

**TRABAJO PRÁCTICO**

**FINAL SISTEMAS OPERATIVOS:**

**“Inter Process Communication”**

**INTEGRANTES:**

* **Lucio Fernandez**
* **Nicolas Mongiello**
* **Pablo Cumpe**

Contenido

[Objetivo del trabajo práctico 3](#_Toc533069476)

[Introducción 3](#_Toc533069477)

[Explicación breve de la idea implementada 4](#_Toc533069478)

[Detalle de implementación 5](#_Toc533069479)

[Test de rendimiento 9](#_Toc533069480)

[Conclusiones 9](#_Toc533069481)

# **Objetivo del trabajo práctico**

El trabajo práctico consiste en aprender a utilizar los distintos tipos de IPCs presentes en un sistema POSIX. Para ello se implementará un sistema que distribuirá tareas de cálculo pesadas entre varios pares.

# **Introducción**

El siguiente trabajo práctico fue implementado en Linux con lenguaje C. El desarrollo del mismo implicó una investigación sobre los siguientes temas:

.PIPES

.NAME PIPE

.SEGMENTOS DE MEMORIA COMPARTIDA

.SINCRONIZACION DE PROCESOS MEDIANTE EL USO DE SEMAFOROS

.CREACION Y USO DE HILOS

.CREACION Y USO DE PROCESOS HIJOS

.SEÑALES

.HASH MD5

.MANEJO DE ARCHIVOS Y DIRECTORIOS

.CREACION DE TEST PARA PRUEBAS

# **Explicación breve de la idea implementada**

El trabajo consta de cuatro componentes:

1) El proceso padre.

2) Procesos esclavo.

3) Dos hilos disparados por el proceso padre dedicados al manejo del buffer de memoria compartida y a la espera del proceso vista.

4) El proceso vista.

El **proceso padre** recibe por parámetro

.La dirección de un directorio que contiene el total de archivos a procesar.

.La ruta donde se escribirá el archivo final de salida con los resultados generados.

.La cantidad de procesos esclavos a crear

La función padre es distribuir los archivos en N procesos hijos (procesos esclavo) para su procesamiento.

La comunicación entre el proceso padre con sus hijos y viceversa, se realiza mediante pipes. Cada proceso hijo cuenta con un pipe independiente en el cual el proceso padre distribuirá equitativamente los procesos pendientes de ser procesados.

Cada **proceso hijo** tiene como objetivo generar el hash de cada archivo enviado por el proceso padre usando el algoritmo md5 (ver sección *Cálculo del hash*) y luego el resultado es devuelto al padre.

El proceso padre a su vez genera dos hilos:

-El primer hilo crea un segmento de memoria compartida en el cual almacena el resultado del procesamiento enviado por los procesos hijo, funcionando como ‘productor del buffer’ (­­­­­­­­ver­­­­ sección: *Manejo del Buffer*). Por último, se encarga de escribir dichos resultados en el archivo de salida.

-El segundo hilo queda a la espera de la aparición de un proceso externo llamado “vista” (ver sección: *Proceso Vista*), al cual le comparte mediante un named pipe el identificador del segmento de memoria compartida.

El **proceso vista**, luego de conectarse con el segmento de memoria compartida, imprime por pantalla los resultados obtenidos del buffer, funcionando como consumidor del mismo.

Una vez finalizados los dos hilos y todos los procesos esclavo , finaliza el **proceso padre**.

El siguiente esquema grafica la descripción anterior:

**Main**

**Thread\_maneja buffer**

ad

**Thread\_espera\_vista**

**……**

KILL

**BUFFER**

**PROCESO VISTA**

**Resultados HASH**

# **Detalle de implementación**

A continuación, se agrega el pseudocódigo de las partes más relevantes de la implementación:

**Manejo del Buffer de memoria compartida:**

Como la lectura y la escritura del buffer de memoria compartida no se realizan a nivel atómico se debe implementar una sincronización de procesos mediante semáforos para asegurar exclusión mutua sobre el recurso compartido. Para ello, se utilizará el esquema productor/consumidor, en donde el proceso padre (mediante el hilo **thread\_maneja\_buffer**)funcionará como proceso productor del buffer y el proceso vista lo hará como consumidor del mismo. La transferencia en este punto se manejará por bloques de Bytes.

**Thread\_maneja\_buffer (Productor)**

semaforo mutex=1;

semaforo sem\_productor= BUFFER\_SIZE;

semaforo sem\_consumidor=0;

void PRODUCTOR( ){

int pos=0;

crear segmento de memoria compartida para buffer;

crear memoria compartida para semáforos;

attach a memoria compartida;

while(hay dato){

escribit datos en archivo de salida;

sem\_wait(sem\_productor);

sem\_wait(mutex);

data[pos%BUFFER\_SIZE]=dato;

pos++;

sem\_post(mutex);

sem\_post(sem\_consumidor);

}

}

Los semáforos implicados se crean en memoria compartida para ser accesibles desde ambos procesos.

**Proceso vista (Consumidor)**

El proceso vista recibe por parámetro el ID del proceso padre, el cual utiliza para enviarle una señal mediante el comando KILL. El proceso padre una vez recibida la señal, envía a la vista el identificador del segmento de memoria compartida a través de un named pipe para que pueda comenzar la lectura del buffer.

Luego de leer la totalidad del contenido del segmento de memoria compartida, el proceso vista envía otra señal para avisar al proceso main que la lectura a finalizado.

void vista(pid idMain){

enviar señal al proceso padre;

esperar identificador de segmento de memoria compartida;

attach to shared memory;

int pos=0;

while(hay dato){

sem\_wait(sem\_productor);

sem\_wait(mutex);

data[pos%BUFFER\_SIZE]=dato;

pos++;

sem\_post(mutex);

sem\_post(sem\_consumidor);

}

enviar señal al proceso padre indicando finalización;

fin;

}

}

**Proceso padre**

void main(){

validar entrada;

inicializar pipes para comunicación con procesos hijo;

lanzar hilos;

generar n procesos hijo (esclavos);

distribuir los archivos a procesar entre los hijos;

esperar procesos hijos;

esperar hilos;

fin main;

}

**Cálculo Hash MD5**

Teniendo en cuenta la premisa del trabajo práctico para este punto, empezamos a investigar y vimos que existía la posibilidad de ejecutar un comando de sistema operativo y capturar lo devuelto a través de la salida estándar de comandos.

Esto lo hicimos con la función popen() que recibe por parámetro el comando y el tipo de operación (lectura o escritura). Este comando crea un pipe, crea un nuevo proceso hijo e invoca al comando del sistema operativo. Una vez ejecutado el comando, la función popen() (en modo lectura) vuelca el resultado a la salida estándar de comandos.

De esta manera, resolvimos el cálculo del hash definiendo una función que ejecute el comando md5sum a través de la función popen() y leyendo el resultado de la línea de comandos.

**Observaciones**

-Pudimos observar que en caso de que se llene el sector de memoria compartida el **proceso padre** queda en “stand by” hasta que se libere, es decir que aparezca el **proceso vista** y empiece a consumir y liberar la memoria.

-Se genero un script para generar N archivos de distinto tamaño, a mayor tamaño del archivo mayor es el tiempo en procesarlos.

# Test de rendimiento

Realizamos algunas pruebas de rendimiento con el fin de analizar el tiempo de ejecución en función de la cantidad de procesos determinados para una ejecución.

Para eso, creamos un script cuyo objetivo era medir el tiempo de ejecución y volcar el resultado en un archivo de texto plano. El script recibe por parámetro la cantidad de procesos hijos a crear.

Pseudocodigo del script

tiempoComienzo = tomar instante de comienzo de ejecución;

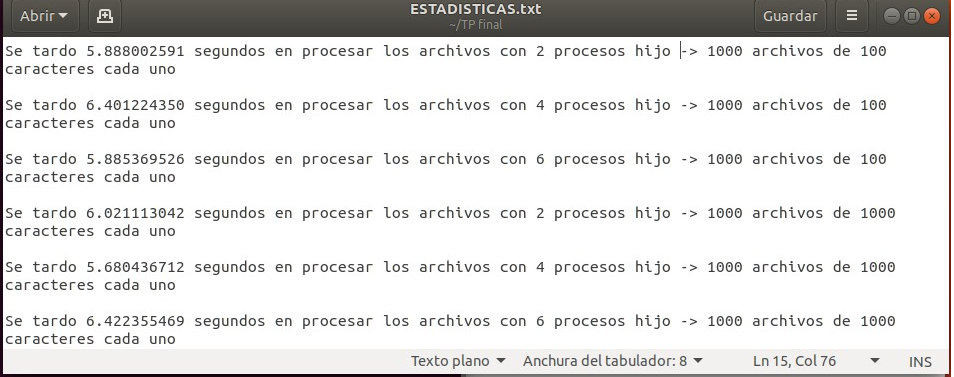
Ejecutar procesoPrincipal;

tiempoFinalizacion = tomar instante de finalización;

tiempoTotal = tiempoFinalizacion – tiempoComienzo;

crearArchivo “estadísticas” con el resultado de tiempoTotal;

A continuación, mostramos los resultados de las pruebas que hicimos



**Observaciones**En función de los datos recolectados, nos llamó la atención que los tiempos no disminuyeron a medida que aumentábamos la cantidad de procesos hijos.  
Quizás para tener una mejor estimación en los tiempos de ejecución, podríamos ejecutar el proceso una cantidad considerable de veces (por ejemplo 1000) y luego promediar todas las mediciones, de manera tal de calcular un tiempo más preciso.

# Conclusiones

Creemos que los mecanismos que ofrece POSIX para compartir y manipular datos entre aplicaciones diferentes es muy interesante. No obstante, también creemos que es muy importante la correcta manipulación de herramientas para gestionar la sincronización y el acceso a memoria compartida para así poder evitar bloqueos o inconsistencia de datos producida por varios procesos que quieren operar sobre el mismo segmento de memoria compartida.