elaboración propia

Con referencia a lo previamente explicado, el programa a desarrollar recibe cuatro argumentos desde la línea de comandos: dos archivos con arreglos de enteros (archivo00 y archivo01) y dos enteros (N1 y N2) que indican la cantidad de elementos en cada archivo. Estos valores son procesados para leer los archivos y almacenar los arreglos en memoria dinámica utilizando malloc().

Seguido a esto, el programa realiza dos llamadas a fork() para crear un total de cuatro procesos, organizados de la siguiente manera:

- Padre: Proceso principal que coordina la comunicación entre todos los procesos e imprime la suma total.
- Primer Hijo: Calcula la suma de todos los elementos (los dos arreglos).
- Gran Hijo: Calcula la suma de todos los elementos del archivo00 (primer arreglo).
- Segundo Hijo: Calcula la suma de todos los elementos del archivo01 (segundo arreglo).

La decisión de utilizar solo dos llamadas a la funcion fork() esta dada principalmente por el análisis del siguiente fragmento de código, presentado en la figura 3, que ilustra el uso del fork():

```
void creacion_fork(int linea){
	pid_t proceso;
	proceso = fork();
	if(proceso<0){
		printf("Error creación proceso\n");
		exit(1);
	}else if (proceso==0){
		printf("%d: Proceso =HIJO=: %d\n",linea, getpid());
	}
	else{
		printf("%d: Proceso =PADRE= %d\n",linea, getpid());
	}
 }

int main(int argc, char *argv[]){
	int p = (int) atoi(argv[1]);
	for(int i=0; i<p; i++)
		creacion_fork(i);
	printf("\n");
	return 0;
}
```

Figura 3. Uso de la función fork(). Elaboración propia

En este código, cada llamada a fork() crea un nuevo proceso. La estructura del árbol resultante, presentado en la figura 4, demuestra cómo dos llamadas a fork() son suficientes para generar cuatro procesos: el proceso padre, el primer hijo, el segundo hijo y el gran hijo. En este árbol:

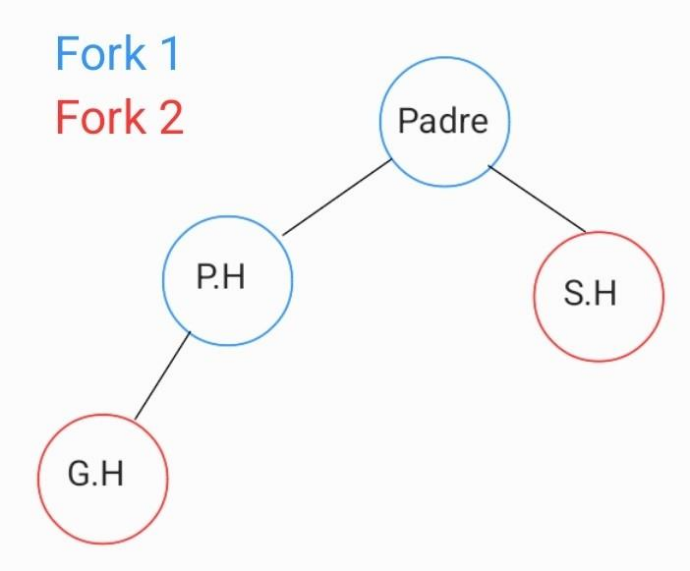

Figura 4. Arbol de procesos para dos llamados de la funcion fork(). Elaboración propia

- El proceso padre crea al primer hijo en la primera llamada a fork().
- Cada uno de estos procesos (padre e hijo) ejecuta una segunda llamada a fork(), lo que genera el segundo hijo y el gran hijo, respectivamente.
- La identificación de los procesos se logra utilizando las condiciones if y else, donde el proceso padre recibe un PID positivo y los hijos reciben un valor de retorno igual a cero como ya lo habíamos mencionado previamente.

Por otra parte, cada proceso hijo calcula la suma de los valores correspondientes a su archivo y envía el resultado al proceso padre mediante pipe(). El proceso padre, que coordina la recepción de estos datos, suma los resultados parciales y muestra el resultado final. Esto demuestra una coordinación efectiva entre procesos, donde pipe() actúa como el canal de comunicación que facilita la transferencia de información entre los procesos.

Es así como, en síntesis de lo anterior, la implementación del programa para el taller se puede desglosar en los siguientes pasos detallados:

1. **Recepción de Argumentos desde la Línea de Comandos**:
    - El programa recibe dos enteros N1 y N2 que indican el número de elementos de los arreglos que se leerán, y dos archivos archivo00 y archivo01 que contienen dichos arreglos de enteros.
    - La línea de ejecución toma la forma: ./taller_procesos N1 archivo00 N2 archivo01.
2. **Lectura de los Arreglos**:
    - Se emplea una función llamada leer_arreglo() para cargar los números de los archivos en arreglos de enteros con memoria dinámica.
    - Para cada archivo (archivo00 y archivo01), se reserva memoria con malloc() para N1 y N2 enteros respectivamente, y se lee el contenido de los archivos.
3. **Creación de Procesos**:
    - Se realizan **dos llamadas a fork()** para crear la estructura de procesos necesaria.
        - **Primera llamada a fork()**: genera dos procesos, el **proceso padre** y el **primer hijo**.
        - **Segunda llamada a fork()**: se ejecuta tanto en el **proceso padre** como en el **primer hijo**, resultando en la creación del **gran hijo** (desde el primer hijo) y el **segundo hijo** (desde el padre).

La estructura de procesos queda organizada de la siguiente forma, como se ilustra en la imagen proporcionada:

- El **Padre** (P)
- El **Primer Hijo** (PH)

- El **Gran Hijo** (GH) que desciende del Primer Hijo
- El **Segundo Hijo** (SH) que desciende del Padre

4. **Identificación de los Procesos**:
    - Para distinguir entre los procesos y asignarles tareas específicas, se utilizan los valores retornados por las dos llamadas a fork():
        - **Gran Hijo** (primerfork == 0 y segundofork == 0): Calcula la suma de los elementos del primer arreglo (archivo00) y envía su resultado al padre a través de un pipe.
        - **Segundo Hijo** (primerfork > 0 y segundofork == 0): Calcula la suma de los elementos del segundo arreglo (archivo01) y comunica su resultado al padre usando un pipe.
        - **Primer Hijo** (primerfork == 0 y segundofork > 0): Recibe los resultados parciales del Gran Hijo y del Segundo Hijo, los suma, y envía esta suma total al proceso Padre.
        - **Padre** (primerfork > 0 y segundofork > 0): Espera la finalización de los procesos hijo y gran hijo y recibe la suma consolidada del Primer Hijo.
5. **Comunicación entre Procesos**:
    - Se implementa un arreglo bidimensional de pipes para gestionar la comunicación de datos entre los procesos:
        - **Pipe 1**: Comunica el resultado del Gran Hijo al Primer Hijo.
        - **Pipe 2**: Envía el resultado del Segundo Hijo al Primer Hijo.
        - **Pipe 3**: Transfiere la suma total calculada por el Primer Hijo al Padre.
    - Cada proceso hijo escribe en su correspondiente pipe, y el proceso Padre lee de los pipes para combinar los resultados y mostrar el cálculo final.
6. **Liberación de Memoria**:

- Tras finalizar las operaciones, se utiliza free() para liberar la memoria previamente reservada con malloc(), asegurando que no existan fugas de memoria en la aplicación.

7. **Impresión del Resultado Final**:
    - El **proceso Padre** muestra la suma total de ambos arreglos después de recibir y procesar los resultados parciales enviados por los procesos hijos, concluyendo así el flujo de trabajo del programa.

Este enfoque permite una distribución eficiente de tareas entre los procesos y utiliza de manera precisa las funciones de fork() y pipe() para lograr una comunicación ordenada. El uso de solo dos llamadas a fork() para generar los cuatro procesos reduce la complejidad del código y facilita su control mediante la comparación de los valores retornados, garantizando que cada proceso realice la tarea que le corresponde, lo que en ultimas permite optimizar tanto el rendimiento como la claridad del programa.

Sin embargo, para facilitar aún más la gestión del código y su compilación, decidimos dividir las funcionalidades del programa en diferentes archivos y emplear un Makefile para simplificar el proceso de compilación. La estructura final del proyecto incluye:

1. **Archivos de Código y Encabezado**:
    - **taller3.c**: Contiene la lógica principal del programa, incluyendo la creación de los procesos mediante fork() y la coordinación de la comunicación entre ellos a través de pipes.
    - **funcs.c**: Agrupa funciones auxiliares, como:
        - leer_arreglo: Función que lee el contenido de los archivos archivo00 y archivo01 y los almacena en arreglos dinámicos.

        - suma: Función que realiza la suma de los elementos de un arreglo, utilizada tanto por el Gran Hijo como por el Segundo Hijo para enviar sus resultados parciales.
    - **taller3.h**: Encabezado que declara las funciones utilizadas en funcs.c y las estructuras necesarias para compartir entre los diferentes archivos .c. Esto asegura que tanto taller3.c como funcs.c puedan acceder a estas funciones de manera centralizada.
2. **Archivos de Entrada de Datos**:
    - **archivo00** y **archivo01**: Archivos de texto que contienen los arreglos de enteros a sumar. Estos archivos son leídos por el programa al inicio y sus contenidos se cargan en memoria para ser procesados por los diferentes procesos.
3. **Automatización de la Compilación con Makefile**:
    - Se creó un **Makefile** para simplificar la compilación y el enlazado de los archivos del taller:
        - El comando make permite compilar todos los archivos de manera automática, generando el ejecutable.
        - Esto elimina la necesidad de compilar manualmente cada archivo con gcc, reduciendo errores y agilizando el proceso de desarrollo.

Esta estructura modular no solo mejora la organización del código, sino que también facilita el mantenimiento y la ampliación del programa. Al separar la lógica principal de las funciones auxiliares, es más sencillo realizar pruebas y realizar modificaciones en funciones específicas sin afectar el resto del programa.

En conclusion programa ejemplifica cómo utilizar eficientemente las llamadas fork() y pipe() para manejar la creación de procesos y la comunicación entre ellos en un entorno de programación en C. A través de la creación de solo dos llamadas a fork(), se logra una estructura de procesos organizada en forma de árbol, lo que permite distribuir el trabajo entre los procesos y maximizar la eficiencia. Además, la implementación de pipe() para la transmisión de datos demuestra una adecuada sincronización y coordinación de tareas entre los procesos, haciendo que este ejemplo sea una referencia valiosa para el estudio de la programación concurrente en sistemas Unix/Linux.