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 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;
    long int end;
} 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`).

Al acabar cada ronda de minado se cancelam los hilos de los trabajadores desde el *main* solo para asegurar que nunca se empiece una nueva ronda con trabajadores aun activos (aunque sea algo prácticamente imposible). Los hilos se marcan como *detached* para evitar pérdidas de memoria como se pued ver en la siguiente imagen de una ejecución de un minero con 1000 trabajadores (lo que supone 2000 llamadas a *pthread_ init*).

```
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

Dependiendo de la velocidad relativa de ejecución de los hilos (trabajadores y *main*), los trabajadores pueden acabar por la llamada a *pthread_ cancel* como hemos mencionado anteriormente, o por su cuenta al salir del bucle de la búsqueda de la **POW** cuando el trabajador que ha encontrado la solución cambia una variable global para indicarlo.

Al finalizar las rondas de minado se imprime la cadena de vloques 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.

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)
        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 el ganador de una ronda se va a salir, porque esta es su última ronda (o porque ha recibido `SIGINT`), en la siguiente ronda la red pueda seguir funcionando. TURNSTILE::

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.

extra

sigint al ganador (turnstile)

2 monitores