Informe Sistemas Operativos TP1

Grupo 17

Integrantes:

- Albertoni Salini Juan Bautista (64189)
- Gago Juan Diego (64137)
- Oliva Morroni Juan amancio (64105)

Profesores:

- Aquili Alejo Ezequiel
- Gleiser Flores Fernando
- Godio Ariel
- Mogni Guido Matías

2C 2024

1 Introducción

En el presente informe se presenta la primera entrega del trabajo para la materia de Sistemas Operativos. El objetivo del mismo es desarrollar un programa en C que permita calcular el hash MD5 de varios archivos en paralelo. Para ello, se requiere implementar comunicación entre procesos.

2 Proyecto.

2.1 Estructura general.

El proyecto consta de 3 programas en C. El **application.c, child.c, view.c** además de un **Makefile** y un archivo **pshm_ucase.h**. El core del trabajo consiste en la comunicación entre los tres programas en C.

- **application.c:**

  El propósito general de este programa es la coordinación de los 3 programas para lograr el objetivo final. El proceso no debe coordinar simplemente 3 procesos, ya que la cantidad de procesos child es "variable", cambiando el *define* que se encuentra en application debe coordinar diferente cantidad de *slaves*.

  Genera los *child* y luego por cada uno crea 2 *pipes* para la conexión Slave-Master y Master-Slave. Luego debe repartir los archivos que recibe como argumento y enviar a cada hijo 2 argumentos mediante el pipe correspondiente al iniciar (nuevamente podría andar para N, pero se decidió que envíe de a 2) y luego a medida que cada child vaya terminando el padre irá leyendo la información y de haber más archivos para procesar se le irán enviando de a 1 archivo a los slaves que antes hayan terminado. A medida que se lee la información application va escribiendo el contenido leído por el pipe del hijo en la Shared Memory y en el archivo result.txt. Por último mediante la Shared Memory efectuará la conexión con el view (en caso de ejecutarse)

- **child.c:**

  Este proceso se encarga de calcular el md5 mediante el programa md5sum. Recibe el path a los archivos que debe calcular el md5 por entrada estándar y devuelve el md5 por entrada estándar.

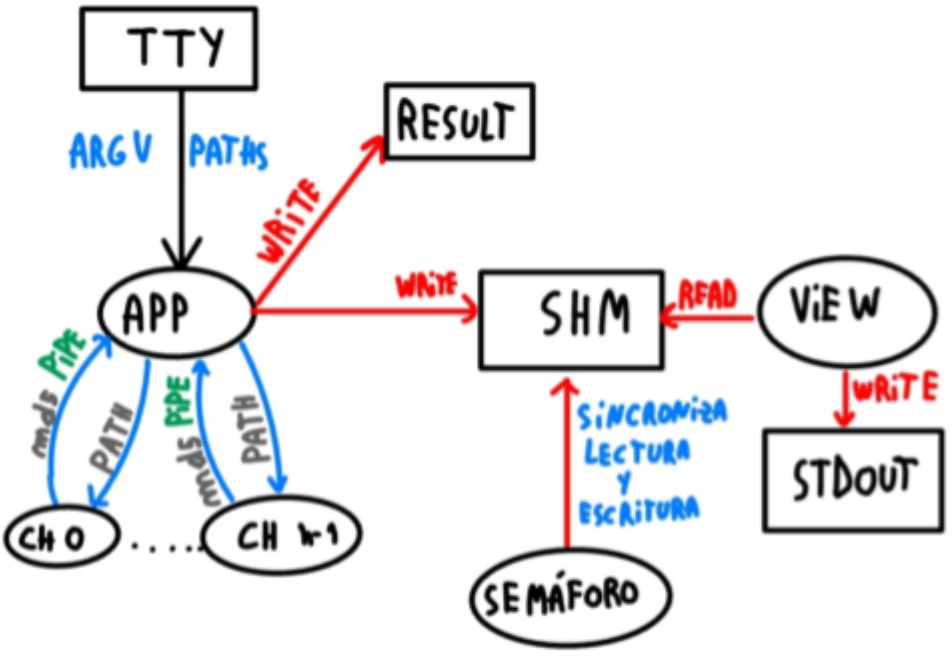
- **view.c:** El proceso vista recibe por entrada estándar el path a la Shared Memory que debe conectarse. Luego de conectarse a la misma el proceso imprime lo que se le va cargando por el buffer dentro de la Shared Memory.

- **Makefile:** Permite fácilmente crear los ejecutables mediante el comando "*make*"

- **pshm_ucase.h:** La estructura utilizada para la shared memory. El fragmento de código es referenciado del ejemplo de "shm_open" del manual.

2.2 Diagrama ilustrativo de cómo se conectan los procesos.

Para facilitar el gráfico, a application.c lo llamaremos app y por cada proceso de child.c lo ilustraremos como ch y su número habiendo un total de n childs. Por último la shared memory la ilustramos como SHM.



1. Diagrama ilustrativo del sistema

2.3 Decisiones durante el desarrollo.

- Se decidió enviar 2 archivos inicialmente a cada proceso hijo.
- Se definió una cantidad fija y considerable de esclavos mediante un define (DEFINE SLAVES 5).

- Para comunicar los diferentes procesos, se utilizaron pipes entre el proceso padre y los hijos. Se hizo uso de dos pipes por hijo, uno para lectura del hijo y escritura del padre (enviar archivos para su procesamiento), y otro para lectura del padre y escritura del hijo (enviar md5 sum). De igual manera se hizo uso de un buffer compartido mediante memoria compartida entre el proceso aplicación y el proceso vista para enviar los resultados.
- Se definieron tamaños de buffer arbitrarios que el equipo consideró "suficientemente grandes" para el desarrollo del trabajo. Esto se hizo debido a que no se requería de un buffer circular o similar.
- Como mecanismo de sincronización se utilizó un semáforo que indica al proceso cuando hay un resultado o resultados disponibles para imprimir y bloquea al mismo en caso de no haber resultados. Con respecto a este punto, se hizo uso de un semáforo sin nombre, siguiendo el ejemplo del manual *shm_open (3)*.

3 Guia técnica del desarrollo.

3.1 Limitaciones del desarrollo

❖ Durante el desarrollo se trató de implementar que en aplicación le puedan mandar archivos los cuales no existían y que *application* los evite. En la consigna del TP indica que se debe asumir que los archivos existen y que se posee los permisos requeridos, por lo que se descartó dicha implementación. De igual manera, el grupo considera que es una limitación importante el hecho de que el código no esté preparado para que un usuario final pueda pasarle una lista de archivos de los cuales alguno no cumpla con dichas características.

❖ Se tomó la decisión de no implementar un buffer circular por lo explicado anteriormente, pero ello lleva a otra limitación importante. En caso de pasarle una lista lo suficientemente larga de archivos a *application* el buffer no podría guardar todos los mismos y se pisaría memoria.

❖ Al ejecutar el programa con valgrind y pipear la salida al proceso vista, el programa puede quedarse esperando indefinidamente. El comando utilizado es el siguiente,
*$ valgrind --leak-check=full --show-leak-kinds=all -v ./application * | ./view*
Utilizando el comando *top* se puede ver que el proceso application termina pero no el proceso vista.

❖ Se podría considerar como un "bug" a algo descubierto de casualidad relacionado a la cantidad de esclavos. y por mera casualidad en algún momento ese número se cambió a 3 y

con algunas cantidades de archivos el proceso aplicación no finaliza. Se intentó averiguar la raíz del problema pero el mismo no nos resultó fácil de replicar ya que no pudimos hallar un patrón o algo similar que nos diera indicio de qué estaba pasando.

3.2 Instructivos para compilar y ejecutar.

En el proyecto se puede usar el archivo Makefile el cual permite escribir en la terminal *make* y que se compilen todos los archivos. En el caso de que se haga un cambio en algún archivo se puede volver a ejecutar *make* y se va a recompilar solamente dicho archivo. La forma de eliminar los archivos creados por *make* es ejecutar *make clean* el cual removerá todos los archivos generados previamente. Por último está la opción de ejecutar *make clean all* el cual borra todos los archivos que hayan sido creados y crea el resto.

Una vez compilado los archivos se puede a ejecutar de varias formas:

- *$ ./application <files> | ./view*

  Ejecuta en una terminal y la salida estándar de *application* la cual es el path de la Shared Memory es leída por entrada estándar por el *view.*

- *$ ./application <Files>*
- *$ ./view <SharedMemoryPath>*

  La segunda forma de ejecutar es mediante dos terminales, en la primera se ejecuta *application y* en no más de 2 segundos se debe ejecutar el *view* para que se conecte a la shared memory, cuyo path se pasó por argumento.

- *$ ./application <files>*

  En dicha forma de ejecutar se vuelve a hacer en una sola terminal pero no se conecta el proceso *vista* por lo que no imprime en pantalla y el resultado se verá meramente en *result.txt.*

3.3 Citas de fragmentos de código y herramientas utilizadas.

A continuación se listan fragmentos de código que hemos extraído de terceros.

Hemos utilizado este *struct* extraído del manual para el manejo de la memoria compartida. Hemos tomado como referencia el ejemplo provisto en el mismo que para la inicialización de los semáforos utilizaba *sem_init* y trabaja con semáforos sin nombre.

```c
struct shmbuf {
    sem_t  resultadoDisponible;           /* POSIX unnamed semaphore Puedo leer tranqui*/
    sem_t  resultadoLeido;           /* POSIX unnamed semaphore Puedo escribir tranquilo*/
    size_t cnt;             /* Number of bytes used in 'buf' */
    char   buf[BUF_SIZE];   /* Data being transferred */
    int totalFiles;
};
```

2. Struct de manual.

```c
int
main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "Usage: %s /shm-path\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    char *shmpath = argv[1];

    /* Create shared memory object and set its size to the size
       of our structure */

    int fd = shm_open(shmpath, O_CREAT | O_EXCL | O_RDWR,
                      S_IRUSR | S_IWUSR);
    if (fd == -1)
        errExit("shm_open");

    if (ftruncate(fd, sizeof(struct shmbuf)) == -1)
        errExit("ftruncate");
```

3 Ejemplo de uso de memoria compartida del manual (1).

```
if (ftruncate(fd, sizeof(struct shmbuf)) == -1)
    errExit("ftruncate");

/* Map the object into the caller's address space */

struct shmbuf *shmp = mmap(NULL, sizeof(*shmp),
                           PROT_READ | PROT_WRITE,
                           MAP_SHARED, fd, 0);
if (shmp == MAP_FAILED)
    errExit("mmap");

/* Initialize semaphores as process-shared, with value 0 */

if (sem_init(&shmp->sem1, 1, 0) == -1)
    errExit("sem_init-sem1");
if (sem_init(&shmp->sem2, 1, 0) == -1)
    errExit("sem_init-sem2");
```

4 Ejemplo de uso de memoria compartida del manual (2).

Con respecto a comandos que nos fueron de utilidad a lo largo del desarrollo del trabajo práctico utilizamos:

$ ls -l /proc//fd:* Utilizado para visualizar los descriptores de archivos y así poder identificar si se estaban abriendo y cerrando de forma adecuada.

$ top: Utilizado para visualizar el estado de los procesos en tiempo real y así ver la creación y liberación de procesos hijos.

Por último utilizamos las dos herramientas recomendadas por la cátedra tanto valgrind como pvs-studio. Para el uso de pvs se siguió la guía de github de Alejo Aquili y para ejecutar con valgrind se probó con las tres formas descritas anteriormente.

$ valgrind --leak-check=full --show-leak-kinds=all -v <*Opcion*>

<*Opcion*> = ./application <filepaths> | ./view

<*Opcion*> = ./application <filepaths>

<*Opcion*> = ./application <filepaths> y en otra terminal ./view <*SharedMemoryPath*>