Universidad Veracruzana

Facultad de Estadística e Informática

Sistemas Operativos

Proyecto Final

Comunicación entre Procesos

Nicole Moreno Hernández

Karol Quinto Guadalupe

13 de diciembre de 2023

Contenido

Introducción

El propósito del siguiente documento es exponer la solución desarrollada en código en el lenguaje C para abordar la problemática discutida en clase. La situación planteada involucra tres o más procesos que requieren ser gestionados por otro proceso central. Con el fin de facilitar la explicación, se empleará el término CPU para referirse a este último proceso.

La dificultad principal de la problemática radica en que el proceso CPU no tiene conocimiento de qué proceso solicitó la acción. En otras palabras, el CPU debe transformar la información proporcionada por el usuario, como por ejemplo multiplicar un número x por 10, pero sin conocer la identidad del proceso solicitante. Esta condición añade complejidad al sistema y simula un escenario similar al área de redes, donde la fuente de la petición debe ser comunicada a través de un proceso intermedio.

La solución implementada busca establecer una comunicación efectiva entre los procesos involucrados, permitiendo al CPU realizar las tareas solicitadas sin conocer directamente la fuente de la petición. Este enfoque mejora la eficiencia del sistema, facilitando su adaptabilidad a situaciones similares en el ámbito de la informática y las redes.

La implementación en lenguaje C aborda la complejidad del problema, permitiendo la ejecución coordinada de múltiples procesos y garantizando la correcta gestión de las peticiones sin comprometer la privacidad del solicitante, la manera que encontramos es utilizando pid, pipes y buffers, a continuación, se describirán un poco su funcionamiento.

Cabe recalcar que el proceso que decidimos que se haría era transformar una cadena dada por el usuario, de letras minúsculas a mayúsculas.

¿Qué son los pids, pipes y buffers?

Pid_t no tiene mucho que explicar, es un identificador de procesos de Linux, pid es por sus siglas en ingles Process ID.

En cuento a los pipes, se puede decir que son útiles para comunicar procesos, también son llamados tubos.

Para usar pipes desde C hay que llamar a la función pipe y pedirle los dos descriptores de archivo que serán los extremos del "tubo" que se habrá construido.
Ejemplo:

```
int p[2];
pipe(p);
```

Después de la invocación a pipe el arreglo de enteros p contiene en p[0] un descriptor de archivo para leer, y en p[1] un descriptor de archivo para escribir. Esto quiere decir que los pipes funcionan en una sola dirección.

El Buffer es un espacio de memoria en el que se almacenan datos evitando que el programa que los necesita se quede sin datos durante una transferencia.

Los datos son almacenados en un buffer mientras se transfieren desde un dispositivo de entrada o antes de enviarlos a un dispositivo de salida. También puede utilizarse para transferir datos entre procesos.

Código

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>
#include <ctype.h>

#define BUFFER_SIZE 256

void transCadena(char *cad) {
   while (*cad) {
      *cad = toupper((char)*cad);
      cad++;
   }
}

int main(int argc, char *argv[]) {
   int n;
   int pipe1[2], pipe2[2];

   printf("Ingrese el número de procesos a crear: ");
   scanf("%d", &n);
   getchar();

   for (int i = 0; i < n; ++i) {
```

```
pid_t pid_p2, pid_p3; //pid de las variables

char leerBuffer[BUFFER_SIZE];
char escribirBuffer[BUFFER_SIZE];

    // Crear el primer pipe
    pipe(pipe1);
    pipe(pipe2);

// Crear el proceso intermediario
pid_p2 = fork();

if (pid_p2 == 0) {
    // Código del intermediario

    // Leer la cadena del proceso original
    read(pipe1[0], leerBuffer, BUFFER_SIZE);
    printf("Parte 2 intermediario %d lee: %s\n", i+1, leerBuffer);

    // Pasar la cadena a mayúsculas
    transCadena(leerBuffer);

    // Enviar la cadena al proceso de conversión a mayúsculas
    write(pipe2[1], leerBuffer, strlen(leerBuffer) + 1);

    exit(EXIT_SUCCESS);
} else {
    // Código del proceso original

    // Pedir al usuario que ingrese una cadena
    printf("Ingrese una cadena\n");
    printf("Proceso %d : ", i+1);
    fgets(leerBuffer,BUFFER_SIZE, stdin);
    int lon = strlen(leerBuffer);
    leerBuffer[lon-1]=0;

    // Enviar la cadena al intermediario
    write(pipe1[1], leerBuffer, strlen(leerBuffer) + 1);

    // Leer la cadena convertida a mayúsculas
    read(pipe2[0], escribirBuffer, BUFFER_SIZE);
    printf("Proceso %d lee: %s\n", i+1, escribirBuffer);
}
```

```
    }

    return 0;
}
```

Explicación

Librerías

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

<stdio.h> Es función básica dentro del lenguaje C, pues es quien permite las estradas(scanf) y salidas (printf).
<stdlib.h> Permite utilizar la memoria dinámica, como free.
<unistd.h> Está relacionada con la creación de procesos por ejemplo fork().
<string.h> Se utilizó la manipulación de cadenas.
<ctype.h> Es la librería que permite transformar la cadena de letras minúsculas a mayúsculas.

```c
#define BUFFER_SIZE 256
```

Esta no es una librería, es una declaración del tamaño de un Buffer, que le decidimos poner un tamaño de 256.

TransCadena

```c
void transCadena(char *cad) {
    while (*cad) {
        *cad = toupper((char)*cad);
        cad++;
    }
}
```

En este caso, no hay mucho que explicar, pues es la función que recorre toda la cadena y convierte cada letra en mayúsculas, gracias al método toupper dado por la librería ctype.h.
Este método es el que podemos llamar nuestro proceso 3, es el proceso que recibe las instrucciones y devuelve la información, es nuestro “CPU” refiriéndonos al ejemplo de la introducción.

Creación de los pids y buffers

```c
int main(int argc, char *argv[]) {
    int n;
    int pipe1[2], pipe2[2];

    printf("Ingrese el número de procesos a crear: ");
    scanf("%d", &n);
    getchar();

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

        pid_t pid_p2, pid_p3; //pid de las variables

        char leerBuffer[BUFFER_SIZE];
        char escribirBuffer[BUFFER_SIZE];
```

Se presenta todo el maín con la declaración de las variables n, refiriéndose al número de procesos que se ejecutarán y pipe1, pipe2, ambos unidireccional, pipe 1 dirge todos los procesos hacía el proceso “Intermediario” que va a llamar a la función transCadena() y pipe2 se encargará de devolver la información.

Posteriormente tenemos que los pids que se inicializan, estos ya dentro del ciclo for, el número de iteraciones que hará el ciclo for es el mismo que el número de procesos que se pidió.

Pid_p2 es para el identificador del proceso que entró y el pid_p3 es la identificación del proceso que saldrá.

Tenemos los Buffers, dos, uno para la información de lectura y otro para la información de escritura, ambos del mismo tamaño como se mostró en el #define BUFFER_SIZE.
Es importante recalcar que getchar() es para que no haya ninguna confusión con la cadena, ya que si no se “borra” el espacio, puede existir un error de escritura.

Proceso hijo

```
    // Crear el primer pipe
    pipe(pipe1);
    pipe(pipe2);

    // Crear el proceso intermediario
    pid_p2 = fork();

    if (pid_p2 == 0) {
        // Código del intermediario

        // Leer la cadena del proceso original
        read(pipe1[0], leerBuffer, BUFFER_SIZE);
        printf("Parte 2 intermediario %d lee: %s\n", i+1, leerBuffer);

        // Pasar la cadena a mayúsculas
        transCadena(leerBuffer);

        // Enviar la cadena al proceso de conversión a mayúsculas
        write(pipe2[1], leerBuffer, strlen(leerBuffer) + 1);

        exit(EXIT_SUCCESS);
    } else {
```

En esta parte se puede ver como se crean ambos pipes, además que al primer proceso se le pone el identificados pid_p2 y el proceso se crea con la función fork().

Posteriormente, entra a un if, que pregunta sí es un proceso hijo, si es el caso, entonces leerá el proceso originar, por un lado, del tub, lo guardará en leerBuffer, con el tamaño de 256.

Se muestra la frase que el usuario ingreso y el número de proceso que es, de ahí se llama al método transCadena(), con el parámetro leerBuffer, para que la pueda transformar después por la otra parte del tuvo, envía la cadena que leyó el buffer y con strlen con parámetro de leerBuffer que indica la longitud de la cadena.

Y se le agrega exit(EXIT_SUCCESS) para indicar que ya se terminó el proceso.

Pedir cadena

```
            exit(EXIT_SUCCESS);
        } else {
            // Código del proceso original

            // Pedir al usuario que ingrese una cadena
            printf("Ingrese una cadena\n");
            printf("Proceso %d : ", i+1);
            fgets(leerBuffer,BUFFER_SIZE, stdin);
            int lon = strlen(leerBuffer);
            leerBuffer[lon-1]=0;

            // Enviar la cadena al intermediario
            write(pipe1[1], leerBuffer, strlen(leerBuffer) + 1);

            // Leer la cadena convertida a mayúsculas
            read(pipe2[0], escribirBuffer, BUFFER_SIZE);
            printf("Proceso %d lee: %s\n", i+1, escribirBuffer);
        }
    }

    return 0;
}
```

En esta parte, si no es un proceso hijo, entonces busca al proceso original, quien pide una cadena y lo lee con fgets, quien ingresa en la variable leerBuffer la información, con un tamaña de 256.

Las siguientes dos líneas, es para que no haya ningún problema con la lectura de cadenas con espacios.

Posteriormente por el lado izquierdo del peipe entra la información de leerBuffer y por último, se lee en el segundo tubo esa información, es decir la información ya procesada y la muestra en pantalla, junto con el id de proceso.

Conclusión:

El código fue un reto hacerlo, pues teníamos varías ideas, sin embargo, nos dimos cuenta que los tubos fue la mejor opción para hacerlo, la dificultad la encontramos en como un proceso se iba a comunicar con otro, con la necesidad de que pasará primeramente por un proceso intermediario, aunque, encontramos la solución haciendo dos pipes.
Para finalizar se puede decir, que los procesos generados era la parte, el if era la parte 2, pues era el intermediario quien decidía que hacer con la información de leerBuffer o escribirBuffer y por último la parte 3 quien era el método transCandena(), es así como encontramos la solución.