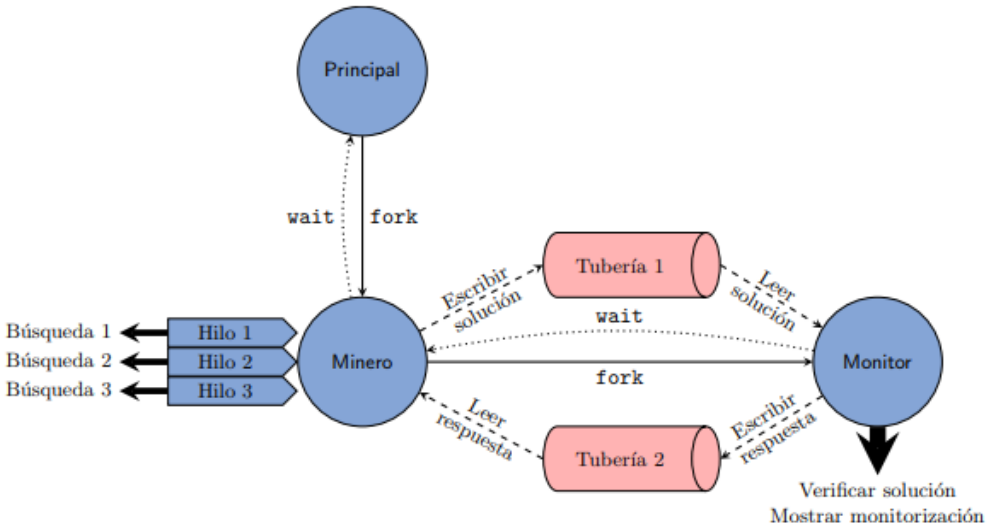
Práctica 1 SOPER PRÁCTICAS

Autores: Daniel Birsan & Jorge Paniagua Moreno

Crearemos un programa basado en Blockchain para el minado a partir de una función hash. Las funciones a implementar para el correcto funcionamiento del ejercicio 13 son las siguientes, además del main:

```c
// mining.h
#ifndef MINING_H
#define MINING_H

#define NUM_PROC 3
#define MAX_THREADS 100

typedef struct _Data{
    long pow_ini;
    long pow_fin;
    long target;
}Data;

int minero(int n_threads, long obj);
void* solve(void* data);
int monitor(long buf[2]);

#endif
```

Con las anteriores funciones seguiremos el esquema presentado en el enunciado de la práctica 1 de Sistemas Operativos.



FUNCIÓN MINERO:

```c
int minero(int n_threads, long obj)
{
    Data data[n_threads];
    int i;
    int error;
    pthread_t* threads;

    stop = 0;

    if (n_threads <= 0 || obj < 0 || obj > POW_LIMIT || n_threads > MAX_THREADS) return -1;

    threads = (pthread_t*) malloc(sizeof(pthread_t) * n_threads);
    if(!threads){
        return -1;
    }

    i = 0;
    while(i < n_threads)
    {
        data[i].target = obj;
        data[i].pow_ini = i* POW_LIMIT/n_threads;
        data[i].pow_fin = (i+1)* POW_LIMIT/n_threads;

        error = pthread_create(&threads[i], NULL, solve, &data[i]);
        if (error != 0) {
            fprintf(stderr, "pthread_create: %s\n", strerror(error));
            free(threads);
            return -1;
        }
        i++;
    }
    i = 0;
    while(i < n_threads)
    {
        error = pthread_join(threads[i], NULL);
        if (error != 0) {
            fprintf(stderr, "pthread_join: %s\n", strerror(error));
            free(threads);
            return -1;
        }
        i++;
    }

    free(threads);
    return 0;
}
```

Parámetros:

-n_threads: número de hilos que queremos crear.
-obj: target a buscar.

Funcionalidad:

La función minero creará un array de hilos (tantos como reciba por parámetro) reservando memoria dinámicamente, además establecerá el intervalo de búsqueda para cada hilo. Llamará a solve para encontrar el resultado y parará los hilos mediante un flag global (stop).

Retorno:

-”0” en caso de que todo vaya correctamente
-”1” en caso de algo de error

FUNCIÓN SOLVE:

```c
void* solve(void* data)
{
    if (!data) return NULL;

    Data* d = data;
    long i = 0;

    i = d->pow_ini;
    while(i < d->pow_fin && stop == 0)
    {
        if(d->target == pow_hash(i))
        {
            res = i;
            stop = 1;
        }
        i++;
    }
    return NULL;
}
```

Parámetros:

-data: estructura de datos del programa.

Funcionalidad:

Esta función se encarga de encontrar el resultado al target recibido por parámetro llamando a la función pow_hash, los valores de i dependen de cada hilo que utilice esta función, cuando encuentra el resultado, iguala una variable global al resultado y pone el flag de parada a 1 para que todos los hilos paren de buscar.

Retorno:

-”NULL”, para evitar fallos de compilado

FUNCIÓN MONITOR:

```c
int monitor(long buf[2])
{
    if(!buf) return 1;

    if (pow_hash(buf[1]) == buf[0] && buf[0] != buf[1])
    {
        printf("Solution accepted: %08ld --> %08ld\n", buf[0], buf[1]);
        return 0;
    } else {
        printf("Solution rejected: %08ld !-> %08ld\n", buf[0], buf[1]);
        if (buf[0] == buf[1]) printf("Too many threads\n");
        return 1;
    }

    return 1;
}
```

Parámetros:

-buf[2]: array de tipo long, en la primera posición contiene el target y en la segunda el resultado para este.

Funcionalidad:

La función monitor comprobará que el resultado recibido por argumento es correcto para el target también recibido llamando a la función pública pow_hash, comprobando el resultado mediante un if escribirá por pantalla si la solución ha sido aceptada o no, así como los valores correspondientes.

Retorno:

-”0” si la solución es aceptada

-”1” si la solución es denegada

FUNCIÓN MAIN:

Parámetros:

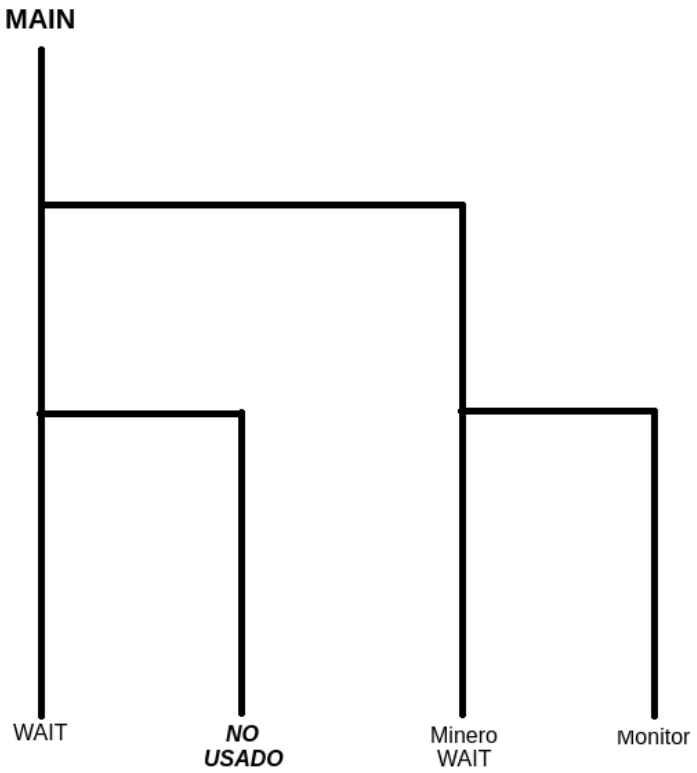
-argv[0]: target del programa
-argv[1]: número de rondas a ejecutar
-argv[2]: número de hilos que queremos crear

Funcionalidad:

El main se encargará de crear los distintos procesos mediante forks así como de abrir los pipes correspondientes para permitir la comunicación entre el proceso nieto y el proceso hijo. Creará un bucle durante el número de rondas para permitir el constante cambio del target leyendo de uno de los pipes el retorno del monitor para evitar fallos, el cual, a su vez, lee del pipe contrario el target y la solución para comprobar el resultado. Los forks permiten que el minero y el monitor se ejecuten a la vez y los pipes permiten que estos procesos se comuniquen entre ellos. Minero no ejecutará su siguiente ronda hasta que monitor no acabe de escribir la solución, de la misma forma que el proceso principal no acabará si minero no ha ejecutado todas sus rondas, de ahí los waits correspondientes en el código. Con estos waits conseguimos también las macros WIFEXITED, para comprobar que ambos procesos consiguen finalizar correctamente.

Salida:

-EXIT_SUCCESS: toda la función ha ido correctamente
-EXIT_FAILURE: se da cuando hay algún error en la función



Procesos creados con forks

```
./mrush 0 20 21
Solution accepted: 00000000 --> 38722988
Solution accepted: 38722988 --> 82781454
Solution accepted: 82781454 --> 59403743
Solution accepted: 59403743 --> 44638907
Solution accepted: 44638907 --> 98780967
Solution accepted: 98780967 --> 49574592
Solution accepted: 49574592 --> 16391022
Solution accepted: 16391022 --> 41194344
Solution accepted: 41194344 --> 61259182
Solution accepted: 61259182 --> 18852291
Solution accepted: 18852291 --> 74660642
Solution accepted: 74660642 --> 59894312
Solution accepted: 59894312 --> 55978326
Solution accepted: 55978326 --> 18633092
Solution accepted: 18633092 --> 61031588
Solution accepted: 61031588 --> 64519942
Solution accepted: 64519942 --> 64076854
Solution accepted: 64076854 --> 03955668
Solution accepted: 03955668 --> 27818400
Solution accepted: 27818400 --> 53980520
Monitor exited with status: 0
Miner exited with status: 0
```

Ejemplo de ejecución para target 0, 20 rondas, 21 hilos

LIMITACIONES:

Nuestro programa soporta un máximo de hilos para determinado número de rondas, por ejemplo, con las rondas anteriores su máximo de hilos es 21. Esto se puede deber a una sobrecarga del sistema operativo por el exceso de hilos ejecutándose simultáneamente.

TESTS:

Para probar el correcto funcionamiento de nuestro programa hemos probado todos los controles de errores realizados así como la creación de hilos, procesos no creados correctamente, zombies o huérfanos y retornos no válidos.