Universidad Autónoma de Madrid

Departamento de Informática

Sistemas Operativos

Memorias Práctica - 3

Hecho por: Diego Rodríguez Ortiz

y Alejandro García Hernando

Introducción

En esta práctica hemos aprendido el uso de la memoria compartida y lo importante que es la sincronización de distintos procesos que la comparten. La idea del programa es la coordinación de tres procesos que se comunicarán para hacer las labores de minar, comprobar e imprimir los resultados minados

Además, la comunicación entre Comprobador y Monitor se debe hacer mediante un sistema productor consumidor, y entre Minero y Comprobador una cola de mensajes. Para evitar problemas de concurrencia y condiciones de carrera hay que coordinar los procesos mediante semáforos. Por último, cabe comentar que en caso de error en el proceso Minero, se mandará un paquete de finalización a Comprobador quién también hará que Monitor finalice; y si falla Comprobador, hará que Monitor finalice y Minero acabará cuando acabe todas las rondas.

Estructura de código

El proyecto contiene 9 ficheros: 5 de código y 4 auxiliares

-Makefile: contiene una serie de comandos para la compilación y ejecución del código. Además de un comando que permite comprobar que se han cerrado correctamente los semáforos y un segundo que los elimina en caso de que no se hayan cerrado correctamente.

– minero.c: la función principal del proceso Minero se encuentra aquí. Esta función recoge los parámetros introducidos por el usuario y crea la cola de mensajes. Posteriormente, mina las rondas que se le ha indicado y finaliza iniciándose a Comprobador mandando un último mensaje con el código de finalización.

– monitor.c: la función principal de Comprobador y Minero se encuentra en este archivo. Esta función recoge los parámetros introducidos por el usuario y trata de crear la memoria, si está creada llama a la función Monitor y en caso contrario a Comprobador.

– funciones.c y funciones.h: son archivos creados para albergar las principales funciones del programa como son comprobador() y monitor(). También está la función privada _finish_comprobador(), llamada para finalizar la función comprobador, además de encargarse de cerrar y liberar los recursos pertinentes.

– utils.c y utils.h: estos archivos cuentan con las funciones auxiliares que son llamadas desde los archivos ya mencionados. Las funciones de estos archivos son las empleadas para la sincronización de procesos y para facilitar que los procesos se desincronicen, y comprobar si el programa falla o por el contrario funciona correctamente.

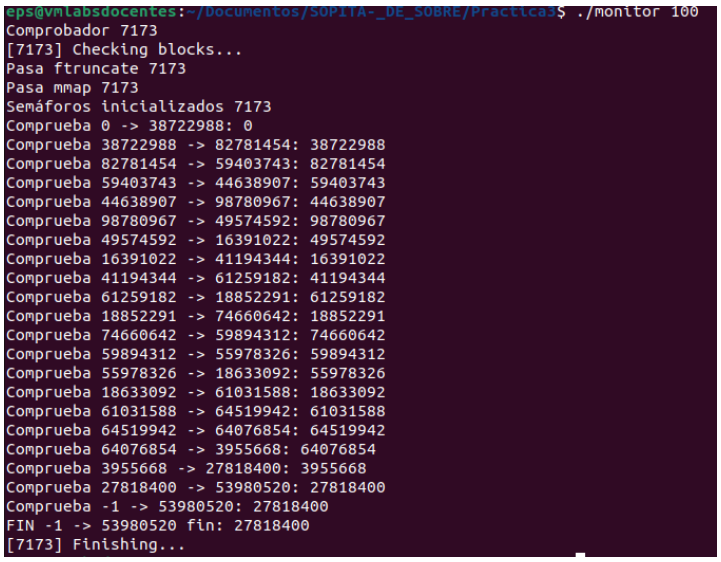
-pow.c y pow.h: contienen únicamente la función hash facilitada.

Pruebas

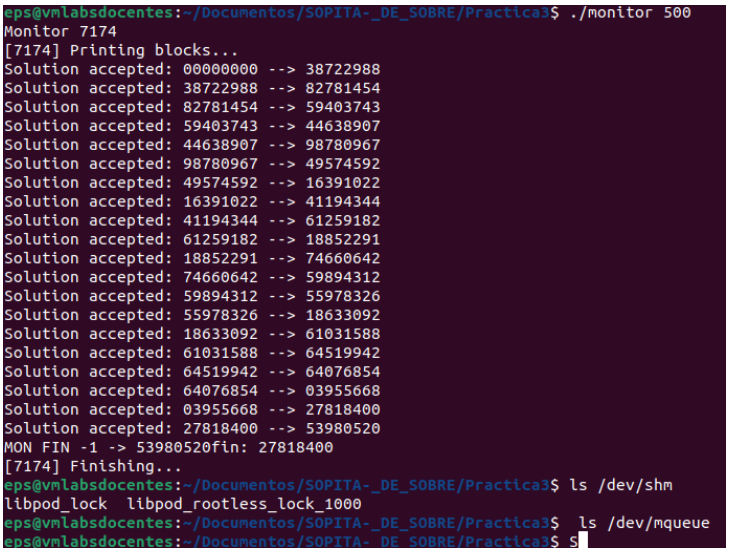
Finalización correcta de todos los procesos con DEBUG

Para cerciorarse del correcto funcionamiento de nuestro programa definimos la macro DEBUG que imprime por pantalla numerosos comentarios que nos permiten ver la evolución de los procesos, cada uno en su terminal. En la siguiente imagen se aprecian la traza de ejecución de los procesos. Además se adjunta otra imagen donde se ve como se liberan correctamente todos los recursos.

Ejecución del minero, observamos que los bloques generados son los mismos que los bloques comprobados.



Ejecución del comprobador, observamos que los bloques comprobados son los mismos que los bloques impresos y los generados.



Ejecución del monitor, observamos que los bloques impresos son los mismos que los bloques comprobados y los generados. Además nos cercioramos que se ha liberado todo correctamente

Control de parámetros de entrada

Esta simple prueba simplemente nos asegura que los parámetros introducidos al programa son correctos y no pueden generar problemas durante la ejecución del mismo.

```
eps@vmlabsdocentes:~/Documentos/SOPITA-_DE_SOBRE/Practica3$ ./minero -1 100
1 < <N_ROUNDS> < 1000 and 1 < <N_SECS> < 1000
eps@vmlabsdocentes:~/Documentos/SOPITA-_DE_SOBRE/Practica3$ ./minero 1001 100
1 < <N_ROUNDS> < 1000 and 1 < <N_SECS> < 1000
eps@vmlabsdocentes:~/Documentos/SOPITA-_DE_SOBRE/Practica3$ ./minero 10 0
1 < <N_ROUNDS> < 1000 and 1 < <N_SECS> < 1000
eps@vmlabsdocentes:~/Documentos/SOPITA-_DE_SOBRE/Practica3$ ./minero 10 10000
1 < <N_ROUNDS> < 1000 and 1 < <N_SECS> < 1000
eps@vmlabsdocentes:~/Documentos/SOPITA-_DE_SOBRE/Practica3$
```

```
eps@vmlabsdocentes:~/Documentos/SOPITA-_DE_SOBRE/Practica3$ ./monitor -1
1 < <LAG> < 1000
eps@vmlabsdocentes:~/Documentos/SOPITA-_DE_SOBRE/Practica3$ ./monitor 0
1 < <LAG> < 1000
eps@vmlabsdocentes:~/Documentos/SOPITA-_DE_SOBRE/Practica3$ ./monitor 1001
1 < <LAG> < 1000
eps@vmlabsdocentes:~/Documentos/SOPITA-_DE_SOBRE/Practica3$
```

Control de errores en Minero

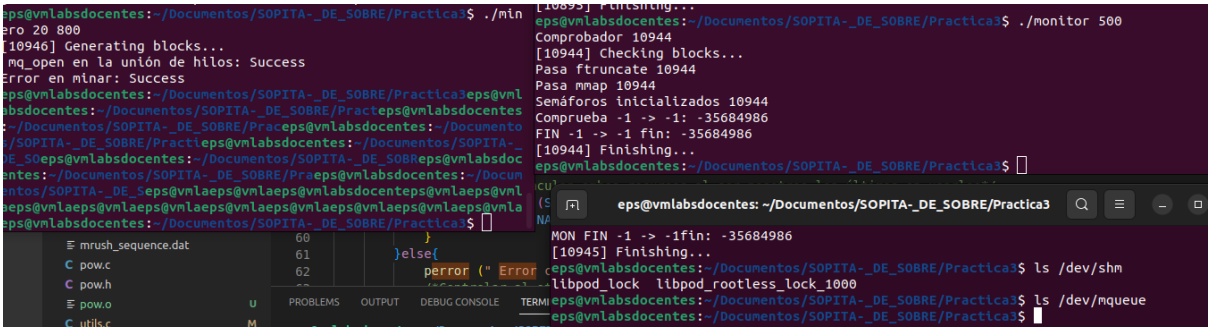
En esta prueba hemos forzado que Minero de error como se aprecia en la siguiente figura. En este caso también con la opción DEBUG activa, vemos como el error se controla bien y se notifica el final de la ejecución al Monitor para que termine de forma controlada. Las siguientes fotos muestran algunos ejemplos de las comprobaciones realizadas.

```
if (pthread_join(threads[i], sol + i)||1)
{

    for (i++; i < N_HILOS; i++)
        (void)pthread_join(threads[i], sol + i);
    error(" mq_open en la unión de hilos");

    return ERROR;
}
```

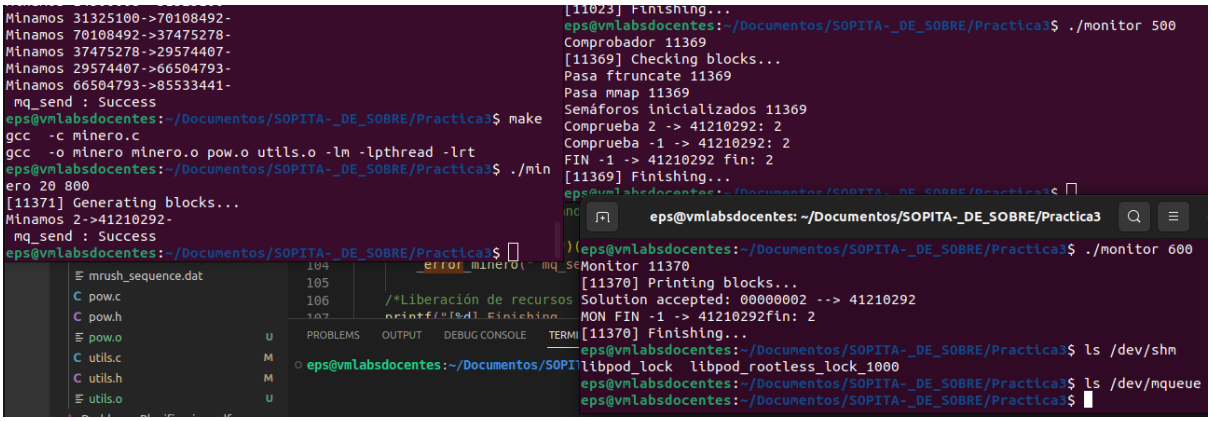
Simulación de error en el pthread_join



Ejecución controlada

```
if (mq_send(mq, (char *)(&msg), sizeof(Message), 0) == ERROR||1)
    _error_minero(" mq_send ", mq, msg);
```

Simulación de error en mq_send



Ejecución controlada

Control de errores en el Monitor y el Comprobador

De la misma forma que en la prueba anterior forzamos el error y observamos como el minero finaliza de forma controlada. No se nos ha ocurrido cómo gestionar el este tipo de errores en relación con el proceso monitor al desconocer el pid del proceso monitor no se nos ha ocurrido como indicarle que ha ocurrido un error.

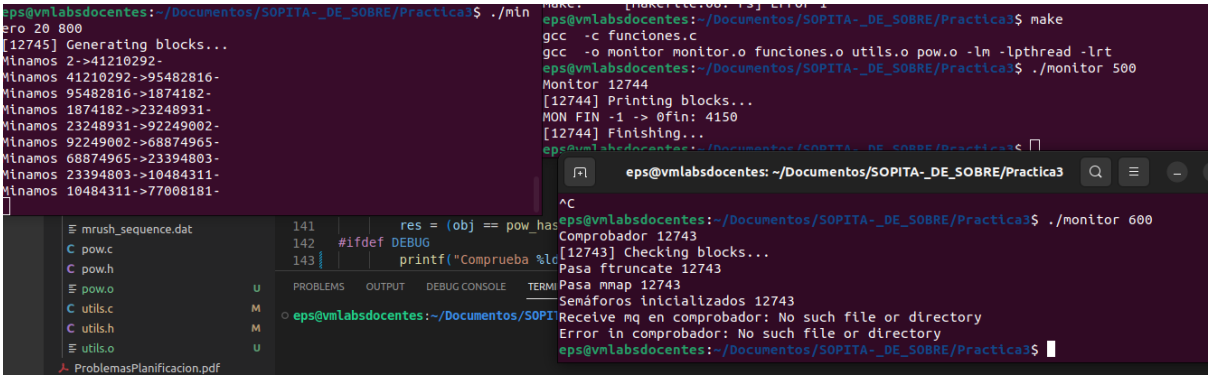
Estos son algunos ejemplos de las pruebas realizadas:

```c
if (mq_receive(mq, (char *)&msg, sizeof(Message), 0) == ERROR || 1)
{
    down(&mapped->sem_vacio);
    down(&mapped->sem_mutex);

    mapped->num[OBJ][index] = -1;
    mapped->num[SOL][index] = sol;
    mapped->num[RES][index] = res;
    index = (index + 1) % QUEUE_SIZE;

    up(&mapped->sem_mutex);
    up(&mapped->sem_lleno);
    mq_close(mq);
    return _finish_comprobador("Receive mq en comprobador", 0, mapped, 3, ERROR);
}
```

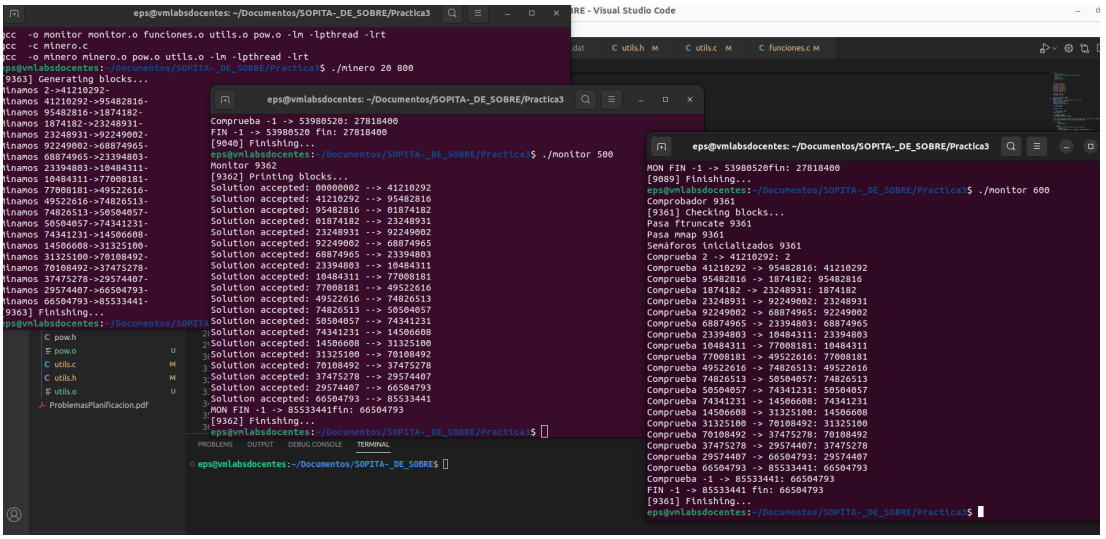
Simulación de error en mq_receive



Ejecución controlada salvo para minero

Sleeps aleatorios

Finalmente para evitar suposiciones de tiempo añadimos una macro TEST que si está definida añade sleeps aleatorios durante la ejecución de los distintos procesos, de esta forma comprobamos que no hayamos realizado ninguna suposición de tiempos. Esto es un ejemplo de la ejecución con la opción TEST activada



Análisis de la ejecución

¿Es necesario un sistema tipo productor-consumidor para garantizar que el sistema funciona correctamente si el retraso del Comprobador es mucho mayor que el de Minero? ¿Por qué?

No; si el retraso del Comprobador es mucho mayor que el de Minero, es posible que se intente mandar un mensaje cuando la cola esté llena. El carácter bloqueante del envío de mensajes evita los problemas de concurrencia, pues Minero se bloqueará hasta que el Consumidor saque un paquete de la cola, momento en el que se introducirá de manera efectiva. De este modo no se perderán paquetes.

¿Y si el retraso del Comprobador es muy inferior al del Minero?¿Por qué?

No; si el retraso del Comprobador es muy inferior que el de Minero, es posible que se intente recibir un mensaje cuando la cola esté vacía. El carácter bloqueante de la recepción de mensajes evita los problemas de concurrencia, pues el Comprobador se bloqueará hasta que Minero saque un paquete de la cola, momento en el que se introducirá de manera efectiva. De este modo no se producirán problemas de concurrencia ocasionados por condiciones de carrera.

¿Se simplifica la estructura global del sistema si entre Comprobador y Monitor hubiera también una cola de mensajes?¿Por qué?

Sí; porque no hace falta utilizar semáforos de sincronización para coordinar los procesos. Utilizando un sistema productor-consumidor es necesario utilizar memoria compartida y múltiples semáforos para evitar las condiciones de carrera. Mientras que si hubiera una cola de mensajes entre Comprobador y Monitor, el carácter bloqueante del envío y recepción de de mensajes evita los problemas de concurrencia.

Conclusión

Esta práctica se enfoca en la gestión de memoria compartida y en la sincronización de procesos para evitar condiciones de carrera sobre esa memoria. Los tres procesos utilizados como ejemplo son los procesos Minero, Comprobador y Monitor, que realizan tareas de forma coordinada. Esto nos ha permitido familiarizarnos con la concurrencia y compartición de recursos. Para evitar los problemas descritos a lo largo del informe es de gran utilidad utilizar semáforos y mensajes. Además, la noción de memoria compartida nos permite mejorar la comunicación entre procesos otorgándonos una herramienta muy potente para gestionar la concurrencia.