Proyecto Final: *Miner Rush*

FECHA DE ENTREGA: ANTES DEL 10 DE MAYO
(HORA LÍMITE: 9 DE MAYO A LAS 23:59)

> "*There are three eras of currency: commodity based, politically based, and now, math based.*"
>
> Chris Dixon.

INTRODUCCIÓN AL PROYECTO

El objetivo de este proyecto es programar el software para poder crear una red de mineros de bloques inspirada en la tecnología **Blockchain**. La red se compone de un conjunto de procesos denominados ***mineros*** que tendrán como misión la resolución de una prueba de esfuerzo (*Proof of Work*, POW). Todos los mineros tendrán que intentar resolver de forma concurrente la misma prueba, y solo uno, el primero en encontrarla, recibirá la recompensa: una moneda.

Una prueba de esfuerzo o POW consiste en resolver un reto matemático, habitualmente hallar el operando que permite obtener un determinado resultado tras aplicar una función hash no invertible, es decir que la única forma de encontrar el operando es por fuerza bruta. Se va a implementar una red distribuida en la que cada nodo trabaja de forma independiente, y al encontrar una solución debe recibir la aprobación de al menos la mitad del resto de nodos para considerar que ha sido aceptada por la red. La solución se incorpora en un contenedor de información denominado ***bloque*** y que guarda una cierta relación con los bloques anterior y posterior, de manera que formen una lista doblemente enlazada. Cada minero tendrá por tanto una copia idéntica de la cadena de bloques o al menos de segmentos de la misma.

El proyecto se inspira en los conceptos utilizados por los sistemas basados en Blockchain, pero con diferencias importantes. Se puede consultar información sobre el funcionamiento del sistema Blockchain más implantado, el Bitcoin, en este enlace.

DESCRIPCIÓN GENERAL

La red se compone de un conjunto de procesos denominados ***mineros***. Cada minero es un proceso multihilo que se ejecuta de forma paralela. A cada uno de estos hilos se le denominará ***trabajador***. Por tanto un minero tiene varios trabajadores minando a la vez para encontrar la solución del problema. Cuando uno de los trabajadores encuentre la solución lo notificará al resto de nodos (mineros) de la red para que verifiquen su trabajo y puedan votar en consecuencia. El número de mineros será variable y dinámico, es decir que en un determinado momento puede haber 4 mineros, y en otro puede haber 6. La red debe ser capaz de incorporar dinámicamente nuevos mineros y adaptarse a la situación si alguno de ellos causa baja.

La implementación de los mineros se hará mediante un programa C. Además se implementará otro programa, denominado monitor, que permitirá a los distintos mineros enviar datos informativos sobre su ejecución mediante la creación de un log único para toda la red. Los bloques contendrán, entre otros, los siguientes campos:

- `target`: Resultado del que se desea hallar el operando.
- `solution`: Operando que permite hallar el target.
- `wallets`: Registro de monedas conseguidas por cada minero.

Para simplificar la codificación, se permite suponer que el número máximo de mineros que van a participar en el proceso de minado de una red nunca excederá una determinada constante denominada `MAX_MINERS`. De esta forma, el fichero `miner.h` incluirá la siguiente definición:

```
typedef struct _Block {
    int wallets[MAX_MINERS];
    long int target;
    long int solution;
    int id;
    int is_valid;
    struct _Block *next;
    struct _Block *prev;
} Block;
```

La estructura anterior puede ampliarse si se desea. Los campos `next` y `prev` pueden no utilizarse, pero no se permite eliminar campos o cambiar el nombre de la estructura.

La lógica general del sistema será la siguiente:

1. Registrar minero en la red.
2. Preparar y lanzar ronda de minado.
3. Minar.
4. Comunicar ganador y actualizar la cadena de bloques.
5. Lanzar nueva ronda de minado.

El registro en la red de un minero consiste básicamente en añadir el PID del proceso a una lista de mineros que se encuentra registrada en memoria compartida. Esta lista es la que sirve para que cada minero sepa de la existencia del resto. En caso de que no haya ningún minero anterior, es decir que el proceso lanzado sea el primer minero, se deberá crear la memoria compartida. Los siguientes mineros se unirán a ella utilizando el mismo nombre, definido en `miner.h` como `SHM_NAME_NET`:

```
#define SHM_NAME_NET "/netdata"
```

Para ello, se sugiere incluir la siguiente definición para la estructura de memoria compartida:

```
typedef struct _NetData {
```

```
    pid_t miners_pid[MAX_MINERS];
    char voting_pool[MAX_MINERS];
    int last_miner;
    int total_miners;
    pid_t monitor_pid;
    pid_t last_winner;
} NetData;
```

En particular:

- `miner_pid` contiene un listado con los PIDs de los mineros activos.
- `voting_pool` puede usarse para registrar los votos de cada minero.
- `last_miner` puede usarse para indicar la posición del último minero activo.
- `total_miners` indica la cantidad de mineros activos [1].
- `monitor_pid` y `last_winner` recogen el PID del proceso monitor (se usará un valor no válido para indicar que no hay monitor activo) y el del último ganador, respectivamente

La preparación de la ronda de minado persigue que todos los elementos necesarios estén correctamente inicializados antes de que se inicie la ronda. Dicha ronda se iniciará de forma sincrónica, es decir, todos los procesos recibirán una notificación de inicio "al mismo tiempo". Se considera que notifica "al mismo tiempo" cualquier implementación que notifique de manera secuencial, por ejemplo un bucle con envío de señales o modificaciones de semáforos.

El minado consistirá en buscar la solución a la prueba de esfuerzo.

La comunicación de ganador se realizará enviando una señal `SIGUSR2` al resto de mineros. Por tanto, recibir una señal `SIGUSR2` implica que el minero se considere perdedor en esa ronda de minado. En caso de que la solución sea correcta, deberá actualizar su cadena de bloques creando un nuevo bloque con la información compartida por el ganador en memoria compartida, en el segmento de nombre `SHM_NAME_BLOCK`:

```
#define SHM_NAME_BLOCK "/block"
```

Una vez que todos los mineros hayan actualizado su cadena de bloques, el minero ganador organizará una nueva ronda de minado que se iniciará de manera sincrónica, como se ha indicado antes. Una vez lanzada la autorización de inicio, el orden de ejecución de la ronda de minado dependerá del sistema operativo (no previsible) hasta que uno de los trabajadores encuentre una solución.

REQUISITOS

Minero

1. Lanza una cierta cantidad de hilos determinada en su primer parámetro de entrada. El hilo principal del programa no realiza la tarea de minado (buscar la solución del reto matemático), delegando la misma de forma exclusiva a los hilos creados.
2. Acepta dos parámetros de entrada que deberán ser los siguientes, respetando el orden:
   Arg 1: El número de trabajadores del minero.

[1] Esta implementación permite que mineros se den de baja y por tanto que haya posiciones en `miners-pid` "quemadas". Esta aproximación tiene una limitación obvia de escalado y se podrá optar por utilizar otro tipo de implementación dinámica más complicada que permita crecer sin limitaciones. No obstante, para el proyecto bastará con suponer que nunca se van a crear más de `MAX_MINERS` mineros.

Arg 2: El número de rondas de minado a realizar. En caso de que el número de rondas de minado sea cero o negativo, el minero continuará indefinidamente hasta recibir la señal de detención.

3. El reto matemático viene definido en el archivo `miner.c` (proporcionado con el enunciado) mediante la función `simple_hash`. Esta es la función hash que debe utilizarse en el proyecto.
4. Cuando un trabajador (hilo) encuentre una solución al problema deberá notificarlo a todos los trabajadores, tanto del propio minero como al resto de mineros. La comunicación a los trabajadores del propio minero (compañeros) se podrá implementar como se desee, pudiendo utilizarse una variable global. Para comunicar al resto de mineros que se ha encontrado una solución se deberá usar la señal `SIGUSR2`.
5. Cuando un minero recibe la señal `SIGUSR2`, sus trabajadores deberán detener su tarea de minado. El minero verifica si la solución encontrada es correcta y vota a favor o en contra de la misma de manera consecuente.
6. Para compartir la solución con el resto de mineros, se utiliza memoria compartida. Se establecerá un mecanismo de sincronización de forma que el minero ganador sea el único que escriba la solución en memoria compartida durante una determinada ronda de minado. El resto de mineros accederá a la misma una vez la solución se haya establecido. La tarea de los mineros perdedores es verificar si la solución es correcta para el objetivo fijado y votar en consecuencia.
7. La votación se realiza una vez por ronda de minado, después de que un ganador haya enviado la señal `SIGUSR2`. Para votar, los mineros pueden utilizar un array de voto guardado en la memoria compartida (en la estructura sugerida se denomina `voting_poll`). Si el minero considera que el voto es positivo su posición del array tendrá el valor `1`, si considerá que la solución no es válida el valor será `0`. Se podrá usar un valor distinto para marcar que aún no se ha registrado ese voto.
8. Para saber si hay mayoría de votos, el minero ganador de la ronda previa comprueba el *quorum* de votos, es decir la cantidad de mineros que van a participar en la votación. Se considera que un minero está activo si el envío de una señal al PID del minero tiene éxito. Para conocer el *quorum* el minero ganador envía la señal `SIGUSR1`, y analiza el resultado de la llamada a `kill` en cada caso. La cantidad de mineros activos se comparte en la red mediante la actualización de memoria compartida. Se sugiere el uso del campo `total_miners`.
9. Con el *quorum* actualizado, el nuevo ganador envía la señal `SIGUSR2` para iniciar el proceso de voto.
10. El minero ganador cuenta los votos para saber si su propuesta ha sido aceptada. En caso de serlo da valor `1` al campo `is_valid` del bloque compartido en memoria. Esta variable compartida se usa de manera que los perdedores sepan si la votación ha sido positiva o negativa. Es el minero ganador quien debe modificarla.
11. En caso de que el bloque sea aceptado (válido) todos los mineros lo incorporan a su cadena de bloques propia.
12. Una vez que todos los mineros han incorporado el bloque (si ha sido aceptado), el minero ganador prepara un bloque nuevo que tendrá como objetivo (*target*) la solución del anterior (último bloque añadido a la cadena de bloques). En el caso del primer minero de la red, actuará como el ganador de la ronda anterior, y será el encargado de preparar la siguiente ronda.
13. En caso de que haya un proceso monitor activo, los mineros intentan enviar a través de una cola de mensajes el nuevo bloque creado por cada uno al programa monitor. Si es un minero ganador la prioridad será `2`, si es un minero perdedor la prioridad será `1`.
14. Se debe implementar una función que imprima en un fichero identificado por su PID la cadena de bloques propia del minero antes de acabar su ejecución.

15. El minero acabará su ejecución al recibir la señal `SIGINT`, o cuando haya realizado el número de rondas indicado en el tercer parámetro de entrada.

Trabajador

1. Cada hilo trabajador recibe del hilo principal una estructura con al menos el objetivo de la POW.
2. Esa misma estructura se puede utilizar para devolver datos al hilo principal.
3. Cada ronda de minado tendrá un conjunto de trabajadores nuevo. Por tanto, al acabar el trabajo los hilos deben finalizar.

Funcionamiento de la Red

1. La red se compone de un conjunto de procesos minero y opcionalmente un proceso monitor (se debe implementar este programa, pero no es necesario para que la red funcione con normalidad).
2. Estos procesos se pueden lanzar desde una misma shell, o desde shells diferentes.
3. Una vez se ha lanzado el primer minero, cualquier otro minero lanzado se unirá a la red.
4. Se considera que un minero se ha unido a la red cuando se junta (*attach*) a los segmentos de memoria compartida con la información de la red y del bloque compartidos.
5. En la información de red se pueden compartir elementos de sincronización como semáforos.
6. La estructura compartida con información de red tendrá al menos un campo con los PIDs de todos los procesos activos (*attached*) que se denominará `miners_pid`.
7. Los mineros pueden unirse en cualquier momento.
8. Los mineros pueden acabar y salir de la red en cualquier momento.
9. La información sobre la red debe adaptarse de forma dinámica a los cambios anteriores. Uno de los mineros, el ganador se encarga de actualizar los datos de red compartidos.
10. El funcionamiento de la red será robusto frente a altas o bajas de mineros, en particular se prestará atención a la validez de la información compartida y a evitar bloqueos (*deadlocks*).
11. Si se considera necesario para evitar bloqueos provocados por la ausencia de alguno de los mineros, se podrán establecer mecanismos de *timeout* de 2 o 3 segundos. Una forma sencilla de hacerlo es mediante el uso de `alarm`. Si un minero está esperando a empezar una ronda, pero pasan más de 2 segundos sin recibir la señal de inicio puede deberse a que el minero ganador de la ronda anterior ha sufrido un problema. Al recibir la alarma puede detectar este problema y terminar.

Monitor

1. Crea un proceso hijo.
2. Recibe de los mineros mensajes con los nuevos bloques.
3. El proceso padre dispone de un buffer de memoria para los 10 últimos bloques. Tras recibir un nuevo mensaje, comprueba si el identificador del bloque recibido ya está en el buffer. Si es así, comprueba si la solución y el objetivo coinciden. En caso afirmativo, imprime por pantalla `"Verified block X with solution Y for target Z"`, donde `X`, `Y` y `Z` son los valores correspondientes. En caso de error imprime el mensaje `"Error in block X with solution Y for target Z"`. Si el identificador no se encuentra en el buffer, se incorpora eliminando el más antiguo.
4. El proceso padre enviará por tubería una copia de cada nuevo bloque al proceso hijo.
5. El proceso hijo mantendrá una cadena con todos los bloques recibidos del padre.

6. El proceso hijo imprimirá en un fichero la cadena de bloques completa cada 5 segundos.
7. El proceso monitor finaliza al recibir la señal `SIGINT`. Padre e hijo finalizan de manera ordenada.
8. Se podrá reiniciar el proceso monitor mediante una nueva llamada de ejecución. El proceso monitor retomará el trabajo.

EVALUACIÓN Y RÚBRICA

Se valorará que el código esté documentado y estructurado de forma modular. Se recomienda encarecidamente que el proyecto se desarrolle de forma incremental, es decir, que se vayan añadiendo funcionalidades sobre un programa que funcione correctamente. Para aprobar **no** es necesario implementar todos los programas con todos los requisitos. Se sugiere la siguiente guía de autoevaluación.

Calificación Base

La calificación base se puede calcular según el nivel de desarrollo elegido. Cada nivel muestra un techo de la calificación base en función de la funcionalidad implementada. La calificación exacta dependerá de la claridad del código, los algoritmos utilizados, control de errores, nivel de modularidad, etc. El estudiante puede valorar estos puntos de igual manera y comentarlos en la memoria a la hora de auto-asignarse una calificación.

Los niveles de desarrollo, y el techo de la calificación, son:

Nivel	Descripción	Máximo
Nivel 1	Hay un minero multihilo capaz de crear una cadena de bloques.	2,0 ptos.
Nivel 2	Sobre el programa anterior, se ha añadido la comunicación con el monitor que imprime correctamente la información.	3,0 ptos.
Nivel 3	Sobre el programa anterior, se han añadido los mecanismos básicos para tener una red de mineros, aunque no tiene sistema de votación.	6,0 ptos.
Nivel 4	Sobre el programa anterior, se han añadido los mecanismo de votación pero sin cumplir con todos los requisitos del proyecto.	9,0 ptos.
Nivel 5	El proyecto cumple con todos los requisitos indicados en este documento.	10,0 ptos.

Modificadores

Sobre la calificación base se aplicarán los modificadores detallados a continuación. El estudiante puede justificar en la memoria su elección y el nivel de seguridad que tiene sobre la misma.

Descripción	Modificador
La funcionalidad que se afirma que cumple el programa funciona solo a veces.	Hasta un 50 % del máximo.
La funcionalidad que se afirma que cumple el programa funciona en más del 90 % de las ejecuciones, o solo falla en condiciones muy concretas e identificadas.	Hasta un 75 % del máximo.
La funcionalidad que se afirma que cumple el programa funciona siempre, y se aporta documentación teórica y/o pruebas que lo demuestran.	Hasta un 120 % del máximo.

Penalizaciones

La calificación se puede ver también afectada por las siguientes penalizaciones:

Descripción	Penalización
El programa da siempre error de segmentación (*segmentation fault*).	Calificación a 0,0 ptos.
El programa da en ocasiones error de segmentación.	Hasta −2,0 ptos.
El programa no libera correctamente los recursos.	Hasta −2,0 ptos.
El programa no tiene ninguna o casi ninguna modularidad.	Hasta −1,0 ptos.
El programa tiene una falta muy relevante de control de errores.	Hasta −1,5 ptos.

Mejoras

La calificación puede verse beneficiada por las siguiente bonificaciones:

Descripción	Bonificación
Se han implementado mejoras sugeridas en el enunciado o propuestas por el estudiante.	Hasta 1,0 ptos.
La memoria se ha estructurado como una autoevaluación del proyecto, donde se justifica la nota base que se auto-asigna, los modificadores y si procede la aplicación de penalizaciones especiales y adición de mejoras sobre los requisitos planteados en el enunciado (la bonificación dependerá de la relevancia, detalle y corrección de la memoria).	Hasta 1,0 ptos.
Se han implementado (casi) todos los requisitos principales (sistema de votación incluido) y el programa demuestra un rendimiento excelente (velocidad de ejecución) según las condiciones de prueba que fije el profesor en su ordenador.	Hasta 1,0 ptos.