Eduardo Terrés Caballero
Gregorio Martínez Blázquez

PROYECTO FINAL

Dinámica seguida

En primer lugar, se bloquean la señal *SIGINT* durante la apertura de los diferentes recursos (memoria compartida, semáforos …), de forma que se mantiene un control respecto a los procesos que mantienen abierta la memoria y se evita que un proceso pueda morir en medio de la espera para un semáforo *mutex*. Así pues, hasta haber inicializado todos los recursos, un proceso no podrá finalizar mediante *SIGINT*.

La fase de apertura e inicialización de recursos sigue una lógica tal que primero se intenta crear la memoria compartida y se inicializan los campos. En caso de ya estar creada, tan solo se abre. En el caso de la cola de mensajes, se permite continuar con la ejecución, aunque la apertura hubiese sido fallida, aunque teniendo en cuenta que se pierde la funcionalidad de monitor. Una vez reservada memoria para diferentes vectores de variables para la gestión de los hilos, así como de la estructura para la introducción de parámetros durante la creación de los hilos, se bloquean las señales *SIGUSR2 y SIGINT* de cara a los ciclos, de forma que puedan ser tratadas de forma síncrona y predecible.

Respecto a este apartado, consideramos que hemos hecho un buen trabajo de diseño, aunque no se trate de la parte complicada. El control de errores se encuentra modulado en diferentes funciones de liberación de memoria. f*reeAll* cierra los recursos introducidos por parámetros; *unlink_and_unmap* implementa la lógica que permite que solamente el último proceso realice la llamada a *unlink*, de forma que las bajas y altas de mineros es consistente, durante el mismo periodo de minado, así como para posteriores ejecuciones. Cabe destacar que los semáforos *mutex* empleados son semáforos con nombre, mientras que los semáforos empleados para la sincronización de procesos en los turnos son semáforos sin nombre, albergados en la memoria principal. Se ha hecho uso de señales principalmente

Respecto a la lógica seguida durante las rondas, una vez inicializado el bloque de la nueva ronda, el minero añade su *PID* a la red, y comienza la minería con la creación de los hilos. El proceso queda entonces en estado de *sigsuspend*, a la espera de la señal *SIGUSR2*. Como todos los mineros tratarán de minar una misma función hash, que requiere poca capacidad de computación, ya que no se requieren requisitos adicionales de número de ceros en la solución – como implementan blockchains como la de *Bitcoin* -, es frecuente que un minero perdedor reciba dos veces la señal *SIGUSR2*. Esto se debe a que, entre que el ganador notifica al resto de mineros, y la cancelación de los hilos, los trabajadores del perdedor encuentran la solución. Es esta incertidumbre respecto al número de señales – 1 o 2 – y el momento en que se reciben uno de los problemas más complicados que hemos encontrado durante el desarrollo del proyecto. La solución que hemos dado consiste en sincronizar todos los procesos, de forma que los perdedores pueden desechar la posible señal recibida por uno de estos dos agentes.

En rasgos generales, este problema se ha tratado transaccionando la lógica hacia una basada en semáforos, en vez de señales, que permiten llevar a cabo una lógica más robusta, donde no hay

pérdidas de señales por *merge*, y se reduce sustancialmente el número de interbloqueos terminados por alarmas, en vez de por el correcto funcionamiento de la red.

Nuestra perspectiva respecto a la solución implementada es que nos habría gustado poder mantener la implementación inicial – enteramente con señales, a excepción de semáforos *mutex* -. No obstante, aunque pudiese funcionar durante el ciclo normal de ejecución, se trataba de una opción muy sensible ante imprevistos y mineros piratas. En definitiva, hemos optado por hacer uso de semáforos para la sincronización entre procesos, haciendo uso de señales únicamente en los momentos indicados en la memoria.

El funcionamiento del sistema de rondas es reflejo del orden indicado en la memoria, con sincronización en aquellos lugares que la precisa, - el minero ganador espera a que los perdedores copien el bloque, los perdedores esperan a que el ganador cree un nuevo bloque… -. La gran mayoría de las esperas de semáforos que no son de exclusión mutua incluye un mecanismo de *timeout* de 3 segundos para poder recuperarse de interbloqueos con la correcta finalización del proceso. Los semáforos de exclusión mutua no han precisado espera con tiempo, ya que la señal de salida se encuentra bloqueada y únicamente se comprueba al inicio de cada ronda. Así pues, los mineros están obligados a terminar la ronda, para poder mantener la integridad de la red. Es por esto que los mecanismos de *timeout* son, a priori, caminos de ejecución imposibles si todos los mineros envían las señales e interactúan con los semáforos de la forma prevista. Permiten entonces que la red se desintegre de forma ordenada y liberando la totalidad de la memoria en caso de que se caiga un minero por una causa ajena.

Los bloques se han implementado como una lista enlazada, de forma que se reserva memoria para un bloque al inicio de cada ronda, como ya se ha mencionado. Una vez los mineros finalizan sus rondas de trabajo, imprimen los bloques en un fichero.

Monitor

El monitor únicamente puede acceder a la red si hay un minero en ella. De esta forma, evitamos que pueda haber un monitor sin que haya red abierta. Por otra parte, los mineros comenzarán a enviar los bloques al monitor una vez este se haya introducido en la red, evitando saturar la cola de mensajes con los diez primeros bloques y rompiendo la integridad referencial ente bloques (de la lista enlazada), ya que se podrían recibir 10 bloques y, una vez se libere la cola, indexar más bloques de otra parte de la cadena.

Una vez inicializados y abiertos los recursos, el programa sigue la lógica descrita, siendo el hijo quien imprime en el fichero y el padre quien filtra los bloques que recibe de la cola.

Cabe destacar que consideramos que cuando un bloque falla, se debe incluir en la red, aunque se repetirá el mismo *target* en la siguiente ronda, permite entonces mantener un registro de los bloques fallidos.

Funcionalidad adicional – Expulsión de piratas

Como la prueba de trabajo consiste en la evaluación de una función determinista, es posible detectar piratas que actúan con la misma lógica que el resto de programas, pero no hacen uso de la misma

función hash, o directamente no tienen función hash. Como la red es muy débil frente a mineros piratas, ya que con un más de un 50% de los votos pueden crear bloques incorrectos, el ganador será el encargado de matar a los procesos que voten en contra de una solución válida.

Autoevaluación

En cuanto a calificación base, hemos implementado tanto la red de mineros con el sistema de votación como el monitor con todos los requisitos pedidos en el *PDF* de la práctica. Consideramos que el nivel de modularidad es bueno, no pensamos que sea gran cosa, pero aceptable, y que hemos implementado un amplio control de errores, aunque este ha podido afectar a la claridad del código puesto que una considerable cantidad de este es código de errores. Para compensar este hecho hemos tratado de comentar el código de manera aclaratoria para no perderse con el control de errores.

Tras obtener el código que considerábamos necesario fuimos ejecutando y probando nuestro proyecto continuamente, arreglando los errores que detectamos y consideramos que nuestro código está muy por encima del nivel "el programa funciona solo a veces". Además, creemos hemos superado también el nivel "el programa funciona en más del 90% de las ejecuciones, o solo falla en condiciones muy concretas e identificadas" pues hemos llegado a un punto en que nuestras pruebas, incluyendo también mineros piratas (entendidos como que un minero pirata puede ser un minero cuya función hash está cambiada para que produzca valores aleatorios por ejemplo) que tratan de cargarse la red de mineros, eran pasadas correctamente y con la ejecución que esperábamos acorde con los requisitos.

En cuanto a las penalizaciones esperamos no recibir ninguna pues estamos convencidos, por nuestro control de errores y numerosas ejecuciones de prueba tratando de generar errores, que nuestro programa no da errores de segmentación, libera los recursos adecuadamente (le hemos pasado *valgrind* tanto al monitor como a los mineros para comprobar que no da ni errores ni se filtra memoria) y por supuesto, pensamos que no tenemos una falta relevante de control de errores.

Hemos tratado de implementar algunas mejoras o soluciones algunos problemas que no se han llegado a plantear: hemos añadido que para evitar problemas con posibles mineros piratas que alteren la función hash, como se explica en el apartado de mejora adicional. Por simplificar el análisis de la ejecución hemos puesto que al imprimir el proceso creado por el monitor en un fichero los bloques recibidos por la cola de mensajes no impriman aquellos que son ceros. En la memoria, a parte, de explicar los aspectos más importantes, hemos añadido esta sección de autoevaluación para tratar de obtener la bonificación extra. No hemos querido abusar del uso de espera en los semáforos para obtener lo que consideramos un mejor código, pero es evidente que si un proceso muere por una causa externa en medio de un semáforo *mutex*, la red se quedaría en estado de interbloqueo. Se trata de un aspecto que admite margen de mejora.

Conclusiones

La conclusión principal respecto al trabajo realizado es que no es perfecto. De forma segura, hay aspectos que se podrían mejorar sobre el diseño del proyecto, de forma que fuera imposible que la red llegase a alguna situación de fallo. Aunque hayamos hecho una gran cantidad de pruebas, existirán caminos de ejecución muy concretos específicos, en el que no hayamos pensado y que provoquen que la red se desintegre. Dado este caso, tan solo podemos facilitar una correcta finalización de los mineros tras los *timeouts*. No consideramos que hayamos hecho un programa sin bugs o errores, pero podemos afirmar que los que hemos detectado los hemos tratado de solucionar.