Sistemas Operativos

Práctica 3

Sistema de monitoreo multiproceso

Angela Valderrama Ricaldi
Luis Felice Ruggiero
17/04/2023

Introducción

Esta tercera práctica consiste en la última iteración del proyecto final de la asignatura Sistemas Operativos antes de unificar todos los módulos en este último. Los conocimientos tratados están relacionados con la memoria compartida, a la cual pueden acceder varios procesos y las colas de mensajes, un mecanismo para permitir la comunicación entre procesos.

Para esta iteración, el ejercicio de codificación consiste en la implementación de un sistema de monitoreo multiproceso, en el cual, tendremos tres programas a ejecutar con funcionalidades encapsuladas en dos módulos: Monitor y Minero. De esta manera, se mantiene la encapsulación de código y la programación modular que ayuda a tener organización en la implementación del sistema.

Los tres programas que se ejecutarán se corresponden con un Minero y dos instancias de Monitor, en el cual distinguimos a un proceso Comprobador y a un proceso Monitor. La diferencia entre los dos la encontramos en quién es el proceso que crea el segmento de memoria compartida, en este caso, el proceso Comprobador.

El proceso Minero se encarga de realizar una prueba de fuerza para encontrar la solución de un objetivo dado durante un número determinado de rondas. En cada una de ellas, comunicará el objetivo y el resultado obtenido al proceso Comprobador mediante un mensaje en la cola de mensajes. Cuando el Minero termine las rondas de minado, éste enviará un último mensaje indicando que ha finalizado su tarea al Comprobador y terminará su ejecución.

El proceso Comprobador, recibirá los mensajes mediante la cola de mensajes y comprobará que la solución enviada es válida realizando la prueba de fuerza a la inversa. Después de la comprobación, comunicará el resultado al proceso Monitor, insertando la información en el segmento de memoria compartida al proceso Monitor para que éste imprima por pantalla si la solución ha sido aceptada o no.

Aunque la práctica esté enfocada en desarrollar los conocimientos adquiridos sobre la memoria compartida y las colas de mensajes, se han utilizado conceptos relacionados con prácticas anteriores como los semáforos para permitir la ejecución concurrente de procesos, así como la función para implementar la prueba de fuerza.

Módulos

Monitor

Este módulo contiene la implementación del programa de monitorización. Para ejecutarlo, se debe escribir *./monitor <LAG>* siendo LAG el retraso en milisegundos de cada monitorización. Para el funcionamiento del sistema, se deberán lanzar dos procesos de este programa con el comando mencionado anteriormente. Para la implementación del módulo se han creado las diferentes estructuras de datos que nos serán de utilidad en el desarrollo del programa.

```
/**
 * @brief Estructura que representa un bloque
 *
 */
typedef struct Bloque
{
    char msg[MAX_MSG];
} Bloque;
```

Figura 1. Definición de la estructura Bloque

Esta estructura representa un Bloque, que contendrá el objetivo y la solución de una ronda de la prueba de fuerza. Mediante las colas de mensajes, sólo se pueden enviar cadenas de caracteres, es por eso por lo que Bloque contiene la cadena *msg* donde se guardará la información con el formato *<target>-<res>.*

```
/**
 * @brief Estructura a compartir entre procesos
 *
 */
typedef struct shm_struct
{
    Bloque buffer[6];
    sem_t sem_empty;
    sem_t sem_mutex;
    sem_t sem_fill;
    int front;
    int rear;
} shm_struct;
```

Figura 2. Definición de la estructura shm_struct

Esta estructura representa el segmento de memoria compartida entre el proceso Comprobador y el proceso Monitor. Contiene el array circular o buffer de tipo Bloque en el cual se irá insertando y extrayendo bloques según el proceso Comprobador vaya recibiendo de la cola de mensajes y el proceso Monitor vaya consumiendo del buffer. Como se trata de un array circular, es necesario llevar un registro de las últimas posiciones leídas y escritas y para eso están las variables *front* y *rear*.

Para asegurar el acceso concurrente a esta zona compartida de memoria, se lleva a cabo un sistema de sincronización basado en el esquema Productor-Consumidor, para el cual es necesario una serie de semáforos: sem_empty, sem_mutex y sem_fill. Al inicializarlos, estos semáforos serán sin nombre porque ya se comparten en el segmento de memoria.

*void main(int argc, char *argv[])*

Uno de los procesos corresponde con el proceso Comprobador y el otro con el proceso Monitor. El primer proceso crea el segmento de memoria compartida y el segundo, al ver que ya está creada se limita a abrirla. Para diferenciar los dos procesos, nos fijaremos en el valor de retorno de la función *shm_open.*

```
/* Crear segmento de memoria compartida o abrirlo si ya existe */
fd_shm = shm_open(SHM_NAME, O_RDWR | O_CREAT | O_EXCL, S_IRUSR | S_IWUSR);
if (fd_shm == -1)
{
    if (errno == EEXIST)
    {
        /* Si ya existe, soy el proceso Monitor y abro el segmento de memoria */
        fd_shm = shm_open(SHM_NAME, O_RDWR, 0);
        if (fd_shm == -1)
        {
            perror("Error opening the shared memory segment\n");
            exit(EXIT_FAILURE);
        }

        monitor(lag, fd_shm);
        exit(EXIT_SUCCESS);
    }
    else
    {
        perror("Error creating the shared memory segment\n ");
        exit(EXIT_FAILURE);
    }
}

/* He creado el segmento de memoria y soy el proceso Comprobador */
comprobador(lag, fd_shm);
```

Figura 3. Distinción del proceso Comprobador y Monitor

Como podemos observar, con la función *shm_open* se crea un segmento de memoria compartida con las diferentes *flags* que indican los permisos de ese segmento y el modo de creación. Con la bandera O_EXCL, especificamos que ese segmento con el nombre proporcionado en SHM_NAME, debe ser exclusivo (único). De esta manera, si un segmento con nombre SHM_NAME ya ha sido creado y se vuelve a ejecutar *shm_open* con las banderas O_CREAT y O_EXCL, la función devolverá -1 y el valor de *errno* correspondiente a EEXIST indicando que ese segmento que se ha intentado crear ya existe.

Esto es justo lo que queremos que ocurra para poder diferenciar a los procesos Comprobador, que creará el segmento y el proceso Monitor, que intentará crear el segmento de nuevo, pero al ya existir, deberá abrirlo sin las flags O_CREAT y O_EXCL.

Una vez se haya hecho esta diferenciación, simplemente llamamos a las funciones pertinentes para ejecutar ya sea el monitor o el comprobador, enviando como argumentos el tiempo de lag especificado al ejecutar el programa y el descriptor de fichero del segmento de memoria.

void comprobador(int lag, int fd_shm)

Este proceso va a ser el encargado de recibir los bloques a través de un mensaje en la cola de mensajes, provenientes del proceso Minero y después de comprobarlos, los enviará al proceso Monitor mediante su segmento de memoria compartida para que los imprima por pantalla.

Primero, como este es el proceso que ha creado el segmento de memoria compartida, debemos asignar el tamaño adecuado con ftruncate y mapear el segmento en nuestro propio espacio de direccionamiento con la función mmap. En nuestro caso, el tamaño del segmento se corresponde con el tamaño de la estructura shm_struct.

La función *mmap* devuelve la dirección del segmento en la memoria del proceso, es decir, devuelve la dirección de un objeto shm_struct. Después de recibirlo de vuelta, debemos inicializar las diferentes variables contenidas en la estructura mediante la función *init_struct().* Una vez esté la estructura del segmento, abriremos la cola de mensajes por la que recibiremos mensajes del proceso Monitor, para empezar a escuchar de ella.

```
attributes.mq_maxmsg = MQ_LEN;
attributes.mq_msgsize = MAX_MSG;

printf("[%d] Checking blocks...\n", getpid());

/* Establecer tamaño del segmento de memoria compartida */
if (ftruncate(fd_shm, sizeof(shm_struct)) == -1)
{
    perror("ftruncate");
    shm_unlink(SHM_NAME);
    exit(EXIT_FAILURE);
}

/* Mapeo de segmento de memoria */
shm_struc = mmap(NULL, sizeof(shm_struct), PROT_READ | PROT_WRITE, MAP_SHARED, fd_shm, 0);
close(fd_shm);
if (shm_struc == MAP_FAILED)
{
    perror("mmap");
    shm_unlink(SHM_NAME);
    exit(EXIT_FAILURE);
}

/* Inicializar estructura de memoria compartida */
init_struct(shm_struc);

/* Abrir cola de mensajes */
if ((mq = mq_open(MQ_NAME, O_CREAT | O_RDWR , S_IRUSR | S_IWUSR, &attributes)) == (mqd_t) -1) {
    perror("mq_open") ;
    exit(EXIT_FAILURE) ;
}
```

Figura 4. Configuración principal del proceso Comprobador

Posterior a la configuración principal del proceso, empezará el bucle principal donde se realizarán las siguientes tareas:

- Se recibe el mensaje de la cola de mensajes mediante la función *mq_receive()* y se guarda en la variable *recv*.
- Si el mensaje recibido es “end”, se mensaje se corresponde con el bloque de finalización. Como consecuencia, se cambia el valor de la variable *exit_loop* a 1 para salir del bucle principal.
- Si se recibe cualquier otro mensaje, éste será un bloque con el formato *<target>-<res>* y se guardarán del contenido del mensaje los valores de *target y res* que se corresponden con el objetivo y la solución después de ejecutar la prueba de fuerza.
- Como su propio nombre indica, este proceso se encarga de comprobar que esos valores son correctos mediante la función *comprobar()* que devuelve el valor 0 si es incorrecto o 1 si es válido. Con esta información, se forma el Bloque a escribir en el buffer circular con el siguiente formato: *<target> <res> <flag>* siendo flag el valor de retorno de la función *comprobar()* para que el proceso Minero sepa si la solución es aceptada o no.
- A continuación, se procede a escribir en el segmento de memoria el bloque recibido. Para ello, el proceso Comprobador y Minero seguirán un esquema Productor-Consumidor para poder permitir la sincronización entre los dos procesos y el acceso concurrente a la memoria compartida. Esto se controla con el uso de sem_empty y sem_mutex, verificando que hay espacio para escribir en el buffer de bloques y accediendo para añadir uno.
- Si podemos escribir, al ser un buffer circular, primero incrementamos el valor de la variable *rear,* que indica la última posición escrita y escribimos en la posición *rear % BUFFER_SIZE* (donde BUFFER_SIZE será el tamaño del array circular, que en nuestro caso será 6).
- Finalmente, indicaremos que es seguro acceder a la sección crítica de nuevo, y que se ha llenado una posición más en el array circular con los semáforos sem_mutex y sem_fill, respectivamente.
- Para concluir, se esperarán los segundos indicados por el usuario en la variable LAG.

Si se ha salido del bucle principal, es porque se ha recibido el bloque de finalización y éste se ha escrito con éxito en el array circular. Por eso, el proceso Comprobador, ejecutará la función *munmap* para desasociar el segmento de memoria compartida. Además, dejará de escuchar de la cola de mensajes, cerrándola con la función *mq_close* y borrando los enlaces con la función *mq_unlink*. Por último, se termina la ejecución del programa.

void monitor(int lag, int fd_shm)

Este proceso solo se encargará de mostrar los resultados que envía el proceso Comprobador, por pantalla. Para ello, este proceso sólo debe mapear el segmento de memoria compartida para poder trabajar con él

de una forma más sencilla. Una vez se haya realizado el mapeo, empezará el bucle principal del programa que realizará las siguientes acciones:

- Mediante el uso de los semáforos sem_fill y sem_mutex, esperará a que haya algún Bloque escrito en el buffer circular (siguiendo el esquema Productor-Consumidor) y esperará a que sea seguro acceder a la sección crítica para poder leer la estructura compartida.
- Si puede leer de la memoria compartida, actualizará el valor de la variable *front,* que indica la última posición leída y leerá de *front % BUFFER_SIZE*.
- Al igual que el comprobador, si el mensaje leído se corresponde con “end”, cambiará el valor de la variable *end_loop* a 1 para salir del bucle principal.
- Si ese no es el caso, imprimirá el mensaje enviado por el comprobador por pantalla haciendo distinción de si la solución es válida o no, según el valor de la variable *flag*. Después, permitirá la escritura de otro proceso e indicará que se consumió una entrada de la cola mediante los semáforos sem_mutex y sem_empty.
- Finalmente, realizará la espera indicada por el valor de la variable lag que se estableció como un argumento al ejecutar el programa.

```c
/* Extrae bloque de memoria compartida */
while (end_loop == 0)
{

    /* Espera a que haya un bloque en el buffer */
    sem_wait(&shm_struc->sem_fill);
    /* Protegemos el acceso a memoria compartida */
    sem_wait(&shm_struc->sem_mutex);

    /* Leer bloque de memoria compartida */
    shm_struc->front++;
    if (shm_struc->front == BUFFER_SIZE)
    {
        shm_struc->front = 0;
    }

    shm_struc->front = shm_struc->front % BUFFER_SIZE;

    /* Comprueba si es el bloque de finalización */
    if (strcmp(shm_struc->buffer[shm_struc->front].msg, "end") == 0)
    {
        end_loop = 1;
    }
    else
    {
        /* Leer objetivo y solución */
        sscanf(shm_struc->buffer[shm_struc->front].msg, "%ld %ld %d", &target, &res, &flag);

        if (flag == 1)
        {
            printf("Solution accepted: %08ld --> %08ld\n", target, res);
        }
        else
        {
            printf("Solution rejected: %08ld --> %08ld\n", target, res);
        }
    }

    sem_post(&shm_struc->sem_mutex);
    sem_post(&shm_struc->sem_empty);

    /* Espera */
    usleep(lag);
}
```

Figura 5. Bucle principal del proceso Monitor

Cuando el proceso Monitor salga del bucle principal, será porque ha recibido el bloque de finalización. Teniendo en cuenta que este proceso sólo de encarga de consumir bloques y de mostrar su contenido por pantalla, es el indicado de destruir los semáforos utilizados ya que es el último en acceder a ellos. Después de limpiar esos recursos, para finalizar, eliminará de su espacio de direcciones el segmento de memoria compartida y terminará su ejecución.

*void init_struct(shm_struct *shm_struc)*

Como hemos mencionado antes, el proceso Comprobador que crea el segmento de memoria compartida, debe inicializar la estructura que se guarda en ese espacio. Por eso, esta función, inicializa los valores de *front* y *rear* que sirven para llevar un control de las posiciones leídas y escritas. Además, se crean los semáforos sin nombre necesarios para el esquema Productor-Consumidor que llevarán a cabo los proceso Comprobador y Monitor, sem_mutex, sem_empty y sem_fill.

```c
void init_struct(shm_struct *shm_struc)
{

    if (shm_struc == NULL)
    {
        return;
    }

    /* Inicializar front y rear del buffer circular */
    shm_struc->front = -1;
    shm_struc->rear = -1;

    /* Inicializar semáforos sin nombre a utilizar */
    if (sem_init(&shm_struc->sem_mutex, 1, 1) == -1)
    {
        perror("sem_init");
        exit(EXIT_FAILURE);
    }

    if (sem_init(&shm_struc->sem_empty, 1, (BUFFER_SIZE - 1)) == -1)
    {
        perror("sem_init");
        exit(EXIT_FAILURE);
    }

    if (sem_init(&shm_struc->sem_fill, 1, 0) == -1)
    {
        perror("sem_init");
        exit(EXIT_FAILURE);
    }
}
```

Figura 6. Inicialización de los atributos de la estructura shm_struct

int comprobar(int target, int res)

Esta función se utiliza para comprobar que la solución recibida por el proceso Minero es correcta. Para ello, se vuelve a realizar la prueba de fuerza en la solución para verificar que se recibe de vuelta el valor del objetivo. Si es así, la solución será válida y se devolverá un 1, mientras que, si la solución no es válida, el valor de retorno de la función será 0.

```
int comprobar(int target, int res)
{

    int flag = 0;

    if (target == pow_hash(res))
    {
        flag = 1;
    }

    return flag;
}
```

Figura 7. Función para comprobar la prueba de fuerza

Miner

En este módulo se implementa la funcionalidad relacionada con el programa de minería. Para su ejecución, es necesario escribir en terminal ./miner <ROUNDS> <LAG> siendo ROUNDS el número de rondas que se van a realizar y LAG el número de milisegundos entre cada ronda.

*void main(int argc, char *argv[])*

El proceso Minero se encarga de crear una cola de mensajes con capacidad para 7 mensajes mediante la cual comunicará al proceso Comprobador el objetivo y la solución de cada ronda. Para empezar, el primer objetivo es 0 y en las siguientes rondas el próximo target será la solución encontrada. El formato del mensaje que enviará el Minero es *<target>-<res>* y se enviará mediante la función *mq_send().* Después de enviar el mensaje, realizará la espera determinada en la variable lag y cambiará el valor del nuevo objetivo.

Cuando las rondas hayan acabado, se enviará un mensaje de finalización, “end”, al proceso Comprobador para que éste sepa que el minado ha acabado y debe avisar al proceso Monitor, liberar los recursos y terminar su ejecución. Después de enviar este último mensaje, el proceso Minero cerrará la cola de mensajes y terminará su ejecución.

```
    if ((mq = mq_open(MQ_NAME, O_CREAT | O_RDWR, S_IRUSR | S_IWUSR, &attributes)) == (mqd_t)-1)
    {
        perror("mq_open");
        exit(EXIT_FAILURE);
    }

    target = 0;
    printf("[%d] Generating blocks...\n", getpid());

    /* Empezar las rondas de prueba de fuerza */
    for (i = 0; i < nrounds; i++)
    {

        res = prueba_de_fuerza(target);

        /* El formato del mensaje será target-res */
        sprintf(msg, "%d-%d", target, res);

        /* Enviar el mensaje */
        if (mq_send(mq, msg, strlen(msg) + 1, 1) == -1)
        {
            perror("mq_send");
            mq_close(mq);
            exit(EXIT_FAILURE);
        }

        /* Esperar el tiempo de lag */
        usleep(lag);

        /* Obtener el nuevo target */
        target = res;
    }

    /* Enviar mensaje de finalización */
    strcpy(msg, "end");

    if (mq_send(mq, msg, strlen(msg) + 1, 1) == -1)
    {
        perror("mq_send");
        mq_close(mq);
        exit(EXIT_FAILURE);
    }

    /* Cerrar la cola de mensajes */
    mq_close(mq);

    printf("[%d] Finishing\n", getpid());

    exit(EXIT_SUCCESS);
}
```

Figura 8. Funcionalidad principal del proceso Minero

int prueba_de_fuerza(int target)

En esta función se ejecuta la prueba de fuerza, mediante la función proporcionada *pow_hash(),* para los valores comprendidos entre 0 y POW_LIMIT – 1 con la finalidad de resolver el objetivo *target.*

```
int prueba_de_fuerza(int target)
{

    int i;

    for (i = 0; i < POW_LIMIT; i++)
    {
        if (pow_hash(i) == target)
        {
            break;
        }
    }

    return i;
}
```

Figura 9. Función que implementa la prueba de fuerza

Preguntas

En base al funcionamiento del sistema anterior:

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

  El sistema tipo productor-consumidor asegura que sólo se producen un número determinado de *productos* y que sólo se consume si existe al menos un *producto.* Es decir, en nuestro caso, el *producto* se introduce en el buffer circular que tiene un tamaño máximo de 6 *productos.* De esta manera, se controla que el proceso Monitor consumirá del buffer si existe al menos un producto (sem_fill) y el proceso Consumidor sólo producirá si hay espacio en el buffer (sem_empty).

  Si el retraso del Comprobador es mayor que el del proceso Monitor, esto significa que pasa más tiempo entre cada producción que tiempo entre cada consumición, con lo cual, se consumen bloques más rápido que los que se producen.

  En este caso, el proceso Minero se quedará bloqueado en el semáforo sem_fill esperando a que haya un *producto* que consumir. El esquema productor-consumidor es necesario para asegurar la protección de la sección crítica y la consumición errónea de bloques o valores repetidos.

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

  Si el proceso Comprobador tiene un retraso menor que el del Minero, significa que el tiempo de producción es más rápido que el de consumición, con lo cual, se produce más rápido de lo que se consume.

  De esta manera, es muy posible que el buffer se llene rápido y que el productor no pueda añadir el producto porque no hay espacio para escribir. Es decir, el proceso Comprobador se quedaría bloqueado en el semáforo sem_empty esperando a que haya alguna posición del buffer circular libre para poder escribir o lo que es lo mismo, que el proceso Minero consuma al menos un producto.

  Para este caso, el sistema productor-consumidor sigue siendo necesario para proteger la sección crítica y la producción masiva de bloques que derivaría a pérdidas de bloques porque al proceso Monitor no le daría tiempo a leerlos.

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

  Sí se simplificaría, los semáforos no serían necesarios ya que los bloques se transmitirían mediante la cola de mensajes entre el proceso Comprobador y el proceso Monitor. Al no haber, memoria compartida, no es necesario proteger la sección crítica. Los mensajes se encolarían en la cola y el tiempo de retraso no influiría en la lectura o escritura porque se mantendría el orden de los mensajes.