Sistemas Operativos

Proyecto Final

Miner Rush

Angela Valderrama Ricaldi
Luis Felice Ruggiero
07/05/2023

Introducción

Esta cuarta práctica consiste en la implementación total del proyecto final de la asignatura Sistemas Operativos. Este proyecto ha seguido un desarrollo incremental, en el cual, en cada práctica, con los conocimientos adquiridos se implementaba una parte funcional del objetivo principal del programa. De esta manera, en esta última iteración, se han unificado todas las partes funcionales de las anteriores prácticas con algunos cambios y nuevos módulos.

En cada práctica, se trataban conocimientos específicos que luego se ponían en práctica para desarrollar un programa sencillo y funcional que luego formaría parte de este proyecto final. Estos conocimientos están relacionados con la ejecución de procesos y la comunicación entre ellos haciendo uso de diferentes mecanismos como los semáforos, las señales, las tuberías, las colas de mensajes y la memoria compartida.

Este proyecto consta de procesos: Minero, Registrador, Comprobador y Monitor que se comunican entre ellos para llevar a cabo un sistema inspirado en la tecnología Blockchain pero con menos complejidades. El proceso Minero se encarga de crear hilos que intentarán resolver una prueba de esfuerzo y conseguir una moneda si la solución es correcta. El proceso Comprobador es el encargado de comprobar que la solución es válida mientras que el proceso Monitor mostrará la evolución del sistema por pantalla. Por último, el proceso Registrador estará asociado a un único proceso Minero y registrará las rondas de minado realizadas por él.

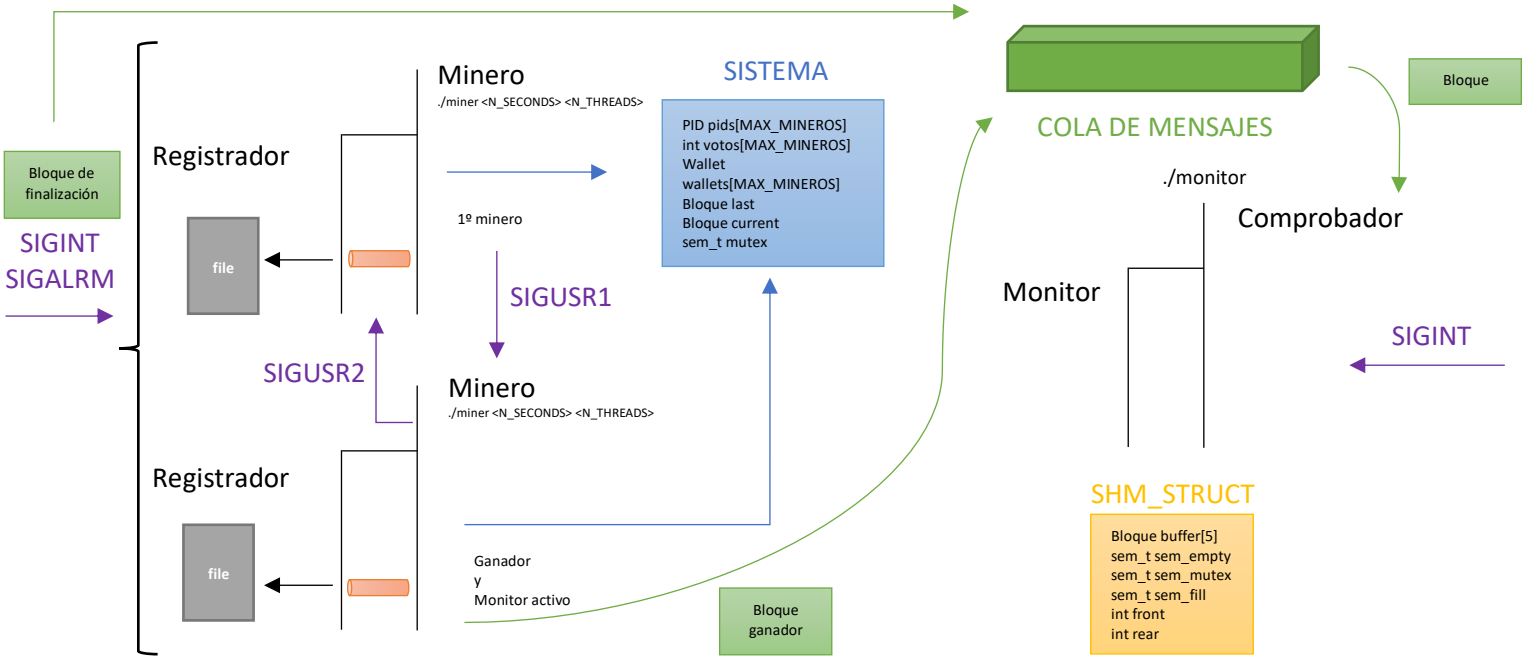
Estos procesos, se crean a partir de dos programas: ./miner y ./monitor con los argumentos necesarios para su ejecución y se comunicarán entre ellos mediante señales, tuberías, colas de mensajes y memoria compartida.

Al ser un programa concurrente y multiproceso, para asegurar la sincronización, coordinación y persistencia de datos, se han utilizado como mecanismo, los semáforos con nombre entre procesos que no se conocen entre ellos y sin nombre para procesos que son conocidos.

El desarrollo de este proyecto final, ha sido mucho más sencillo gracias a las iteraciones de las prácticas anteriores a pesar de haber encontrado dificultades en el camino. Por eso, estamos orgullosos y agradecidos de haber llevado a cabo un proyecto de esta magnitud y completarlo en toda su totalidad.

Diseño

Para la implementación del sistema *Miner Rush*, se ha seguido el siguiente diagrama:



El desarrollo de este proyecto requiere la participación de 4 procesos: Minero, Registrador, Comprobador y Monitor que se crearán al ejecutar los programas ./miner y ./monitor con sus respectivos argumentos si los necesitaran.

Al ejecutar ./miner <N_SECONDS><N_THREADS> se crea un proceso principal Minero que creará un proceso hijo Registrador. El proceso minero, si es el primero en conectarse al sistema (mediante la sincronización del semáforo *sem_minero*), creará un segmento de memoria compartida entre mineros, preparará el primer bloque a minar con su determinado objetivo y avisará al resto de mineros con la señal SIGUSR1. En cambio, otro minero arrancado con ./miner <N_SECONDS><N_THREADS> abrirá ese segmento de memoria compartida y esperará a recibir la señal SIGUSR1 para empezar el minado.

El minero que primero encuentre la solución a la prueba de esfuerzo, se proclamará ganador y comunicará la resto de mineros que dejen de minar y que empieza la votación con la señal SIGUSR2. Los demás mineros perdedores votarán el bloque resuelto y si éste resulta aceptado, el proceso ganador enviará el bloque al proceso Comprobador mediante una cola de mensajes.

Este proceso Comprobador es el proceso principal del programa ./monitor. Este crea el segmento de memoria compartida entre los procesos Comprobador y Monitor. El proceso Comprobador, una vez se haya creado la memoria compartida, creará un proceso hijo Monitor. El proceso Comprobador, recibirá un bloque mediante la cola de mensajes, comprobará que la solución del bloque es la correcta y comunicará la validación al proceso Monitor escribiendo el bloque en un buffer circular de su memoria compartida. El proceso Monitor, leerá del buffer circular e imprimirá por pantalla los datos relacionados con el bloque resuelto así como su validación.

El proceso Minero ganador, preparará el siguiente bloque a minar con el nuevo objetivo y comunicará a los demás mineros que empieza la siguiente ronda con la señal SIGUSR1, la cual estaban esperando recibir. Todos los mineros participantes en esa ronda, enviarán el bloque resuelto ganador a su correspondiente proceso Registrador. Este Registrador, habrá creado un fichero que tendrá como nombre el PID del proceso Minero y escribirá en él los datos relacionados con el bloque resuelto en dicha ronda.

Si la señal SIGINT o SIGALRM es recibida por parte de un proceso Minero, éste termina su ejecución en el sistema, reseteando los valores necesarios así como liberando los recursos utilizados. También, avisará a su respectivo proceso Registrador que debe terminar su ejecución para poder recogerle sin problema y terminar el programa.

Si se trata del último minero en el sistema, éste tendrá que avisar también al proceso Comprobador (si estuviese activo) mediante un bloque de finalización para que él y el proceso Monitor terminen su ejecución. Por otro lado, si el proceso Comprobador recibe la señal SIGINT, éste debe recoger a su hijo Monitor y terminar su ejecución.

El desarrollo del proyecto se ha probado con las líneas de comando proporcionadas en el enunciado y se ha verificado que todos los requisitos se han cumplido. Los principales problemas que se encontraron fueron los relacionados con la sincronización entre mineros, cuando se enviaba la señal SIGUSR2 y cuando se proclamaba un ganador. Además, se tuvo problemas, pero se llegó a implementar, un bloqueo de la señal SIGINT y/o SIGALRM cuando se realizaba el minado ya que los recursos utilizados por los hilos no se liberaban correctamente.

Por lo demás, ha sido una práctica que ha conllevado mucho esfuerzo pero gracias a la implementación incremental en cada práctica de la asignatura, se ha podido conseguir la encapsulación total del programa así como la modularidad requerida.

Módulos

Bloque

Como se ha indicado en el diseño del sistema, el objeto Bloque encapsula toda la información necesaria sobre la prueba de esfuerzo realizada. Es por eso, que es necesario manejar correctamente el objeto y sus diferentes atributos. En este módulo se implementan las funciones necesarias para la manipulación de las estructuras de datos Wallet y Bloque.

```
/**
 * @brief Estructura que representa una wallet
 *
 */
typedef struct Wallet
{
    int pid;
    int coins;
} Wallet;
```

Esta estructura representa una cartera que contendrá el PID del minero al que pertenece y el número de monedas que posee. Las monedas se incrementarán siempre y cuando un minero declarado ganador haya recibido el número de votos mínimo para aceptar su solución.

Para poder manejar las carteras de una manera más simple desde otros módulos, se han implementado las siguientes funciones:

- Wallet wallet_init();
- void wallet_set_pid(Wallet *wallet, int pid);
- pid_t wallet_get_pid(Wallet wallet);
- int wallet_get_coins(Wallet wallet);
- void wallet_inc_coins(Wallet *wallet);
- void wallet_set_coins(Wallet *wallet, int coins);

```c
/**
 * @brief Estructura que representa un bloque
 *
 */
typedef struct Bloque
{
    int id;
    int target;
    int solution;
    pid_t winner;
    Wallet wallets[MAX_MINEROS];
    int votes;
    int positives;
    int flag;
} Bloque;
```

Esta estructura representa el objeto Bloque que guardará la información relacionada con una ronda de minado. Un bloque contiene un identificador, el objetivo a resolver en la prueba de esfuerzo y la solución de dicha prueba de esfuerzo siendo los tres de tipo entero. Además, almacena el PID del minero ganador de la ronda, así como el estado de todas las carteras de los mineros activos en el instante de la ronda. Junto con esta información, se guarda el registro de la votación, exactamente, el número total de votos, los votos positivos y una variable que actúa como *flag* para guardar la validación de la ronda por parte del proceso Comprobador.

Las funciones que se han implementado para el manejo son *getters* y *setters* para simplificar la lógica de los otros módulos y mantener la modularidad, son las siguientes:

- Bloque bloque_init(int id, int target, Wallet *wallets);
- int bloque_set_solution(Bloque *bloque, int solution);
- int bloque_set_winner(Bloque *bloque, pid_t winner);
- int bloque_set_votes(Bloque *bloque, int votes);
- int bloque_set_positives(Bloque *bloque, int positives);
- int bloque_set_flag(Bloque *bloque, int flag);
- int bloque_get_id(Bloque *bloque);
- int bloque_get_target(Bloque *bloque);
- int bloque_get_solution(Bloque *bloque);
- pid_t bloque_get_winner(Bloque *bloque);
- int bloque_get_votes(Bloque *bloque);
- int bloque_get_positives(Bloque *bloque);
- int bloque_get_flag(Bloque *bloque);
- Wallet *bloque_get_wallets(Bloque *bloque);

Sistema

Según el diseño del programa, toda la información relevante sobre el sistema se guarda en un segmento de memoria compartida entre mineros que será del tipo Sistema.

```c
/**
 * @brief Estructura que representa el sistema de minado
 *
 */
typedef struct Sistema
{
    pid_t pids[MAX_MINEROS];
    int votes[MAX_MINEROS];
    Wallet wallets[MAX_MINEROS];
    int n_mineros;
    int n_bloques;
    Bloque last;
    Bloque current;
    sem_t mutex;
} Sistema;
```

Esta estructura incluirá una lista de los PID de los mineros activos hasta un máximo de 100 mineros, así como sus votos y carteras de tipo Wallet correspondientes. Se llevará un control del número de mineros registrados en el sistema y del número de bloques resueltos para poder identificar a los siguientes. Por otro lado, se guardará una instancia del bloque actual a resolver y el anterior. Como este segmento se compartirá entre varios procesos mineros, el acceso se deberá proteger con un semáforo *mutex* también compartido para asegurar la sincronización y evitar lecturas y/o escrituras erróneas.

Para el correcto manejo del objeto Sistema, se han implementado una serie de funciones que simplifican la lógica del módulo. El objeto Sistema que resulta de ejecutar la función *mmap(),* se inicializa con la función *sistema_init()* donde los contadores se ponen a 0, las listas guardan el valor -1 y el semáforo mutex se crea con el valor 1.

Cuando el sistema registra un minero, éste lo hace con la función *sistema_add_minero()* siempre y cuando no se haya alcanzado un máximo de 100 mineros activos en el sistema. Se añade su PID a la lista de PIDs de mineros activos, se establece su cartera inicial con 0 monedas y se incrementa el número de mineros activos en el sistema en ese instante. En cambio, si el minero, decide salir del sistema, se ejecutará la función *sistema_remove_minero()* con la funcionalidad contraria: quitando su PID de la lista de mineros activos, borrando su cartera y decrementando el número de mineros activos.

Para guardar la información relacionada con la votación del bloque ganador, se hace uso de la función *sistema_set_bloque_votes()* y para resetear la información se utiliza la función *sistema_reset_votes().* Cuando se empieza una ronda, el bloque actual se configura con la función *sistema_set_current_bloque()* que inicializa un nuevo bloque con un determinado objetivo e incrementa el número de bloques creados. Además, con la función *sistema_set_last()* se guarda la instancia del objeto bloque de la ronda anterior.

Por otro lado, cuando se necesite actualizar la información del bloque, exactamente cuando el bloque se declare como bloque ganador, se utilizará la función *sistema_update_bloque()* y así establecer la solución

encontrada y el PID del minero ganador de la ronda. Como ya sabemos, al minero ganador se le insertará una moneda más en su cartera y se llevará su registro en la lista de carteras del bloque actual y del sistema con la función *sistema_update_wallets().*

Por último, cuando el último minero salga del sistema, éste debe liberar los recursos utilizados en el segmento de memoria compartida y eliminarlo con éxito. Para ello, se hará uso de la función *sistema_destroy()* que destruye el semáforo mutex creado para sincronizar los procesos minero y de esta manera ya se podrá proceder a la eliminación del segmento de memoria.

Miner_func

Gran parte de la funcionalidad del programa reside en el módulo Miner y es por eso que, para mantener una programación modular, organizada y limpia, hemos creado este módulo *miner_func* donde se encapsulan varias de las funcionalidades del módulo Miner.

mqd_t queue_setup()

Esta función es la encargada de configurar la instancia de la cola de mensajes para un minero. Esta cola de mensajes permitirá la comunicación entre un minero y el monitor. El tamaño máximo de mensajes en la cola es de 10 y el mensaje será de tipo Bloque. La función devuelve el objeto de tipo mqd_t que representa la cola de mensajes.

*void check_voting(Sistema *sistema)*

En esta función se comprueba que todos los mineros activos han votado por el bloque propuesto como ganador. La comprobación consiste en el recuento de votos en la lista de votos del sistema y en una serie de intentos, específicamente 100. Esto se hace así para evitar que el sistema se quede bloqueado en el caso que un minero se salga del sistema antes de añadir su voto. La escritura y en este caso la lectura en un segmento de memoria compartida, tiene que estar protegido y sincronizado para evitar pérdidas de datos con el semáforo mutex del sistema.

*void count_votes(Sistema *sistema)*

Una vez, hayan votado todos los mineros activos o hayan acabado los intentos de comprobación, se procede al recuento de los votos. Los votos con un 1 son positivos, mientras que los que tienen un 0, son negativos. El propio minero ganador se vota a sí mismo para poder asegurar el correcto funcionamiento con un solo minero.

El bloque se considerará aprobado cuando reciba al menos la mitad de los votos y si es así, al minero ganador se le añadirá una moneda en su cartera, quedando registrado en las carteras del bloque y del sistema. El bloque quedará actualizado con los votos totales y positivos de esa ronda y después, se procederá a resetear la lista de votos para la siguiente ronda de minado.

*void vote(Sistema *sistema)*

Todos los mineros que no son el minero ganador se comportarán como mineros perdedores de esa ronda y votarán el bloque propuesto como ganador para aprobarlo o no. Para ello, se ejecuta esta función, en la que comprueban que la solución propuesta es correcta y votan según ese criterio: 1 si es correcta y 0 si no lo es.

*Sistema *first_miner_setup()*

El primer minero que entra en el sistema es el encargado de crear el segmento de memoria compartida para todos los mineros. En esta función se crea la memoria compartida con el tamaño de la estructura Sistema y haciendo el mapeo necesario para poder acceder y modificar, si es necesario, los elementos de la estructura del Sistema. La función devuelve la dirección del objeto Sistema mapeado del segmento de memoria compartida para utilizarlo en la lógica del programa sin problemas.

*Sistema *miner_setup()*

Los demás mineros que no han sido los primeros abren el segmento de memoria compartida ya creado, lo mapean en un objeto de tipo Sistema y comprueban que la información de la estructura ya está inicializada por el primer minero: que el tamaño de la estructura coincide y que el último semáforo está inicializado con el valor correcto.

*void voting_setup(Sistema *sistema, int target)*

El minero proclamado ganador configura el sistema de votación, es decir, actualiza el bloque con la solución encontrada, prepara la lista de votos del sistema y envía la señal SIGUSR2 a los demás mineros para que dejen de minar y empiecen a votar por el bloque proclamado como ganador.

*void send_signal(Sistema *sistema, int signal)*

Con esta función, simplificamos el envío de señales al resto de mineros, sin incluir al minero que envía. De esta manera, se hace uso de la función cuando se quieren enviar las señales SIGUSR1 y SIGUSR2 en los momentos adecuados para empezar una nueva ronda y empezar la votación, respectivamente.

*int send_to_monitor(Sistema *sistema, mqd_t mq)*

El minero ganador envía mediante la cola de mensajes el bloque ganador al proceso Comprobador para que se valide siempre y cuando un proceso Comprobador esté activo para recibir un mensaje. Esta función recibe la instancia de la cola de mensajes para poder usar la función *mq_send().*

*int send_to_registrador(Sistema *sistema, int fd[2])*

Con esta función, todos los mineros envían el bloque ganador de la ronda a su respectivo proceso Registrador antes de empezar la siguiente ronda. El bloque ganador, en este caso, se corresponde con el último bloque resuelto. Además, la función recibe el descriptor de fichero del pipe para poder enviar el bloque mediante la función *write().*

Miner

Este módulo contiene la implementación necesaria del proceso Minero. Se ejecuta a través del siguiente comando *./miner <N_SECONDS><N_THREADS>*, donde N_SECONDS es el número de segundos en los que el proceso minero estará activo y N_THREADS el número de hilos que se van a utilizar para resolver la prueba de esfuerzo.

Los módulos anteriores facilitan la funcionalidad principal del programa que reside en el módulo Miner. Además de ellos y sus respectivas estructuras, el módulo Miner utiliza la siguiente estructura de datos:

```
/**
 * @brief Estructura que representa la información de un hilo
 *
 */
typedef struct Info
{
    int lower;
    int upper;
} Info;
```

Esta estructura representa la información que encapsula cada hilo trabajador que crea el proceso Minero, siendo *lower* y *upper,* variables de tipo entero que indican el rango de valores que se probarán al realizar la prueba de esfuerzo.

*void main(int argc, char *argv[])*

El programa recibe los argumentos del comando a ejecutar y crea la tubería que comunicará los procesos Minero y Registrador. El proceso principal será el proceso Minero, que creará un proceso hijo, el proceso Registrador. El proceso Minero ejecutará su función *minero()*, mientras que el Registrador, la función *registrador().*

void minero(int n_threads, int n_seconds)

Nada más empezar, el proceso Minero cierra el extremo de la tubería que no va a utilizar, el de lectura. Se configuran las máscaras de señales, los manejadores de señales con la función *signals()* y se crean o abren los semáforos necesarios para sincronizar los procesos: *sem_minero*, *sem_winner* y *sem_mqueue.*

- sem_minero: este semáforo será el encargado de diferenciar entre el primer minero en conectarse al sistema y los demás. Es necesario para la creación del segmento de memoria compartida del sistema que utilizarán todos los mineros.

- sem_winner: con este semáforo, se diferencia entre el minero que se declara ganador y los demás que tomarán el papel de mineros perdedores. De esta manera, sólo hay un único minero ganador y el resto votarán por él.
- sem_mqueue: se utiliza este semáforo para detectar si hay un proceso Comprobador activo y esperando a recibir un bloque mediante la cola de mensajes.

Si se es el primer minero, mediante la función *firstminer_setup()* se configura la creación del segmento de memoria compartida del sistema y se devuelve el objeto que lo representa para después inicializar sus datos con la función *sistema_init().*

Por otro lado, si no se es el primer minero, se ejecuta la función *miner_setup()* para abrir el segmento de memoria compartida y recibir el objeto que representa el mapeo.

Todos los mineros, se registrarán en el sistema mediante la función *sistema_add_minero()*, protegiendo el acceso con el semáforo mutex del sistema. Después, se configura la cola de mensajes y se recibe el objeto que la representa con la función *queue_setup()*.

A continuación, se prepara la primera ronda. Si se es el primer minero, se inicia el bloque actual con target igual a 0 y se envía la señal SIGUSR1 a los demás mineros para empezar la ronda de minado con la función *send_signal().* En cambio, si no se es el primer minero, se espera a recibir la señal SIGUSR1 para obtener el target de la ronda y empezar a minar.

El bucle principal se ejecutará siempre y cuando se tenga activada la variable global *got_SIGUSR1* ya sea por haberla recibido o por ser el primer minero o el minero ganador de la ronda. Se desbloquea la señal SIGUSR2 y se bloquea SIGINT y SIGALRM para la función de *minado().* De esta manera, el proceso minero y sus hilos creados pueden recibir la señal SIGUSR2 y parar de minar si es necesario, pero no recibir la señal SIGINT o SIGALRM para que terminen correctamente y liberen los recursos necesarios.

Una vez se hayan completado el minado, ya sea por haber encontrado la solución o por haber recibido la señal SIGUSR2, se vuelve a establecer la flag *found* a 0 y bloquea la máscara de señales que contiene SIGUSR1 y SIGUSR2 y se desbloquea la que contiene SIGINT y SIGALRM.

Seguidamente, se procede a la proclamación del minero ganador utilizando la función *sem_trywait()* con el semáforo con nombre *sem_winner.* Si se es el proceso ganador, protegido con el semáforo mutex, se prepara el sistema de votación con la función *voting_setup().* Después, se comprueba que todos los mineros activos han votado con la función *check_voting()* y se procede al recuento con la función *count_votes().*

A continuación, se guarda el bloque actual resuelto como bloque anterior y con el semáforo *sem_mqueue,* si hay un proceso Comprobador activo y esperando, se envía el bloque resuelto mediante la función *send_to_monitor().*

Para empezar la siguiente ronda, se configura el bloque actual siendo target la solución obtenida del bloque anterior y se envía la señal SIGUSR1 al resto de mineros para empezar la ronda. Se levanta el semáforo *sem_winner* para poder elegir un nuevo minero ganador y se envía el bloque anterior al proceso Registrador a través de la función *send_to_registrador().*

Por último, si se es un minero perdedor, se espera a recibir la señal SIGUSR2 que indica que la votación ha empezado. Protegido entre el semáforo mutex del sistema, el minero ejecuta la función *vote()* para

guardar su voto en la lista de votos del sistema. Después, espera a recibir la señal SIGUSR1 para empezar la siguiente ronda y cuando la reciba, se enviará el bloque resuelto anterior a su respectivo proceso Registrador y se obtendrá el target de la siguiente ronda.

void registrador()

Por otro lado, el proceso hijo, Registrador, cierra el extremo de la tubería que no va a utilizar, el de escritura. Se crea el archivo donde se escribirán los bloques, con el nombre del PID del proceso padre. El proceso Registrador, leerá del pipe los bloques enviados en cada ronda por parte del proceso Minero. Una vez, recibido el bloque, se escribirá toda la información en el fichero mediante las funciones *dprintf()* que escriben en el descriptor de fichero devuelto por la función *open().* El Registrador leerá del pipe siempre y cuando no se cierre el otro extremo de la tubería del proceso Minero. De lo contrario, se procederá a su finalización.

void signals(int n_seconds)

En esta función se establecen las máscaras de señales a utilizar y se configuran los manejadores. En la máscara *set* se añadirán las señales SIGUSR1 y SIGUSR2; mientras que en la máscara *intset* se añadirán las señales SIGINT y SIGALRM. Habrá un manejador único para todas las señales, *handler()*, y la máscara que se bloqueará primero será *set.* Además, se establece una alarma de *n_seconds* con la función *alarm()*.

void minado(int n_threads)

En esta función, se inicia una ronda de minado. Es necesario conocer el número de hilos que se van a crear para poder paralelizar la prueba de fuerza entre ellos. Para comenzar, se crea un array del número de hilos indicados para llevar un seguimiento de su creación y un array de estructuras info pertenecientes a cada hilo.

Se procede al cálculo del número de iteraciones que deberá realizar cada hilo en su prueba de fuerza. En el bucle de creación de hilos, aparte de crear los hilos, se calcula el rango de búsqueda y se guarda la información en la estructura info. Cuando ya se hayan lanzado todos los hilos, comienza la espera de cada uno de ellos mediante la función pthread_join. Por último, se liberan los recursos de memoria utilizados y finaliza la función.

*void *prueba_de_fuerza(void *info)*

Esta es la función que ejecutará cada uno de los hilos en cada ronda. Se realizará una búsqueda secuencial dados los parámetros del rango lower y upper para encontrar el valor que aplicada la función pow_hash da como resultado el target. A continuación, la flag *found* se pondrá a 1 para indicar a los demás hilos que se ha encontrado la solución y deben terminar el minado. Finalmente, se almacenará el nuevo objetivo en la variable global target. Esta búsqueda se ejecutará siempre y cuando la variable *found* y *got_SIGUSR2* tienen como valor 0, es decir, no se ha encontrado la solución y no se ha recibido la señal SIGUSR2.

void clear()

Esta función es la que se ejecuta cuando se recibe la señal SIGINT o SIGALRM, en la cual el proceso Minero termina su ejecución. Lo primero de todo, es eliminar los datos relacionados con ese minero del sistema con la función *sistema_remove_minero().* Si se es el último minero que había en el sistema, entonces se procede a eliminar todos los recursos relacionados con el sistema. Es decir, se avisa al proceso Comprobador (si está activo) con un bloque de finalización, que el sistema va a acabar.

Después, se eliminan los recursos del sistema, específicamente el semáforo mutex y se procede al desmapeo del objeto que representa el sistema y a eliminar su nombre asociado. Se cierra la cola de mensajes, se elimina el nombre asociado a la cola, se cierran los semáforos creados y se eliminan los nombres asociados. Para avisar al proceso Registrador, se cierra el extremo de la tubería.

Si no se fuese el último minero del sistema, sólo es necesario cerrar la cola de mensajes, los semáforos y el extremo de la tubería.

Monitor

Este módulo contiene la implementación del programa de monitorización. Para ejecutarlo, se debe escribir *./monitor*. Para la implementación del módulo se ha creado una estructura de datos que será de utilidad en el desarrollo del programa.

```
/**
 * @brief Estructura a compartir entre Comprobador y Monitor
 *
 */
typedef struct shm_struct
{
    Bloque buffer[5];
    sem_t sem_empty;
    sem_t sem_mutex;
    sem_t sem_fill;
    int front;
    int rear;
} shm_struct;
```

Esta estructura representa el segmento de memoria compartida entre el proceso Comprobador y el proceso Monitor. Contiene el array circular o buffer de tipo Bloque en el cual se irá insertando y extrayendo bloques según el proceso Comprobador vaya recibiendo de la cola de mensajes y el proceso Monitor vaya consumiendo del buffer. Como se trata de un array circular, es necesario llevar un registro de las últimas posiciones leídas y escritas y para eso están las variables *front* y *rear*.

Para asegurar el acceso concurrente a esta zona compartida de memoria, se lleva a cabo un sistema de sincronización basado en el esquema Productor-Consumidor, para el cual es necesario una serie de semáforos: sem_empty, sem_mutex y sem_fill. Al inicializarlos, estos semáforos serán sin nombre porque ya se comparten en el segmento de memoria.

*void main(int argc, char *argv[])*

Se crea el segmento de memoria compartida, se establece su tamaño al de la estructura shm_struct y se mapea en el objeto shm_struc que inicializará sus datos con la función *init_struct().* Después, se procede a la creación del proceso hijo Monitor y una vez se haya hecho esta diferenciación, simplemente llamamos a las funciones pertinentes para ejecutar ya sea el monitor o el comprobador.

void comprobador()

Este proceso va a ser el encargado de recibir los bloques a través de un mensaje en la cola de mensajes, provenientes del proceso Minero y después de comprobarlos, los enviará al proceso Monitor mediante su segmento de memoria compartida para que los imprima por pantalla.

Primero, se creará o se abrirá el semáforo *sem_mqueue* para comunicar al proceso Minero que el proceso Comprobador está activo y esperando a recibir bloques. Después, se configura el manejador de la señal SIGINT, *handler()*.

Una vez esté configurada la señal, abriremos la cola de mensajes por la que recibiremos mensajes del proceso Minero, para empezar a escuchar de ella.

Posterior a la configuración principal del proceso, empezará el bucle principal donde se realizarán las siguientes tareas:

- Se levanta el semáforo *sem_mqueue* para notificar al processo Minero de que el proceso Comprobador está activo y puede recibir mensajes por la cola.
- Se recibe el mensaje de la cola de mensajes mediante la función *mq_receive()* y se guarda en la variable *bloque*. Si el bloque recibido tiene como target -1, se corresponde con el bloque de finalización. Como consecuencia, se cambia el valor de la variable *exit_loop* a 1 para salir del bucle principal.
- De lo contrario, éste será un bloque normal y se comprobará el resultado ejecutando la prueba de fuerza.
- Como su propio nombre indica, este proceso se encarga de comprobar que esos valores son correctos mediante la función *comprobar()* que devuelve el valor 0 si es incorrecto o 1 si es válido. Con esta información, se establece como flag del bloque el valor de retorno de la función *comprobar()* para que el proceso Monitor sepa si la solución es aceptada o no.
- A continuación, se procede a escribir en el segmento de memoria el bloque recibido. Para ello, el proceso Comprobador y Monitor seguirán un esquema Productor-Consumidor para poder permitir la sincronización entre los dos procesos y el acceso concurrente a la memoria compartida. Esto se controla con el uso de sem_empty y sem_mutex, verificando que hay espacio para escribir en el buffer de bloques y accediendo para añadir uno.
- Si podemos escribir, al ser un buffer circular, primero incrementamos el valor de la variable *rear,* que indica la última posición escrita y escribimos en la posición *rear % BUFFER_SIZE* (donde BUFFER_SIZE será el tamaño del array circular, que en nuestro caso será 5).
- Finalmente, indicaremos que es seguro acceder a la sección crítica de nuevo, y que se ha llenado una posición más en el array circular con los semáforos sem_mutex y sem_fill, respectivamente.

Si se ha salido del bucle principal, es porque se ha recibido el bloque de finalización y éste se ha escrito con éxito en el array circular. Por eso, el proceso Comprobador, ejecutará la función *munmap* para desasociar el segmento de memoria compartida. Además, dejará de escuchar de la cola de mensajes, cerrándola con la función *mq_close* y borrando los enlaces con la función *mq_unlink*. De igual manera, se cerrará el semáforo *sem_mqueue* con *sem_close* y se desvinculará el nombre con *sem_unlink.* Por último, se termina la ejecución del programa.

void monitor()

Este proceso sólo se encargará de mostrar los resultados que envía el proceso Comprobador, por pantalla. Para ello, se configura el manejador de la señal SIGINT como SIG_IGN para ignorar la señal cuando se reciba. Esto es porque el único proceso que debe recibirla es el proceso Comprobador. Una vez se haya configurado la señal, empezará el bucle principal del programa que realizará las siguientes acciones:

- Mediante el uso de los semáforos sem_fill y sem_mutex, esperará a que haya algún Bloque escrito en el buffer circular (siguiendo el esquema Productor-Consumidor) y esperará a que sea seguro acceder a la sección crítica para poder leer la estructura compartida.
- Si puede leer de la memoria compartida, actualizará el valor de la variable *front,* que indica la última posición leída y leerá de *front % BUFFER_SIZE*.
- Al igual que el comprobador, si el bloque recibido tiene como target -1, se cambiará el valor de la variable *end_loop* a 1 para salir del bucle principal.
- Si ese no es el caso, se imprimirá la información relacionada con el bloque haciendo distinción de si la solución es válida o no, según el valor de la variable *flag*. Después, permitirá la escritura de otro proceso e indicará que se consumió una entrada de la cola mediante los semáforos sem_mutex y sem_empty.

Cuando el proceso Monitor salga del bucle principal, será porque ha recibido el bloque de finalización. Teniendo en cuenta que este proceso sólo de encarga de consumir bloques y de mostrar su contenido por pantalla, es el indicado de destruir los semáforos utilizados ya que es el último en acceder a ellos. Después de limpiar esos recursos, para finalizar, eliminará de su espacio de direcciones el segmento de memoria compartida y terminará su ejecución.

*void init_struct(shm_struct *shm_struc)*

Como hemos mencionado antes, el proceso Comprobador que crea el segmento de memoria compartida, debe inicializar la estructura que se guarda en ese espacio. Por eso, esta función, inicializa los valores de *front* y *rear* que sirven para llevar un control de las posiciones leídas y escritas. Además, se crean los

semáforos sin nombre necesarios para el esquema Productor-Consumidor que llevarán a cabo los proceso Comprobador y Monitor, sem_mutex, sem_empty y sem_fill.

int comprobar(int target, int res)

Esta función se utiliza para comprobar que la solución del bloque recibido por el proceso Minero es correcta. Para ello, se vuelve a realizar la prueba de fuerza en la solución para verificar que se recibe de vuelta el valor del objetivo. Si es así, la solución será válida y se devolverá un 1, mientras que, si la solución no es válida, el valor de retorno de la función será 0.

int clear()

Esta función es la que se ejecuta cuando llega la señal SIGINT que indica que el proceso Comprobador terminará su ejecución al igual que el proceso Monitor. Nada más empezar, bajamos el semáforo *sem_mqueue* para que el proceso Minero sepa que el Comprobador ya no está activo. El target del bloque será -1 para indicar al proceso Monitor que debe terminar su ejecución.

A continuación, se cierra la cola de mensajes, el semáforo *sem_mqueue* y se procede a escribir en memoria compartida el bloque de finalización. Una vez escrito, usando los semáforos del esquema Productor-Consumidor, se desasocia el segmento de memoria compartida entre Comprobador y Monitor y se espera al proceso hijo con la llamada a la función *waitpid().*