Pontificia Universidad
JAVERIANA
Colombia

Sistemas Operativos

Docente John Corredor.

Autores Karol Dayan Torres Vides - Andres Eduardo Meneses Rincon.

Informe Taller02: Fork( )

PASO 1 Entrada de Parámetros

Primero será necesario asegurarnos de que nuestro sistema nos permita ingresar la información de la siguiente manera: N1 fichero00 N2 fichero01. Para esto necesitamos pasar los argumentos a la función main( ) y los imprimiremos en pantalla para verificar que estén correctos, además de verificar que el numero de argumentos sea correcto.

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

    /* Verificar los parametros que recibe el programa principal */
    printf("Cantidad de elementos del fichero 00: %s\n", argv[1]);
    printf("Nombre primer fichero 00: %s\n", argv[2]);


    printf("Nombre segundo fichero 01: %s\n", argv[3]);
    printf("Cantidad de elementos del fichero 01: %s\n", argv[4]);

    /* Verfifcar el numero de argumentos que recibe el programa principal*/
^C Hprintf("La cantidad de argumentos ingresada contando el nombre del programa fué: %d\n", argc);
    printf(
    if ((argc< 5) || (argc > 5)){
       printf("El número de argumentos recibido es inválido %d\n ", argc);
       return 1;
    }
    return 0;
}
```

PASO 2 Reserva de Memoria y Lectura de Archivos

Ahora es preciso crear los arreglos dinámicos con los que vamos a trabajar, asignamos memoria a través del uso de malloc( ), teniendo en cuenta su estructura la cual nos pide la cantidad de

memoria requerida, la cual encontramos usando sizeof() y los N1-N2, también tendremos en cuenta que esta memoria asignada debería ser liberada al final del programa.

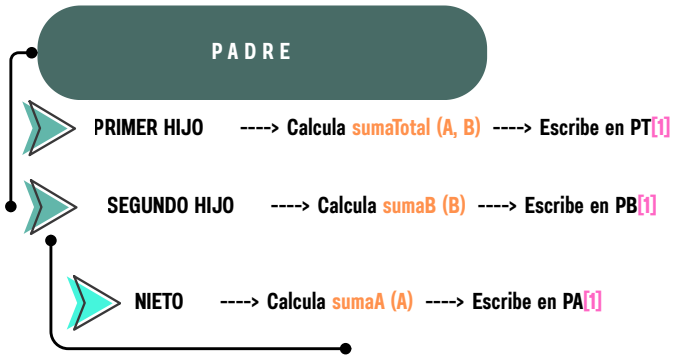
```c
int *A = (int *)malloc((size_t)N1 * sizeof(int));
int *B = (int *)malloc((size_t)N2 * sizeof(int));
if (!A || !B) {
    fprintf(stderr, "Fallo al reservar memoria para A o B.\n");
    return EXIT_FAILURE;
}
```

Hay que verificar que, en los ficheros, los arreglos que se encuentran ahí tienen tamaño N1 y N2 según corresponda. De modo que si hay un N1-N2 menor o mayor en los ficheros, tendremos un error. Se usará fscanf para leer N enteros de los ficheros con esta función:

```c
static void verificarN_ele(const char *path, int *arr, size_t n) {
    FILE *f = fopen(path, "r");
    if (!f) {
        fprintf(stderr, "No se pudo abrir '%s': %s\n", path, strerror(errno));
        exit(EXIT_FAILURE);
    }
    for (size_t i = 0; i < n; ++i) {
        if (fscanf(f, " %d", &arr[i]) != 1) {  // " %d" salta espacios/lineas
            fprintf(stderr, "Error: el archivo '%s' tiene menos de %zu enteros.\n", path, n);
            fclose(f);
            exit(EXIT_FAILURE);
        }
    }
    /* Verificar que no sobren enteros */
    int dummy;
    if (fscanf(f, " %d", &dummy) == 1) {
        fprintf(stderr, "Error: el archivo '%s' tiene más de %zu enteros (excede N declarado).\n", path, n);
        fclose(f);
        exit(EXIT_FAILURE);
    }
    fclose(f);
}
```

PASO 3 Jerarquía de los procesos y pipes

Tendremos varios Pipes, uno por resultado y la jerarquía que se usará es la siguiente:



PASO 4 Realizar las sumas y escribir en los PIPE

Para las operaciones que se estarán realizando (sumas), será necesario una función que sume elementos de los arreglos. Usar variables de tipo int bastaría, pero, ¿Qué sucede si queremos

procesar cantidades muy grandes? Podría usarse entonces variables de tipo long long que almacenan 8 bytes, sin embargo, en sistemas de Linux comúnmente long long y long miden lo mismo.

```c
static long suma_array(const int *arreglo, size_t n) {
    long suma = 0;
    for (size_t i = 0; i < n; ++i) suma += arreglo[i];
    return suma;
}
```

Parseo y validación de argumentos

El programa comienza validando argc==5 y mapeando:

N1 = argv[1], fichero00 = argv[2], N2 = argv[3], fichero01 = argv[4].

Se usan conversiones con strtol para N1 y N2, rechazando valores no numéricos o no positivos.

```c
static long parsearlong(const char *s, const char *nombre) {
    char *fin = NULL;
    errno = 0;
    long v = strtol(s, &fin, 10);
    if (errno != 0 || fin == s || *fin != '\0' || v <= 0) {
        fprintf(stderr, "Error: %s inválido: '%s'\n", nombre, s);
        exit(EXIT_FAILURE);
    }
    return v;
}
```

Reserva dinámica y lectura exacta

Se reservan A y B con malloc(N1*sizeof(int)) y malloc(N2*sizeof(int)).

Se leen exactamente N1 y N2 enteros con fscanf(" %d", ...).

Si faltan enteros: error claro y salida y si sobran enteros en el fichero: error claro y salida.

Esto garantiza coherencia entre N1/N2 y los datos reales.

Creación de pipes (uno por resultado)

Se crean tres pipes antes de cualquier fork() para que los hereden los procesos:

PA: nieto → padre (para sumaA),

PB: segundo hijo → padre (para sumaB),

PT: primer hijo → padre (para sumaTotal).

Regla: fd[0] lee, fd[1] escribe. Cada emisor tiene su pipe exclusivo para evitar mezclar mensajes.

Primer fork() (primer hijo)

El primer hijo calcula sumaTotal = sum(A) + sum(B) con acumulador long long y la envía por PT[1].

Cierra todo lo que no usa: PA[0],PA[1], PB[0],PB[1], PT[0].

Escribe sumaTotal en PT[1], cierra PT[1] y termina.

Segundo fork() (segundo hijo) y nieto

El segundo hijo calcula sumaB y la enviará por PB[1].

Antes, crea al nieto con un fork() adicional: el nieto calcula sumaA y la envía por PA[1].

Cierres importantes:

En el segundo hijo, mantener PA[1] abierto solo hasta forkar al nieto (para que lo herede). Luego cerrarlo para que el padre reciba el EOF de PA cuando termine el nieto.

El nieto cierra todo salvo PA[1], escribe sumaA y cierra PA[1].

Proceso padre

Cierra los extremos de escritura PA[1], PB[1], PT[1] y lee sumaA, sumaB y sumaTotal desde PA[0], PB[0], PT[0].

Hace waitpid a ambos hijos (el nieto ya lo espera su padre) y libera A y B.

Imprime las tres sumas.

Cierres y sincronización

Cada proceso cierra los descriptores que no necesita. Esto evita bloqueos en las lecturas del padre por falta de EOF y previene fugas de descriptores.

waitpid evita procesos zombi.

Conclusiones

En este taller se pudo evidenciar y utilizar lo siguiente:

Demostrar la creación y jerarquización de procesos con fork() y manejo correcto de pid y ppid.

Implementar la comunicación entre procesos con pipe() usando un canal por resultado para evitar condiciones de carrera.

Garantizar el cierre oportuno de descriptores no usados para prevenir bloqueos y asegurar la llegada de EOF al lector.

Validar la coherencia entrada-datos leyendo exactamente N enteros por fichero y reportando errores claros ante inconsistencias.

Sincronizar la finalización de procesos con waitpid() para evitar procesos zombi y asegurar recolección ordenada.

Liberar memoria dinámica y recursos del sistema, asegurando limpieza y finalización correcta del programa.