Instituto Tecnológico de Buenos Aires

Sistemas Operativos

Informe 1: Inter Process Communication

Integrantes grupo 5:

Perez Riviera, Mateo 61170

Szejer, Ian 61171

Ortu, Agustina Sol 61548

Profesores:

Aquili, Alejo Ezequiel

Mogni, Guido Matias

Godio, Ariel

Decisiones tomadas durante el desarrollo

Durante el desarrollo del trabajo práctico se barajaron muchas opciones para el manejo de la comunicación entre master y esclavos, pero finalmente, siguiendo la recomendación de la cátedra, se eligió que la comunicación entre N esclavos y master se realice por 2*N pipes. De esta forma pudimos experimentar con la función select y sus macros.

Luego para la comunicación entre app y visualizer se tomaron las siguientes decisiones:

- La app y visualizer se podían ejecutar con un pipe de forma secuencial o por separado, pasando como argumento a visualizer la clave del sharedMemory.
- El app podría empezar a escribir resultados sin ser necesario la ejecución de visualizer.
- Visualizer podría leer en simultáneo a la escritura, siempre y cuando no se lea exactamente el mismo resultado que está siendo escrito.
- Visualizer sería el encargado de cerrar todo lo relacionado a su comunicación y su sincronización.

Para llevar a cabo estas decisiones se decidió utilizar un semáforo nombrado para la sincronización de los procesos, el cual se crea en ambos y se destruye solo en visualizer, y una sharedMemory la cual se vincula mediante la key. El semáforo se encargará de pausar el visualizer si no hay resultados para leer o están siendo escritos. Y por la sharedMemory se pasarán los resultados y la señal para que visualizer finalice.

Mostramos un diagrama ilustrando cómo se conectan los diferentes procesos:

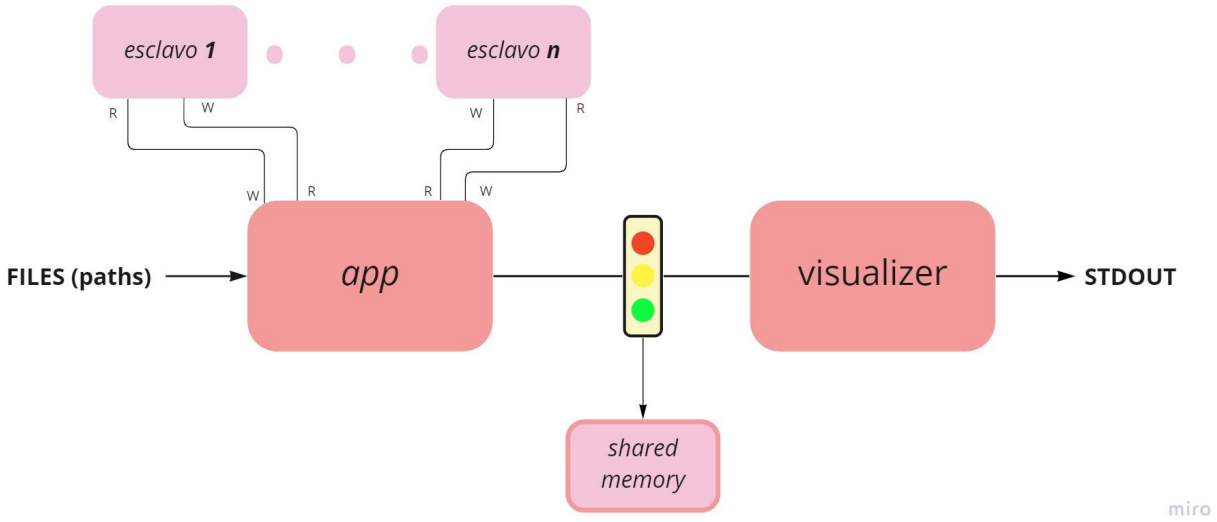

Diagrama de funcionamiento

Instrucciones de compilación y linkedicion

Para compilar y linekeditar todo el proyecto, tan solo se debe correr el comando make clean (Para asegurarse de que no haya archivos residuales) y luego el make all y todos los ejecutables se crearán correctamente.

Para ejecutarlo hay 2 opciones:

1. Correr en la consola ./app.exe ./"Archivos a analizar" | ./visualizer.exe.
2. Correr en la consola ./app.exe ./"Archivos a analizar" y en otra consola ./visualizer.exe "key" con la key que imprima app a su consola al ejecutarse.

Limitaciones

Las limitaciones del programa serían las siguientes:

- Que se mande un path que exceda el tamaño máximo determinado.
- Que se manden tantas tareas como para que la sharedMemory se llene.
- Correr muchos visualizer con la misma key y un solo app.

Problemas encontrados y su solución

Nos encontramos con varios problemas a la hora de crear los pipes y crear los esclavos.

Por ejemplo con la función select nos sucedió que quedaba dentro y no salía de la misma. Luego vimos que no calculamos el mayor file descriptor para el primer argumento de la función, por lo tanto lo implementamos y se soluciono.

También hubo problemas con la comunicación con los esclavos, por alguna razón no se enviaban bien los paths y luego la app no recibia nada. Se soluciono tras la implementación de la función setvbuff al principio del programa esclavos.c que se asegura que la impresión por printf se realice como corresponde.

Visualizer.c no terminaba porque le intentábamos mandar un EOF y este nunca lo recibía. Terminamos tomando la decisión de mandar un '@' como señal.

El programa no funcionaba adecuadamente con más de un archivo, esto se debía a que printElements en visualizer.c al tratar de printear otro resultado retomaba desde el \n del elemento anterior, para solucionar esto nos aseguramos que cada lectura de shared memory era de un resultado

diferente y se consiguió aumentando el puntero en la cantidad de elementos leídos +2. Se le sumó 1 por el \n que no contaba como elemento leido y 1 mas por el \0 final.

Citas de fragmentos de código obtenidos de otras fuentes

Para la creación de la shared memory, buscamos información y nos basamos en un codigo visto por un canal de youtube, del usuario “Jacob Sorber”

```
static int getSharedBlock(){
    //obtengo el id o lo creo en caso de q no exista
    return shmget(IPC_PRIVATE, BLOCK_SIZE, 0644 | IPC_CREAT);
}

char * createMemoryBlock(int *id){
    *id = getSharedBlock();
    if(*id==ERROR){
        return NULL;
    }

    //mapeo el bloque al id devolviendome un puntero al mismo
    char * result=shmat(*id, NULL, 0);
    if(result == (char *)ERROR){
        return NULL;
    }

    return result;
}

char * joinMemoryBlock(int id){
    //mapeo el bloque al id devolviendome un puntero al mismo

    char * result=shmat(id, NULL, 0);
    if(result == (char *)ERROR){
        return NULL;
    }

    return result;
}

//Me voy del bloque
bool leaveMemoryBlock(char * block){
```

```
    return (shmdt(block) != ERROR);
}

//destruyo el bloque
bool destroyMemoryBlock(int id){
    return (shmctl(id, IPC_RMID, NULL) != ERROR);
}
```

Para la creación del semáforo hicimos algo similar pero diferimos mucho más del codigo visto, también por el canal de youtube de "Jacob Sorber"

```
static sem_t * createSharedSemaphore(){
    sem_t* semaphore=sem_open(SEM_ESCRITURA, IPC_CREAT, 0660,0);
    if (semaphore == SEM_FAILED){
        perror("sem_open/producer");
        exit(EXIT_FAILURE);
    }
    sem=semaphore;
    return sem;
}

sem_t * joinSemaphore(){
    if (sem==NULL)
        createSharedSemaphore();
    return sem;
}

int leaveSemaphore(sem_t * sem){
   return sem_close(sem);
}

int terminateSemaphore(sem_t * sem){
   return sem_destroy(sem);
}
```

https://github.com/mateoperezrivera/SO_TP1

Hash del commit : 196b599c037ac00c099dd8695bfeb62218fee55b