Sistemas Operativos. Práctica 3.

Pablo Cuesta Sierra y Álvaro Zamanillo Sáez

Índice

Calificación base

Nivel 1: Minero multihilo

Para este primer nivel hemos diferenciado dos partes. En primer lugar iniciar correctamente los segmentos de memoria compartida del bloque y la red, y por otro lado, la creación de los hilos y su correcta terminanción.

En la red, hemos incluido más atributos (para siguientes niveles) y es el primer minero el que se encarga de inicializarlos correctamente. De la misma forma, si el minero crea el segmento que corresponde al bloque, configura aleatoriamente una "primera solución", que se usará para generar el primer *target* (y, por ende, el resto).

Por otro lado, para lo relativo a los hilos, alocamos un *array* de `pthread_t` e iniciamos la estructura de minado que cada trabajador (hilo) va a recibir. Esta estructura solo se inicializa una vez pues es común para todas las rondas y se libera cuando el minero abandona la red.

```
typedef struct Mine_struct_{
    long int target;
    long int begin; //índice por el que empieza a buscar el hilo
    long int end; //índice hasta el que el hilo busca (no incluido)
} Mine_struct;
```

En esta versión con un solo minero no se produce votación puesto que el *quorum* siempre resulta en 1 minero activo. Por lo tanto, el minero siempre se considera ganador y añade el bloque minado a su propia cadena. La ronda de minado finaliza cuando un trabajdor encuentra la solución y manda una señal a su proceso para que salga de la función bloqueante `sigsuspend`. Todas las señales tratadas tienen el mismo manejador el cual solo pone a 1 la variable de tipo `sig_atomic_t` que indica que señal se ha recibido. Para distinguir si se sale de `sigsuspend` siendo ganador o perdedor, el trabajador que encuentra la solución manda a su proceso `SIGHUP` (el resto de mineros recibirán `SIGUSR2`).

Los trabajadores de cada minero acaban cuando la variable *end_ threads* vale 1. Esta variable se modifica cuando un hilo trabajador encuentra la solución o cuando el *main* despierta tras recibir `SIGUSR2` de parte del ganador. El *main* hace una llamada a *pthread_ join* por cada hilo lanzado para asegurar que no hay pérdidas de memoria. La siguiente imagen muestra una ejecución de un minero con 1000 trabajdores y el análisis de Valgrind, mostrando que no hay pérdidas de memoria.

```
pablo@ub-tp:~/repos/SOPER_p1/p4/src$ valgrind ./mr_miner 1000 2
==2321896== Memcheck, a memory error detector
==2321896== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2321896== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==2321896== Command: ./mr_miner 1000 2
==2321896==
miner:2321896-remaining rounds:1
miner:2321896-remaining rounds:0
destroy everything
==2321896==
==2321896== HEAP SUMMARY:
==2321896==     in use at exit: 0 bytes in 0 blocks
==2321896==   total heap usage: 2,010 allocs, 2,010 frees, 606,353 bytes allocated
==2321896==
==2321896== All heap blocks were freed -- no leaks are possible
==2321896==
==2321896== For lists of detected and suppressed errors, rerun with: -s
==2321896== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figura 1: 100 trabajadores con Valgrind

Al finalizar las rondas de minado se imprime la cadena de bloques en un fichero para lo cual hemos usado como base la función proporcionada cambiado `printf` por `dprintf`. Por último,liberan todos los recursos con las respectivas funciones de *free* y *unlink*.

Nivel 2: Comunicación con el monitor

Para la comunciación con el monitor, se tiene que crear una cola de mensajes (a la que todos acceden abriéndola con su nombre). El primero en llegar la crea (sea minero o monitor), y los demás simplemente la abren.

Además, el primero que llega, también crea los segmentos de memoria compartida e inicializa los campos a su respectivo valor inicial. Para evitar que dos procesos accedan a la vez en su escritura inicial, hemos tenido que usar un semáforo con nombre que impida el acceso simultáneo de dos procesos a la estructura compartida. Este *mutex* es el único semáforo con nombre que hemos utilizado en el proyecto. Lo utilizamos para proteger escrituras simultáneas en los segmentos de memoria compartida.

El único minero que hay por ahora, al declararse ganador, envía por la cola de mensajes su nuevo bloque.

El monitor, como pone en los requisitos del enunciado, no comprueba que la solución sea la correcta (con la función *hash*), sino que comprueba que si tiene un bloque con ese mismo *id*, el bloque recibido tenga la misma solución y objetivo. Por este motivo, y como los mineros solo mandan los bloques que se han votado como correctos, no se da nunca el caso de que el monitor reconozca un "bloque erróneo".

El monitor (proceso padre) le manda a su hijo los bloques nuevos. Para mandar el número de *wallets* que el hijo imprime, hemos hecho uso del campo `is_valid` del bloque que se escribe en la tubería, que no tiene ninguna utilidad en este caso, ya que el monitor solo recibe bloques válidos y, de todas formas, el hijo del monitor nunca utiliza este campo.

Debido a que tanto el padre como el hijo pueden recibir señales durante la escritura o lectura en la *pipe*, hemos utilizado el siguiente algoritmo, que asegura la lectura o escritura completa de la información y el manejo de las señales en cada caso (sería similar en el caso de la escritura, cambiando *read* por *write*):

```
do
{
    size_read = read(fd[0], ((char*)block) + total_size_read, target_size - total_size_read)
    ;
    if (size_read == -1)
    {
        if(errno == EINTR && got_signal)
            handle_signal();
        else if(errno != EINTR)
            handle_read_error();
    }
    else if (size_read == 0)// se ha leído EOF
        return total_size_read;
    else
        total_size_read += size_read;
} while (total_size_read != target_size);
```

En el caso del hijo, `handle_signal()` lo que hace es poner una nueva alarma de 5 segundos e imprimir toda la cadena en su fichero.

En el caso del padre, cuando termina la lectura y ha recibido `SIGINT`, libera todo, cerrando la tubería y termina después de que su hijo haya finalizado.

Al terminar, si el minero o el monitor ve que ya no hay más procesos enganchados a la memoria compartida, se encarga de borrar (*unlink*) los segmentos de memoria compartida, el *mutex* y la cola de mensajes.

Nivel 3: Red de mineros sin votación

En esta primera versión sin votación nuestro objetivo fue asegurar que los mineros empezasen las rondas a la vez, parasen de minar cuando otro minero hubiese encontrado la solución e impedir que dos mieros se declaren ganadores en una misma ronda. Los mineros que ingresan a la red deben esperar a que finalice la ronda actual, es decir, hasta que el ganador de la ronda les de paso a la siguiente ronda.

Tenemos que tener en cuenta que si el ganador de una ronda se va a salir (ya sea porque esta es su última ronda o porque ha recibido `SIGINT`), en la siguiente ronda la red pueda seguir funcionando. Para esto, cada

minero, si es su última ronda, 'quita'[1]su *pid* (poniendo un valor negativo) del array de *pids* y en caso de ser ganador, le cede el puesto al primer minero que encuentra (con *pid* positivo).

Para que esta medida sea efectiva, los mineros tienen que pararse al principio del bucle, hasta que todos los demás hayan terminado la anterior iteración (por si el ganador de la anterior ronda tiene que designar un ganador que prepare la ronda por él). Para esto, hemos puesto un mecanismo de *turnstile*[2](cada minero, al principio del bucle, hace un *down* y, seguidamente un *up* del semáforo que hemos llamado `sem_round_end`). El semáforo está inicializado a 1 y el ganador de la ronda, antes de señalizar la finalización del recuento de votos, lo pone a 0 (con un *down*) para 'bloquear' la entrada de mineros a la siguiente ronda. El último minero en finalizar la ronda hace *up* del semáforo para permitir el paso a los demás.

Nivel 4: Red de mineros con votación

En este nivel aseguramos además las siguientes restricciones de sincronización: los mineros perdedores no pueden empezar a votar hasta que el minero ganador haya subido la solución a memoria compartida y el minero ganador no puede empezar el recuento hasta que todos hayan votado.

Para la parte de la votación hemos usado dos semáforos. El primero `sem_start_voting` obliga a los perdedores a esperar a que el ganador haya preparado el bloque para la votación (subir la solución). Este semáforo anónimo (se encuentra en la memoria compartida de la red), es *n-ario* y el ganador hace up tantas veces como votantes haya (número que ha sido calculado haciendo recuento del *quorum* anteriormente). Por otro lado, el ganador espera al semáforo `sem_end_votation` para hacer el recuento. Este semáforo solo toma valores 0 o 1, y es el último minero perdedor el que hace *up* del semáforo tras votar. Finalmente, los perdedores han de esperar al resultado de la votación. Esta condición de sincronización se resuelve igual que la de esperar para comenzar a votar; en este caso usamos el semáforo `sem_scrutiny`.

En general la idea usada para resolver la sincronización de la votación es reducirlo al caso de dos procesos: un proceso ganador y un proceso perdedor (salvo que este proceso es en realidad un grupo de procesos). De esta forma se trata de un problema de *leader & follower*. Los semáforos del ganador toman valores 0 o 1, mientras que los del grupo perdedor, son semáforos que toman valores (posiblemente) de 0 a n.

Nivel 5

El último requisito que nos faltaba era poner una espera con tiempo máximo en los semáforos, que tenemos ahora para que si un minero se queda pillado en algún punto o termina inesperadamente, el resto deje de esperar y termine. Para esto, hemos usado `sem_timedwait`.

En algunos casos, cuando un minero era muy lento y tardaba mucho (por ejemplo, ejecutando un minero con *valgrind*). Cuando los mineros se daban cuenta de esto y terminaban su ejecución, muchas veces los semáforos quedaban con valores incorrectos que impedían el correcto funcionamiento de la red. Para evitar esto, usamos una variable en la memoria compartida que cuenta el número de mineros activos, de modo que si un minero va a liberar y se da cuenta de que es el último minero activo, destruye todos los semáforos, para que el primer minero que llegue los inicialice correctamente.

Mejoras

Un único monitor activo

La primera mejora introducida es impedir que haya dos monitores activos. Si la red cuenta con un monitor y se intenta lanzar otro, este segundo proceso finalizará. La implementación de esta mejora se reduce a un bloque *if* en la función que inicia el segmento de memoria compartida del monitor.

[1]Esta medida de retirar su *pid* es necesaria para que el minero que este haya declarado ganador no le devuelva la pelota a este en caso de que él también tenga que salirse en esta ronda.

[2]El término *turnstile* provienen de The Little Book Of Semaphores

La siguiente mejora extiende el requisito de la práctica de que si el minero ganador de un ronda sufre un problema y no prepara la siguiente ronda, el resto de mineros puede terminar. En nuestro caso si el ganador de la última ronda recibe `SIGINT` (y, por lo tanto, debe finalizar), "asignará el título de último ganador" a un minero activo (como ya se ha explicado en el Nivel 3). Para asegurarnos de que la red continúe funcionando incluso si un minero recibe `SIGINT`, debemos permitir que se reciba esta señal antes de comprobar si esta ronda es la última del minero. Para ello, en el bucle de las rondas, quitamos y volvemos a poner la máscara del proceso (con `sigprocmask`).

Siguiendo con el buen funcinamiento de la red cuando un minero muere, hemos implementado que la red pueda recuperarse de la muerte de un minero de forma no controlada, como, por ejemplo, al recibir `SIGKILL`. Cuando un minero muere durante una ronda, el resto de mineros finalizaran debido a los mecanismos de *time out* implementados; no obstante, quedaba la cuestión de liberar correctamente los recursos o resetear los parámetro correspondientes por si se lanzaban más mineros. Lo que hacemos es que el primer minero que salga del bucle por un *time out*, ajustará el número de mineros (que será superior al verdadero ya que los que han muerto de forma descontrolada no notificaron su baja) de forma que el último minero en acabar libere todo. De esta forma conseguimos que cuando se manda `SIGKILL` a un minero, el resto de mineros se ven forzados a terminar pero los mineros que se lancen a continuación podrán volver a minar con normalidad.