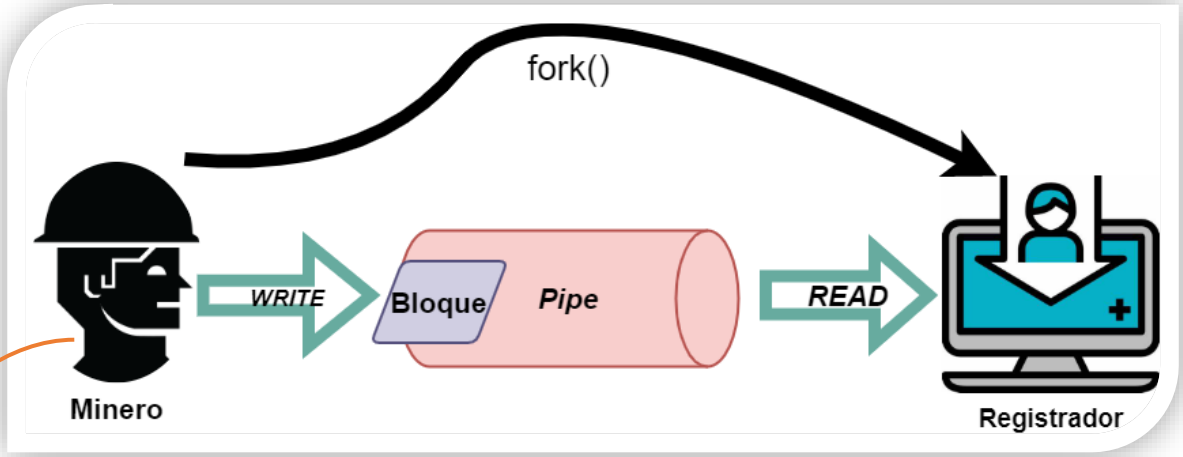
Memoria de Prácticas SOPER: Miner Rush

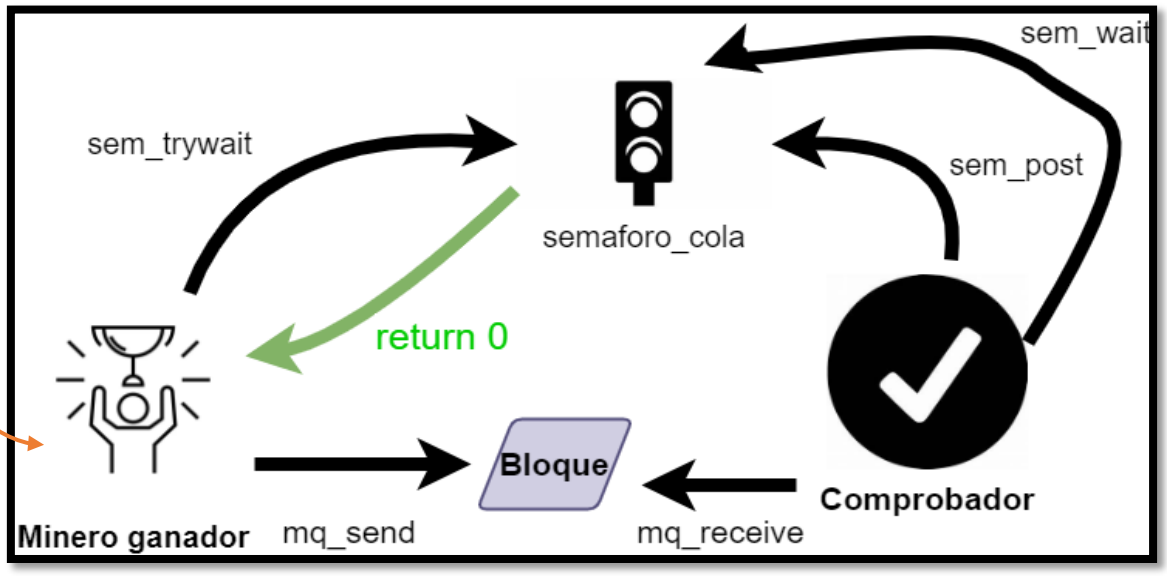
Santiago de Prada Lorenzo

Diagramas del sistema

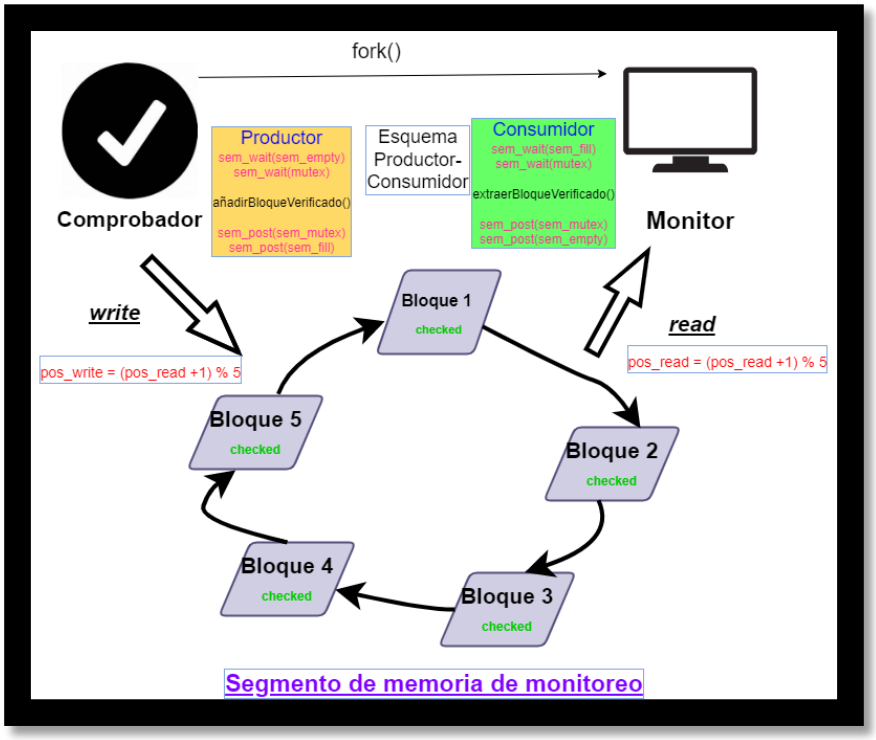
En este diagrama podemos observar la comunicación entre cada proceso minero con su respectivo Registrador, mediante una tubería.



Ronda de minado

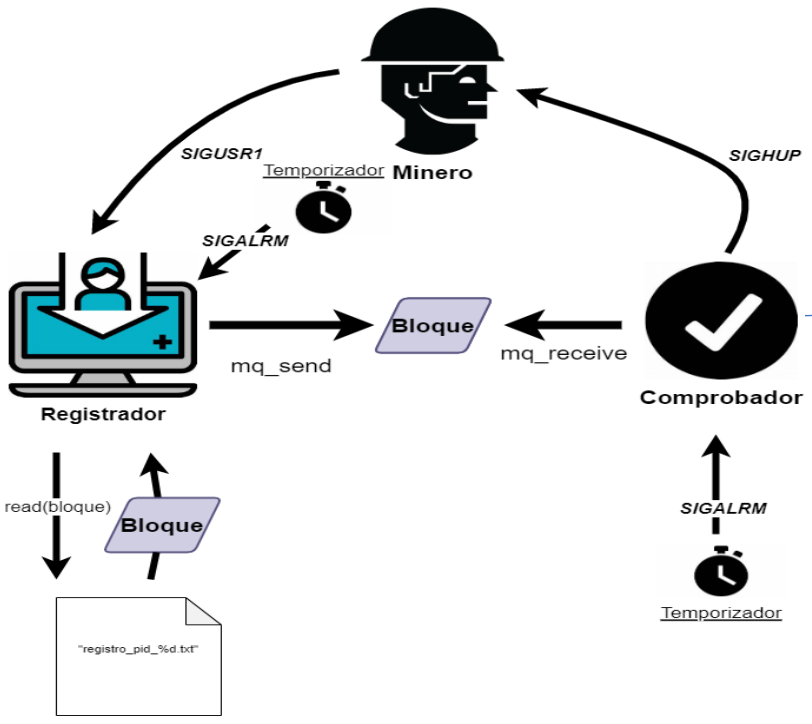


En este diagrama podemos observar el mecanismo de sincronización entre el minero ganador de la ronda con el comprobador. Mediante una cola de mensajes y un semáforo para sincronizarlo.



Aquí observamos como es la comunicación mediante el proceso comprobador con el monitor, mediante un segmento de memoria compartido, que se comporta con un buffer circular que almacena hasta 5 elementos y siguiendo un proceso de sincronización Productor-Consumidor mediante 3 semáforos asociados al segmento de memoria.

Diagrama Mecanismo de recuperación de bloques:



Cuando recibe un bloque el comprobador envía el bloque al monitor siguiendo el esquema del diagrama anterior

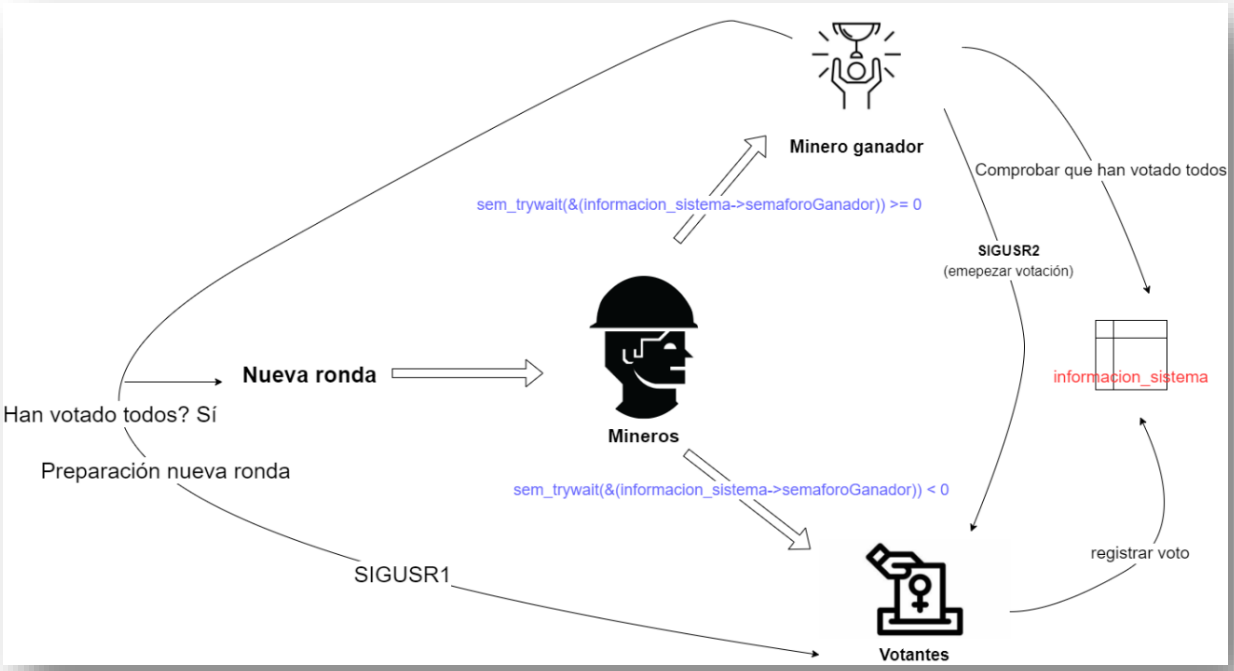


Diagrama del funcionamiento de cada ronda de minado

Diseño del Sistema

archivo “miner.c”:

Este archivo engloba la funcionalidad de los procesos mineros junto a su respectivo proceso registrador, en este archivo se programa la comunicación por tubería mediante el Registrador y el minero, además de realizarse el minado.

También observamos que aquí, en caso de haber un proceso comprobador activo, realizamos la tarea de enviarle los bloques que solicite por la cola.

A la hora de entrar en el sistema, distinguimos 2 tipos de minero: Primer minero y resto de mineros. A cada uno le corresponde respectivamente las funciones info_sistema *Conexión_Memoria_Primer_Minero();

info_sistema *Conexión_Memoria_NO_Primer_Minero();

En las cuales, uno crea el segmento de memoria, da tamaño a ese segmento e inicializa sus semáforos y da valor inicial al resto de información. Y el otro, mapea simplemente el segmento de memoria.

Los mineros van a tener asociado un manejador de señal para las señales:

- SIGUSR1 (Señal que envía el minero ganador al resto de mineros activos para el arranque de una nueva ronda): Este manejador simplemente va a retornar.

- SIGUSR2 (Señal que envía el minero ganador al resto de mineros activos para que terminen de minar y que empiecen a votar): Este manejador inicializa una variable tipo global votacion a TRUE (al principio de cada ronda se pone a FALSE) indicando así a los hilos de cada minero perdedor que dejen de minar y entren en la votación. También, indica a los mineros perdedores que ya habían terminado de minar, pero no les ha dado tiempo a coger el semáforo de minero ganador que salgan de la suspensión y pasen a votar.

- SIGINT (Señal que indica finalización del minado): Pone una variable tipo Booleana global a TRUE, sigintinterrupt, que en cada ronda de minado se comprueba que esté a FALSE, para seguir haciendo rondas, y en caso de que esté a TRUE, se salga del sistema.

- SIGHUP (Señal que elegido yo utilizar para que, el proceso Comprobador, cuando quiera recuperar el historial de bloques, envíe esta señal a los mineros): Su manejador, void handler_registrador_enviar_historial(int sig) , pone la variable tipo booleana Solicitud_historial a TRUE, que se comprueba al principio de cada ronda de minado, y en caso de que esté a TRUE se envía a su correspondiente Registrador la señal SIGUSR1 para indicarles de que envíen el historial por la cola.

En la función Minado(n_threads) es donde los mineros realizan el minado de los bloques. Consiste básicamente en un bucle for, donde minarán el número de rondas pasadas por argumento. Y en cada ronda, los mineros crearán los hilos que intentarán encontrar la solución, cada uno con un rango determinado de números. Se crearán los hilos que determine la variable n_threads.

Para Comprobar que haya algún Comprobador que haya solicitado un mensaje por la cola, utilizamos un semáforo con nombre, semaforo_cola, que cada vez que solicite un mensaje, el Comprobador, realizará un sem_post de este y, cada vez que el minero

ganador quiera ver si tiene que enviarle el mensaje, hace un sem_trywait de este, y en caso de que retorne un valor no negativo lo envía y si no, pasa a la siguiente ronda.

El Registrador, cuando se le solicite recuperar el historial de bloques, abrirá el fichero y leerá con la función getlinea() y el char de retorno de la función lo utilizará para guardar los datos de cada bloque con las funciones sscanf y strtok.

Se leerá hasta que llegue a final de archivo, para ello utilizamos la función feof. O hasta que la bandera que representa el temporizador sea TRUE. Que significará que ya no debe seguir leyendo.

Cada bloque leído lo enviará por la cola de mensajes al Comprobador, cuando lo solicite mediante el semáforo semaforo_recuperacion_bloques, utilizamos otro semáforo distinto al que utiliza con los mineros ganadores para que no existan problemas de condición de carrera con estos.

Además, para el envío de mensajes, todos los registradores de cada minero competirán por ser el próximo en enviarlo mediante otro sémaforo, respetando así el principio de exclusión mutua, con el semáforo orden_envio_historial.

archivo “comprobador.c”:

Este archivo engloba la funcionalidad de los procesos Comprobador y Monitor.

Aquí se encuentra la programación de la solicitud del bloque por la cola y el manejo del segmento de memoria compartida que se comporta como una cola circular de 5 elementos para la comunicación entre el Comprobador y el Monitor.

Los bloques que se introducen en la cola circular de 5 elementos, mediante la cual se comunican el Comprobador y el Monitor, son diferentes que los bloques que lee el Comprobador de la cola de mensajes. Se trata de básicamente de un bloque especial bloque_comprobador_monitor (definido en cola.h), el cual aparte del bloque enviado por el minero al Comprobador, contiene un dato tipo booleano, en el cual registra si verifica la solución o no, para posteriormente introducirlo en el segmento de memoria que se comporta como una cola circular de 5 elementos.

El monitor consiste en un bucle en el cual mientras lee un bloque especial del segmento, cuya variable bloque_especial sea diferente de TRUE imprime los bloques. Cuando es TRUE es una señal de que debe terminar, y por lo tanto desmapea el segmento con munmap y sale del bucle. Además, indicar que tiene un índice,

pos_read, para indicar en que posición del segmento leemos, y para que no sea mayor que 5, obligamos a que sea módulo 5, y lo actualizamos tras cada lectura.

El Comprobador, lo he dividido en varias partes, 1º recupera los bloques mediante la función Recuperación_de_bloques en la cual se encarga de enviar la señal SIGHUP a todos los mineros que estén activos en el momento que este acceda al segmento de memoria del sistema. Para que estos mineros, se encarguen del proceso de envío de bloques anteriormente minado por la cola de mensajes. Antes de eso, ha comprobado a la hora de abrir el Segmento de memoria que se ha creado, en caso de que no se haya creado cierra el segmento ya que no se ha minado todavía ningún bloque.

Posteriormente, entra en un bucle hasta que recibe la señal SIGINT indicándole que termine, o hasta que recibe un bloque donde la variable bloque_especial es TRUE, donde en ese caso, envía ese mismo bloque al Monitor para que termine y se queda esperando al Monitor para finalmente terminar. Mencionar también que tiene un índice, pos_write, en la función Escribir_bloque_en_memoria_Compartida, que siempre va a tomar un valor módulo 5, simulando así que el segmento de memoria entre el Comprobador y Monitor se comporta como una cola circular de 5 elementos.

Además, el Comprobador realiza siempre un sem_post del semaforo_cola para indicar al Minero que está solicitando bloques y que así, el minero ganador se lo envíe.

Para la recuperación de bloques el Comprobador utilizará la función Recuperación_de_bloques(), en la que para detectar que bloques ya hemos recibido utilizaremos un array de booleanos hasta 100 bloques, y cuando recibimos uno nuevo ponemos en la posición correspondiente a su id = TRUE. Al principio, todos están a FALSE. Después de recibir todos los bloques, los introduce en memoria compartida respetando el esquema Productor-Consumidor.

archivo “sistema.h”:

- minero_t: estructura de datos que contiene la información de cada minero del sistema. Entre ellas está su pid, monedas, dato tipo booleano para saber si está activo, registro de los votos realizados, y número de votos realizados.

- info_sistema: estructura de datos que contiene la información del sistema (se almacena en memoria compartida). Contiene como nos indica el enunciado que debe contener, los mineros que han participado en el sistema (estén activos actualmente o no). Esos mineros se guardan en un array que almacena datos de tipo minero_t que corresponde a cada minero. También, almacena el número de mineros que han participado en el sistema, y otro entero para saber los mineros activos actualmente.

  También contiene el último bloque minado y el bloque que se está minando en ese momento.

  Además, contiene un mutex para manejar el acceso a esa información de forma exclusiva, asegurando así el principio de exclusión mutua.

  Por último, contiene un semáforo para que lo coja el minero que primero antes encuentre la solución de cada ronda de minado y asegurarnos así, que solo ese minero sea el encargado de realizar todas las funciones de minero ganador en esa ronda. Ese semáforo, se suelta tras terminar cada ronda de minado y, por lo tanto, en cada ronda tendrá oportunidad de cogerlo cualquier minero activo.

  Para asegurarnos de que antes de entrar en el sistema, los mineros que simplemente abren el segmento de memoria, es decir que no lo crean, porque ya lo ha creado el primer minero del sistema anteriormente, no entren en el sistema hasta que este primer minero no haya inicializado los semáforos que pertenecen al sistema, lo que hacemos es guardar dentro del segmento de memoria del sistema un booleano para que puedan saber si ya han sido creados los semáforos que maneja este sistema. Ya que si no están creados no entramos hasta que el encargado de crearlos los haya inicializado. (Cuando los inicializa pone a TRUE esta variable).

archivo “bloque.h”:

- Cartera: estructura de datos que representa la cartera de cada minero. Cada cartera tiene asociado el pid del minero y las monedas que contiene ese minero.
- Bloque: estructura de datos que contiene identificador de bloque, el target para el que se quiere encontrar la solución y solución encontrada para ese target. También, contiene el pid del minero ganador y un array que almacena las carteras de los mineros que han participado en el minado de ese bloque. También almacenamos un array de booleanos para almacenar los votos de aprobación de la solución. Además, almacenamos, un entero para indicar la cantidad de mineros que han votado y los votos a favor de la solución que se

han registrado. Finalmente tenemos un dato tipo booleano para indicar si es un bloque especial al comprobador y que así pueda finalizar.

archivo "cola.h":

- Segmento_monitoreo: Se trata de la estructura de datos que se va a almacenar en memoria compartida para la comunicación entre los 2 procesos que conforman el monitor (Comprobador y Monitor).
  Esta estructura va a contener: un array de bloques modificados por el Comprobador, ya que estos bloques van a contener una variable más además del bloque en sí, un booleano para registrar si el Comprobador lo ha verificado o no. Este array va a ser de 5 elementos y mediante codificación haremos que se comporte como una cola circular a la hora de extraerlos e introducirlos en memoria compartida.
  Además de eso, contiene tres semáforos, un mutex, un sem_fill y un sem_empty para llevar a cabo el esquema de Productor-Consumidor y asegurando así la exclusión mutua y un orden claro de extracción e introducción.

- bloque_comprobador_monitor: Se trata del bloque modificado por el comprobador que hemos mencionado anteriormente, contiene un bloque normal (Bloque) y además contiene un booleano para que el Comprobador registre si confirma la solución encontrada y el monitor simplemente muestra por pantalla si lo ha confirmado o no el Comprobador.
- Por último, en este fichero guardamos el parámetro struct mq_attr de la función mq_open(…). Que sirve para especificar los atributos de la cola que vamos a crear para la comunicación entre el minero y el Comprobador. En esta cola, enviaremos un elemento de tamaño tipo Bloque.

archivo "hilo.h":

- Parametro_t: Contiene la posición que indica a cada hilo, desde que posición tiene que empezar a buscar la solución. También contiene la variable número de hilos, para que el propio hilo pueda establece la posición desde la que tiene que buscar hasta cual.
- return_t: Contiene el taget para el que encontrar la solución en esta ronda, además de la solución encontrada para ese taget que se guardará en taget2, ya que será el taget de la siguiente ronda. Por último, guarda un dato tipo booleano para que señale si ha encontrado la solución o no.

Problemas

Cuando ejecuto muchos mineros a la vez, se desincronizan, y algún minero no participan justo en la siguiente ronda, si no que para volver a participar debe esperar a que pase la ronda y meterse de nuevo. Existe por tanto una desincronización, según qué momentos.

También, cuando ejecuto el miner.c con Valgrind me finaliza antes de tiempo, dándome el siguiente error, que no sabría exactamente porque es:

==69366==

==69366== Syscall param write(buf) points to uninitialised byte(s)

==69366== at 0x5057371: write (write.c:27)

==69366== by 0x5056358: sem_open (sem_open.c:269)

==69366== by 0x10AE3A: Minero (miner.c:669)

==69366== by 0x10C6AE: main (miner.c:1278)

==69366== Address 0x1ffefff99c is on thread 1's stack

==69366== in frame #1, created by sem_open (sem_open.c:141)

==69366== Uninitialised value was created by a stack allocation

==69366== at 0x5056024: sem_open (sem_open.c:141)