Proyecto final: *Miner Rush*

Autores:

Kevin de la Coba Malam

Marcos Aarón Bernuy

Grupo 2292, pareja 04.

Contenido

[Introducción 3](#_Toc71360738)

[Trabajadores 3](#_Toc71360739)

[Rondas de minado 4](#_Toc71360740)

[Bloques 4](#_Toc71360741)

[Red 4](#_Toc71360742)

[Bloques compartidos 5](#_Toc71360743)

[Concurrencia y semáforos 5](#_Toc71360744)

[Votación 5](#_Toc71360745)

[Monitor 5](#_Toc71360746)

[Problemas y errores 5](#_Toc71360747)

[Autoevaluación 5](#_Toc71360748)

# Introducción

El proyecto emula el funcionamiento de la blockchain en la red. La blockchain no es una estructura simple, es muy compleja por lo que emularla supone un reto. Para emularla de manera local se usa memoria compartida, señales, mensajes, tuberías… todo lo aprendido durante el curso.

Nuestra implementación cumple con todos los requisitos exceptuando el que implica una salida correcta de los mineros cuando uno recibe SIGINT se produce **interbloqueo**. Se ha intentado mitigar el efecto del interbloqueo, pero esto se explicará más adelante.

# Trabajadores

El primer paso en el proyecto es el de implementar trabajadores dentro de un minero. Mediante la librería <pthread.h>, se crean un número determinado de threads que buscan la solución. El padre de estos threads (el minero) se queda a la espera de que estos terminen ejecutando la llamada de pthread\_join.

Los trabajadores usan una estructura definida por nosotros, la estructura es la siguiente:

**typedef** struct **{**

int starting\_index**;**

int ending\_index**;**

long int target**;**

long int solution**;**

**}** worker\_struct**;**

Para hacer la ejecución de los trabajadores lo más eficiente posible la estructura contiene 2 índices:

* **starting\_index** se refiere al índice desde el cual el trabajador debe buscar una solución.
* **ending\_index** es el índice donde debe terminar de buscar la solución.

Por ejemplo: Digamos que el minero debe buscar la solución desde el número 0 hasta el número 999, para esta tarea también se especifican 4 trabajadores. Para paralelizar la tarea cada trabajador tendrá unos índices definidos, el primero tendrá desde 0 hasta 249, el segundo desde 250 hasta 499, el tercero desde 500 hasta 749, y el último desde 750 hasta 999. Esto paraleliza la ejecución de los trabajadores y hace que, a más trabajadores, más veloz el proceso de minado (siempre y cuando la CPU permita el número especificado de threads y tenga varios núcleos).

El campo **target** es común en todos los trabajadores ya que es el target que deben buscar, y el campo **solution** solo es modificado por el trabajador que encuentra la solución.

Una vez un trabajador encuentra una solución puede que los demás sigan buscando, para hacer que dejen de buscar se usa una variable global llamada **solution\_find**. Inicialmente se establece a 0 de modo que cada vez que un trabajador itere buscando una solución comprueba el valor de esta variable, si el valor es diferente de 0 significa que un trabajador ya ha encontrado una solución y por lo tanto debemos salir del bucle. Esta variable es global para todos los threads, e incluso el minero puede acceder a esta ya que está declarada al comienzo del código como **extern solution\_find**.

Para modularizar el código, todas las acciones relacionadas con los trabajadores están declaradas en los archivos **trabajador.c** y **trabajador.h**, también se incluye la declaración de la estructura.

# Rondas de minado

El segundo argumento que el minero recibe es el número de rondas de minado a ejecutar, si es 0 o menor se ejecuta indefinidamente. Para esto se crea una variable llamada **infinite** la cual, en caso de que el segundo argumento cumpa las ejecuciones especificadas, se pondrá a 1, obligando al loop a nunca acabar.

# Bloques

El siguiente paso en la implementación fue hacer uso de los **bloques**, la estructura en este caso ya se nos entrego hecha y no se hicieron modificaciones al respecto.

El manejo de los bloques se hace de forma dinámica, aunque se puede hacer de forma estática también (y ahorraría más de un dolor de cabeza). Para hacer un uso correcto y eficiente de los bloques se han definido varias **primitivas** las cuales se pueden consultar en **block.h,** ahí se encontrarán todas las primitivas.

El minero lo primero que hace es reservar memoria para un bloque, este bloque no tiene definidos ninguno de los campos necesarios. Como todavía no se ha implementado la red el bloque simplemente establece un target aleatorio entre los números 1-1.000.000. Se hace uso de una variable **last\_block** la cuál es un puntero que simplemente se usa para definir el bloque anterior al creado, como en el primer bloque este es null, no tiene gran relevancia. Digamos que ahora estamos con el segundo bloque, *last\_block* en este caso apuntará al bloque creado en la anterior iteración y al llamar a la función int block\_set**(**Block **\***prev**,** Block **\***block**)** el bloque nuevo se actualiza y el id es el id del anterior +1, las wallets se copian, el campo *prev* del nuevo bloque apunta a *last\_block*, el campo *next* del *last\_block* apunta al nuevo bloque y por último el nuevo target es la solución anterior.

Una vez el minero ha inicializado el bloque, este copia el target en la estructura usada por los threads. Una vez se encuentra la solución solo haría falta actualizar la solución del último bloque, cogemos la solución de la estructura del thread y la copiamos en el último bloque.

Para mantener la modularidad todo lo relacionado con el manejo y uso de bloques está en los archivos **block.c** y **block.h**. En este momento de la implementación no se usan bloques compartidos entre procesos, es todo de forma individual.

# Red

La red de mineros es la que permite que los mineros estén “conectados” entre ellos. Para el desarrollo de la red se ha usado la estructura ya proporcionada NetData. En una primera implementación la red solo conecta a los mineros, no significa que usen el mismo bloque todos y se ejecuten al mismo ritmo.

Para crear la red se usa memoria compartida, el primer minero en ejecutarse la crea, los demás se van uniendo a la red modificando pid\_t miners\_pid**[**MAX\_MINERS**];** int last\_miner**;** int total\_miners**;.** Estos campos también se modifican al salir de la memoria dinámica de forma que cuando el último minero salga dejara la variable **total\_miners** a 0, cuando llega a este valor debemos hacer unlink.

Para mantener la modularidad todo lo relacionado con el manejo y uso de la red de mineros está en los archivos **net.c** y **net.h**. La red en este momento no resulta muy útil pero más adelante se implementa la red de la forma esperada.

# Bloques compartidos

Tras haber implementado la red ya podemos usar bloques de forma compartida. Esto se implementa en los archivos **block.c** y **block.h**. El primer paso para hacer los bloques compartidos es el de definir una estructura con la memoria compartida necesaria:

**typedef** struct **{**

long int target**;**

long int solution**;**

int id**;**

int is\_valid**;**

int num\_miners**;**

int wallets**[**MAX\_MINERS**];**

**}** shared\_block\_info**;**

Esta es la estructura usada para definir el bloque compartido entre los procesos. Como se puede apreciar contiene todos los campos de un bloque exceptuando los campos *next* y *prev*, a parte se le añade el campo **num\_miners**.

# Concurrencia y semáforos

**typedef** struct **{**

int total\_miners**;**

int blocked\_loosers**;**

sem\_t net\_mutex**;**

sem\_t block\_mutex**;**

sem\_t mutex**;**

sem\_t vote**;**

sem\_t count\_votes**;**

sem\_t update\_blocks**;**

sem\_t update\_target**;**

sem\_t finish**;**

**}** Sems**;**

Esta es la estructura usada para definir los semáforos que emplearemos, además de dos campos que serán necesarios. Esta estructura y los semáforos definidos en ella serán necesarios para evitar concurrencia que se daba al acceder a la memoria compartida de tanto bloques como de redes como la de los mineros. En un principio únicamente teníamos *total\_miners*, *net\_mutex*, *block\_mutex*, y *mutex* los cuales son semáforos de exclusión mutua. Más adelante implementaremos el resto que serán necesarios para programar de forma correcta la votación.

Esta estructura y su implementación junto con la de las funciones que modifican cada elemento de la estructura está en los ficheros **sem.c** y **sem.h** en los cuales se inicializan todos los semáforos anteriormente mencionados y también serán eliminados cuando sea necesario empleando las respectivas funciones. En ellos además para evitar la repetición de código implementamos *sem\_down()* y *sem\_up()* funciones las cuales reducen el tener que realizar un control de errores cada vez que se emplee *sem\_wait()* y *sem\_post()* en otros ficheros.

# Votación

En esta sección en la que implementamos todos los aspectos relacionados con la votación finalmente implementamos el resto de la estructura de Sem.

El proceso de votación comienza cuando se detecta que hay un ganador, es decir, cuando un minero ha encontrado solución a la operación planteada. Nada más ver esto comprobamos si más de un minero ha encontrado solución y cambiamos el valor *solution* de la memoria compartida de mineros. Tras esto suspendemos el proceso para que espere a la señal SIGUSR2.

A partir de aquí los mineros seguirán dos caminos, el de ganador, o el de perdedor. El caso de perdedor ya estaba implementado y forma parte del proceso previamente explicado de los mineros. En el caso de que se trate de un minero ganador, lo primero de todo será actualizar la solución. Tras esto, recibirá el quorum es decir el número de mineros que participan en la votación y que se encuentra guardado en la memoria compartida de red. Después se actualiza el número de mineros activos, con la variable **total\_miners** de la estructura de semáforos, y en esta actualización se debe incluir el ganador también. Una vez ya está claro el número de mineros que participarán en la votación se envía la señal SIGUSR2 para permitir al resto de mineros comenzar la votación. Para la votación se realizan *ups* del semáforo **vote**. Este periodo de votación dura 2 segundos, tiempo el cual el proceso ganador se encuentra bloqueado habiendo utilizado el semáforo **count\_vote**, y tiempo que permitimos utilizando la función *timedwait(),* tras haber obtenido el tiempo actual con *clock\_gettime()*. Una vez la votación ha sido realizada con éxito se cuentan los votos empleando el **voting\_pool** de la memoria compartida de red. Cuando ya hemos actualizado este campo debemos comprobar que el bloque sea válido, para lo que tiene que haber más de la mitad de votos positivos. En caso de que si sea actualizamos los campos necesarios, y en caso de que no lo sea, se destruye el bloque. Cuando ya se ha hecho la comprobación se resetea la votación para futuras votaciones. Tras esto tenemos que permitir una vez más que cada minero actualice su bloque, para lo que bloquearemos el proceso ganador hasta que se hayan actualizado como previamente se ha explicado, pero en este caso se emplearán los semáforos **update\_target** y **update\_block.** Por último, antes de liberar el proceso de votación y liberar a los votantes, actualizamos el bloque compartido. Tras esto liberamos los recursos como ya estaba implementado y finaliza el proceso de votación.

# Monitor

Para la implementación del monitor en dónde también configuramos la cola de mensajes y realizamos la impresión de los resultados en *blockchain.log,* hemos creado los ficheros **monitor.c** y **monitor.h.** En ellos se encuentran las funciones y procesos necesarios para la monitorización de los mineros.

Antes de implementar el monitor se debe tener en cuenta que en el proceso de votación antes de liberar los recursos debemos añadir en el proceso que se envíe en caso de que haya un bloque válido, este bloque al monitor, y por tanto la implementación del monitor tendrá que ver con la gestión de dicho bloque.

El monitor lo implementamos de la siguiente manera. Creamos un proceso padre e hijo empleando la función fork(). Entre padre e hijo estarán comunicados por un pipeline por lo que el primer paso del proceso hijo será cerrar el extremo de escritura. Tras esto se abrirá el fichero *blockchain.log* para la escritura de todos los bloques en este cada 5 segundos.

De forma paralela el proceso padre se encarga de inicializar los semáforos, unir el monitor a la red, cerrar el extremo de lectura del pipeline, e inicializa el buffer circular de bloques, que será aquel que el proceso hijo imprimirá cada 5 segundos. También se encargará de crear la cola de mensajes.

Una vez hemos realizado todos estos pasos previos podemos centrarnos en la gestión del bloque tanto para proceso padre como para proceso hijo. En la gestión del proceso padre, se recibirá la instrucción mandada. \*\*\*\*\*\*\*\*\*\*\*\*\* **No se que instrucción recibe, creo que es la de comenzar votación** \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*. Además de esto, comprueba si el bloque recibido se encuentra ya en el buffer con la variable **is\_in**. En el caso de que en efecto si este, se realiza la operación y se imprime el mensaje correspondiente a la operación. En cambio, si no se encuentra, metemos el bloque en el buffer. Posteriormente realizamos una copia del bloque y la escribimos en el pipeline para que llegue al proceso hijo, pero únicamente si no se encuentra en el buffer. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* **Dice que lo envía si no está en el buffer, pero cuando comprobamos si ya está, en el caso de que no este lo metemos entonces no entiendo la distinción de si meterlo o no** \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*.

De forma paralela el proceso hijo se había quedado bloqueado esperando a recibir bloque por el pipeline y poder hacer la lectura correcta de este. Una vez lo ha leído correctamente, simplemente realiza una copia y lo guarda en la cadena dinámica.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* no se si habría que indicar algo más con los pipelines y posibles esperas que estos hagan y que no sepa \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Problemas y errores

# Autoevaluación

Hemos tenido debate entre nosotros para decidir cuál sería una autoevaluación correcta. Teniendo en cuenta que se ha implementado todo lo relacionado con la votación correctamente estaríamos en un 9 mínimo y si estuviera perfecto podríamos optar al 10, pero aquí empieza lo dudoso. El problema de implementación tiene que ver con los mineros y por tanto se puede considerar como un error que debería haberse solucionado, aunque cierto es que se trate del punto 15 de mineros por tanto depende mucho de la relevancia dada al error. Junto a esto debemos añadir posibles errores de implementación que no hayamos resuelto de la forma más eficiente. A parte de esto, creemos haber mantenido la modularidad, y no haber cometido fallos que se consideren penalizaciones. En definitiva, partiendo de un 10 reducimos por el error de SIGINT, además de posibles pérdidas de eficiencia y tenemos dos notas propuestas por cada uno de nosotros y la media de ambas resulta en un 9 sin contar con la posibilidad de aumentar dicha nota por una mejora que puede ser tanto por la autoevaluación realizada, como por que el programa muestre un rendimiento excelente.