Sistemas Operativos

Práctica 1

Sistema de minero único multihilo

Luis Felice Ruggiero
Angela Valderrama Ricaldi
06/03/2023

Introducción

Esta primera práctica es la base del proyecto final de las prácticas de la asignatura Sistemas Operativos. Ponemos en práctica los conocimientos adquiridos en esta primera parte sobre los procesos e hilos y la comunicación entre ellos, respectivamente, mediante tuberías.

El ejercicio de codificación consiste en la implementación de un sistema de minero único multihilo. Tenemos un programa principal *mrush* que creará un solo proceso hijo: *minero.* El proceso hijo *minero,* creará a su vez otro proceso, llamado monitor que supervisará el trabajo del minero.

La funcionalidad principal del minero es realizar un número determinado de rondas y en cada una de ellas, lanzar un número dado de hilos que ejecutarán una prueba de fuerza hasta que uno de ellos encuentre la solución. El uso de hilos es imprescindible para poder paralelizar el trabajo de la búsqueda y que el tiempo de ejecución sea menor.

Una vez, ésta se haya encontrado, le notificará al monitor el resultado para que lo valide mediante el uso de tuberías. El monitor recibirá el dato, comprobará su validez y se lo comunicará de vuelta al minero. De esta manera dependiendo de la validez, la siguiente ronda de minado empezará o no.

Para la implementación de este sistema se ha decidido estructurar el código en diferentes módulos, con sus propias estructuras y funciones, quedando así un código más legible y menos complejo.

Módulos

Mrush

Este módulo se corresponde con el programa principal. Se encarga de recibir los argumentos introducidos por la entrada estándar en el formato *./mrush <TARGET_INIT> <ROUNDS> <N_THREADS>*

De esta manera, si los argumentos no se corresponden con el formato establecido, se devolverá un mensaje informativo y se terminará el programa. Si el formato de los argumentos es correcto, entonces se puede proceder a la ejecución completa del programa multiproceso.

Este sistema, crea un proceso mediante la llamada a la función fork() que dependiendo de su valor de retorno se realizará una acción u otra. El proceso hijo ejecutará la función *minero()* enviando como parámetros el objetivo de minado, el número de rondas y el número de threads.

Además, se controla el valor de retorno de esta función, mediante la variable global *miner_stat* y una vez se termine, se procede a la finalización del proceso con *exit(miner_stat).* El proceso principal esperará a que termine el hijo y guardará su valor de exit en la variable *status.* Si esta variable status, se corresponde con EXIT_SUCCESS o EXIT_FAILURE, es decir, no ha terminado inesperadamente, entonces se procederá a terminar el programa con el valor de retorno adecuado.

Para ello, se ha hecho uso de los macros WIFEXITED y WEXITSTATUS de la librería <sys/wait.h>.

```c
/* Lanzamos al minero*/
pid = fork();
if(pid < 0) {
    printf("Error creating minero fork\n");
    exit(EXIT_FAILURE);
}else if(pid == 0) {
    miner_stat = minero(atoi(argv[1]), atoi(argv[2]), atoi(argv[3]));
    exit(miner_stat);
}

/* Esperamos a que termine el minero */
wait(&status);

/* Comprobamos el estado de salida del minero */
if(WIFEXITED(status)) {
    miner_stat = WEXITSTATUS(status);
}else{
    printf("Miner exited unexpectedly\n");
    exit(EXIT_FAILURE);
}

printf("Miner exited with status %d\n", miner_stat);
exit(EXIT_SUCCESS);
```

Figura 1: Creación del proceso minero

Minero

En este módulo se implementan las funciones necesarias para poder llevar a cabo las rondas de minado y el minado en sí. Como hemos mencionado antes, este minero resolverá una prueba de fuerza mediante un número determinado de hilos. Además, creará un proceso monitor que validará la solución encontrada y se comunicará con él mediante tuberías. Para la ejecución, se han definido un par de estructuras de datos con el fin de guardar información relevante y organizar el código:

```
typedef struct info_minero {

    int prevtarget;
    int target;
    int end;
    int validation;

} info_minero;
```

Figura 2. Estructura *info_minero*

Esta primera estructura almacenará información del minero. La variable *prevtarget* almacenará el target que se utilizó en la ronda anterior. Por otro lado, la variable *target* almacenará la solución encontrada que es propuesta como el objetivo de la siguiente ronda. La variable *end* funciona como una flag para saber si el minado ha terminado o no. Finalmente, la variable *validation* actúa también como una flag para identificar si la solución encontrada es válida o no.

```
typedef struct info_hilo {

    int lower;
    int upper;

} info_hilo;
```

Figura 3. Estructura *info_hilo*

Esta segunda estructura encapsula información para cada hilo. La primera variable, *lower,* consiste en un entero que guarda el valor inicial del rango. Con lo cual, *upper,* sería el entero que guarda el valor final del rango de búsqueda de ese hilo. De esta manera, el hilo sabe los límites del rango en el que tiene que ejecutar la prueba de fuerza.

Estas dos estructuras están pensadas para facilitar el acceso y compartición de información relevante tanto para esta práctica como para las futuras ya que esta es la base de las demás.

int minero(int obj, int rounds, int num_threads)

Esta es la función principal del módulo minero. Antes de explicar cómo es la lógica de cada ronda de minado, es conveniente mencionar que nuestro proceso minero utiliza 3 variables globales para gestionar la comunicación entre hilos:

- ***solución***: un entero que funcionará como flag, para que los hilos sepan que se ha encontrado la solución.
- ***target***: entero que se encargará de almacenar el objetivo de la próxima ronda (una vez termine la ronda se actualiza con el próximo objetivo a buscar).
- ***monitor_stat***: también será un entero y almacenará el valor de retorno del proceso monitor.

Se reciben los argumentos del programa principal *mrush* y se procede a configurar las herramientas necesarias para iniciar el minado. Se crean las dos tuberías necesarias para establecer la comunicación entre el proceso minero y el proceso monitor que se lanzará con una llamada a la función *fork()* tal y cómo se realiza en el módulo *mrush.* Aparte de esto, se reserva memoria para la estructura *info_minero* que contendrá información relevante para la comunicación entre ellos.

```c
/* Se guarda el target */
target = obj;

/* Se crean las tuberías */
pipe_stat = pipe(request_validation);
if(pipe_stat == -1){
    fprintf(stderr, "Pipe error\n");
    return EXIT_FAILURE;
}

pipe_stat = pipe(response_validation);
if(pipe_stat == -1){
    fprintf(stderr, "Pipe error\n");
    return EXIT_FAILURE;
}

/* Se crea el proceso monitor */
pid = fork();
if(pid < 0) {
    printf("No se pudo lanzar el monitor.\n");
    return EXIT_FAILURE;
} else if(pid == 0) {
    monitor_stat = monitor(request_validation, response_validation);
    exit(monitor_stat);
}

/* Se crea la estructura de información del minero */
info = (info_minero*)calloc(1, sizeof(info_minero));
if(info == NULL){
    fprintf(stderr, "info_minero error\n");
    return EXIT_FAILURE;
}

/* Se inicia el minado */
info->end = 0;
```

Figura 4: Creación de pipes y del proceso monitor

Como podemos observar en la siguiente imagen, primero actualizamos la estructura de *info_minero*, con el objetivo que vamos a buscar y realizamos la llamada a la función encargada de gestionar cada ronda, *round_init*, a la cual se le debe pasar como argumentos el número de hilos a utilizar. Ésta retornará cuando algún hilo encuentre la solución y se restablecerá la variable global *solucion* a 0 para la próxima ronda.

```
/* Lanzamos las rondas de minado */
for(i = 0; i < rounds; i++){

    /* Se guarda el target anterior */
    info->prevtarget = target;

    /* Empieza la ronda */
    round_init(num_threads);

    /* Se guarda la solucion */
    solucion = 0;

    /* Se calcula el nuevo target */
    info->target = target;

    /* Se cierra la tubería de lectura */
    close(request_validation[0]);

    /* Se escribe en la tubería de escritura */
    nbytes = write(request_validation[1], info, sizeof(info_minero));
    if(nbytes == -1){
        fprintf(stderr, "Write error\n");
        return EXIT_FAILURE;
    }

    /* Se cierra la tubería de escritura */
    close(response_validation[1]);

    /* Se lee de la tubería de lectura */
    nbytes = read(response_validation[0], info, sizeof(info_minero));
    if(nbytes == -1){
        fprintf(stderr, "Write error\n");
        return EXIT_FAILURE;
    }

    /* Se comprueba si la solución es válida */
    if(info->validation == -1) {
        val_status = -1;
        break;
    }

}
```

Figura 5: Gestión de cada ronda de minado

Además, se almacenará la solución y futura target en la estructura *info_minero*. Esta estructura se enviará mediante la tubería *request_validation* al monitor para su validación. Para ello, se cierra la tubería de lectura y se escribe en la de escritura. Para recibir la validación por parte del proceso monitor, se utiliza la tubería *response_validation* y se procede a cerrar el lado de escritura para leer en el de lectura. Si la validación ha resultado no ser válida, el minado debe terminar, con lo cual la ejecución del bucle de rondas no continúa.

```c
/* Se termina el minado */
info->end = 1;

/* Se cierra la tubería de lectura */
close(request_validation[0]);

/* Se escribe en la tubería de escritura */
nbytes = write(request_validation[1], info, sizeof(info_minero));
if(nbytes == -1){
    fprintf(stderr, "Write error\n");
    return EXIT_FAILURE;
}

/* Se comprueba si la solución es válida */
if(val_status == -1) {
    printf("The solution has been invalidated\n");
}
```

Figura 6: Finalización del minado.

Una vez se haya terminado el minado, se procede a cerrar las tuberías que comunican el proceso minero y monitor. Para ello, se establece la variable *end* de la estructura *info_minero* a 1 y se envía mediante la tubería para cerrar el lado de escritura y lectura correspondientes al proceso minero. Una vez el monitor termine su ejecución, el proceso minero procede a realizar la espera con la llamada a la función *wait(&status)* y las macros mencionadas antes para recoger al proceso hijo.

Por último, se liberan los recursos de memoria utilizados, se cierran los extremos de las tuberías correspondientes al proceso minero y se devuelve el valor de retorno dependiendo de si la validación fue satisfactoria o no.

```c
/* Se espera a que termine el monitor */
wait(&status);

/* Se comprueba el estado de salida del monitor */
if(WIFEXITED(status)) {
    monitor_stat = WEXITSTATUS(status);
    printf("Monitor exited with status %d\n", monitor_stat);
} else {
    printf("Monitor exited unexpectedly\n");
}

/* Se cierran las tuberías */
close(request_validation[0]);
close(response_validation[1]);

/* Se libera la memoria */
free(info);

/* Si la respuesta era inválida, se devuelve EXIT_FAILURE */
if(val_status == -1 || status == 1) {
    return EXIT_FAILURE;
}

return EXIT_SUCCESS;
```

Figura 7. Espera, liberación de recursos y finalización del proceso

int round_init (int num_threads)

En esta función, se inicia una ronda de minado. Es necesario conocer el número de hilos que se van a crear para poder paralelizar la prueba de fuerza entre ellos. Para comenzar, se crea un array del número de hilos indicados para llevar un seguimiento de su creación y un array de estructuras *info_hilo* pertenecientes a cada hilo.

Se procede al cálculo del número de iteraciones que deberá realizar cada hilo en su prueba de fuerza. En el bucle de creación de hilos, aparte de crear los hilos, se calcula el rango de búsqueda y se guarda la información en la estructura *info_hilo.* Cuando ya se hayan lanzado todos los hilos, comienza la espera de cada uno de ellos mediante la función *pthread_join.* Por último, se liberan los recursos de memoria utilizados y finaliza la función.

```c
/* Se calcula el número de iteraciones que realizará cada hilo */
init = (int)floor((double)(POW_LIMIT)/num_threads)+1;
end = init;

/* Se crean y se lanzan los hilos que ejecutarán la prueba de fuerza */
for(i = 0; i < num_threads; i++) {

    /* Se calcula el rango de búsqueda de cada hilo */
    thInfo[i].lower = init * i;
    thInfo[i].upper = end - 1;

    /* Se comprueba que el rango de iteraciones no se salga del límite */
    end += init;
    if(end > POW_LIMIT) {
        end = POW_LIMIT;
    }

    /* Se crea el hilo */
    err = pthread_create(&hilos[i], NULL, prueba_de_fuerza, (void *)&thInfo[i]);
    if(err != 0) {
        fprintf(stderr, "pthread_create : %s\n", strerror(err));
        free(hilos);
        return;
    }

}
```

Figura 8: Ronda de minado

*void *prueba_de_fuerza(void *info)*

Esta es la función que ejecutará cada uno de los hilos en cada ronda. Se realizará una búsqueda secuencial dados los parámetros del rango *lower* y *upper* para encontrar el valor que aplicada la función *pow_hash* da como resultado el target. A continuación, la flag *solucion* se pondrá a 1 para indicar a los demás hilos que se ha encontrado la solución y deben terminar el minado. Finalmente, se almacenará el nuevo objetivo en la variable global *target*.

```c
void *prueba_de_fuerza(void *info) {

    int res = 0, i = 0;
    info_hilo *thInfo = (info_hilo*)info;

    /* Se prueba la fuerza bruta */
    i = thInfo->lower;
    while(solucion == 0){

        /* Se comprueba que el número no se salga del rango */
        if(i > thInfo->upper) {
            break;
        }

        /* Se calcula el hash */
        res = pow_hash(i);

        /* Se comprueba si el hash es igual al objetivo */
        if(res == target) {

            /* Se guarda la solucion en la variable global target y
                se cambia la flag para que los demás hilos dejen de minar */
            solucion = 1;
            target = i;

            return NULL;
        }

        i++;
    }

    return NULL;
}
```

Figura 9: Función de prueba de fuerza.

Monitor

Este módulo contiene la funcionalidad principal del proceso monitor. Como su propio nombre indica, este proceso monitorizará el funcionamiento del proceso minero, validando la solución candidata a ser el próximo objetivo de la siguiente ronda. Este proceso se comunicará con el minero, mediante dos tuberías para tratar la solicitud y respuesta de validación.

*int monitor(int *request_validation, int *response_validation)*

Como podemos ver en la figura 10, el proceso monitor es un programa bastante sencillo en el cual se leerá la información proporcionada por el minero a través de uno de los pipes. Para ello, se cierra el lado de escritura del pipe *request_validation* y se lee del lado de lectura. La información estará encapsulada en la estructura *info_minero*, y desde ahí el monitor comprueba si debe cerrar los pipes porque el minado ha terminado o debe comprobar la solución propuesta. Dependiendo del resultado de la validación se actualizará el valor de la variable *validation* de la estructura *info_minero* con un valor u otro. Para enviar ese resultado, se hará uso de la otra tubería *response_validation* enviando toda la información al minero encapsulada en la estructura.

```c
do {

    /* Se cierra el pipe de escritura */
    close(request_validation[1]);

    /* Se lee del pipe de lectura */
    nbytes = read(request_validation[0], info, sizeof(info_minero));
    if (nbytes == -1){
        perror("read");
        return EXIT_FAILURE;
    }

    /* Se comprueba si el minero ha terminado */
    if(info->end == 1) {
        break;
    }

    /* Se comprueba si la solución es correcta */
    res = pow_hash(info->target);

    /* Se envía la respuesta al minero dependiendo */
    if(res == info->prevtarget) {
        printf("Solution accepted: %08d --> %08d\n", info->prevtarget, info->target);
        info->validation = 0;

    }else{
        printf("Solution rejected: %08d !-> %08d\n", info->prevtarget, info->target);
        info->validation = -1;
    }

    /* Se cierra el pipe de lectura */
    close(response_validation[0]);

    /* Se escribe en el pipe de escritura */
    nbytes = write(response_validation[1], info, sizeof(info_minero));
    if (nbytes == -1){
        perror("read");
        return EXIT_FAILURE;
    }

} while(nbytes != 0);
```

Figura 10: Programa monitor