MEMORIA PRÁCTICA 3

SISTEMAS OPERATIVOS

Rubén García de la Fuente
Elena Cano Castillejo
Pareja 04
Grupo 2202

<u>Ejercicio 1: Memoria Compartida.</u> Dado el siguiente código en C: *(0,75 ptos.)*

a) **¿Tiene sentido incluir shm_unlink en el lector? ¿Por qué?** *(0,25 ptos.)*

No, ya que unlink elimina el nombre del segmento de memoria compartida. El segmento se eliminará cuando se haya hecho un close del descriptor de fichero de memoria compartida y cuando los procesos no tengan la región de memoria asociada al segmento, es decir, cuando todos los procesos hayan hecho munmap del segmento de memoria compartida. Por tanto, sólo hace falta borrar el nombre una vez.

b) **¿Cuál es la diferencia entre shm_open y mmap? ¿Qué sentido tiene que existan dos funciones diferentes?** *(0,25 ptos.)*

La función shm_open se encarga de la creación y apertura de los segmentos de memoria devolviendo un descriptor de fichero. La función mmap, sin embargo, se encarga de facilitar el uso de ficheros y memoria compartida llevando a cabo el enlace entre el proceso y la memoria virtual.
Esto se debe a que puede que se quiera crear un segmento de memoria virtual pero sin querer mapearlo.

c) **¿Se podría haber usado la memoria compartida sin enlazarla con mmap? Si es así, explicar cómo.** *(0,25 ptos.)*

Se podría haber utilizado la memoria compartida trabajando directamente sobre el descriptor de fichero, esto es, una vez obtenido el descriptor de fichero tras haber llamado a la función shm_open. Podríamos haber utilizado las funciones write y read para escribir y leer respectivamente sobre el fichero de memoria compartida. Una vez que se hayan realizado todas las acciones de lectura/escritura se cierra el descriptor de fichero para que se pueda liberar la memoria tras la llamada a shm_unlink.

<u>Ejercicio 2: Creación de Memoria Compartida.</u> Dado el siguiente código en C: *(0,75 ptos.)*

a) **Explicar en qué consiste este código, y que sentido tiene utilizarlo para abrir un objeto de memoria compartida.** *(0,25 ptos.)*

El código del ejemplo se encarga de abrir un segmento de memoria compartida y gestionar correctamente los posibles errores.
Tiene sentido el empleo de este código debido a que a la hora de abrir un archivo hay que tener en cuenta que puede que ya exista un segmento con el nombre que se le ha pasado como argumento. En este caso se encargará de abrirlo si ya existe, o de crearlo y lo abrirlo si no, por lo que al emplear este código gestionamos las distintas opciones y devolvemos error solo en el caso de que haya habido un fallo a la hora de abrirlo.

b) **Indicar qué habría que añadir al código anterior para que, por un lado, pueda inicializar el tamaño de la memoria compartida a 1000 B, y por otro lado, futuras ejecuciones no vuelvan a inicializar o destruir lo ya inicializado.** *(0,25 ptos.)*

En la sentencia else en la cual se imprime "`Shared memory segment created`", justo antes del printf habría que añadir `ftruncate(fd_shm, 1000)`.
De esta manera, la primera vez que se cree el segmento de memoria compartida, al no existir, se ejecutará la sentencia else indicada y se truncará esta memoria a 1000B.
Así mismo, cada vez que se llame a shm_open y ya esté creada, no se llamará a ftruncate y por tanto lo volvería a inicializar la memoria compartida.

c) **En un momento dado se deseará forzar (en la próxima ejecución del programa) la inicialización del objeto de memoria compartida "/shared". Explicar posibles soluciones (en código C o fuera de él) para forzar dicha inicialización.** *(0,25 ptos.)*

Para llevar a cabo esta acción, antes de la llamada a shm_open podemos realizar una comprobación de una flag que cambie en función del argumento de entrada introducido, de tal manera que dependiendo del valor de la misma ejecute el shm_open y gestionar el caso de que si existe un segmento con ese nombre lo abra y lo trunque.
Otra opción sería borrar el descriptor de fichero directamente del directorio `/dev/shm` mediante el uso del comando rm, por lo que nuestro programa en la próxima ejecución verá que no existe y creará uno nuevo.

Ejercicio 3: Concurrencia en Memoria Compartida. *(2,75 ptos.)*

a) **Escribir un programa en C ("shm_concurrence.c") que satisfaga los siguientes requisitos:** *(1,50 ptos.)*
   - **Generará N procesos hijos, donde N es el primer argumento de entrada al programa.**
   - **El proceso padre reservará un bloque de memoria, que compartirá con los procesos hijo, suficiente para una estructura de tipo.**
   - **El campo logid ha de inicializarse a -1.**
   - **Cuando el proceso padre reciba la señal SIGUSR1 leerá de la zona de memoria compartida e imprimirá su contenido como "<logid>:<processid>:<logtext>".**
   - **Cada proceso hijo realizará los siguientes pasos (en este orden):**
     - **M veces (M es el segundo argumento del programa):**
       - **Dormirá un tiempo aleatorio entre 100 ms y 900 ms.**
       - **Rellenará la información de la estructura ClientLog para generar una nueva línea de log, de manera que processid sea igual al PID del proceso, se incremente el campo logid (clinealogid = clinealogid + 1), y logtext contenga el**

mensaje "Soy el proceso <PID> a las HH:MM:SS:mmm" (utilizando la función getMilClock que se proporciona).
- **Enviará la señal SIGUSR1 al proceso padre.**

- **Terminará correctamente.**

- **El proceso padre terminará cuando todos los procesos hijos hayan terminado.**

El código del programa es el siguiente, se incluyen explicaciones a continuación:

```
void manejador (int sig) {
       if (sig == SIGUSR1) {
              printf ("Log %ld: Pid %ld: %s\n",shm_struct->logid,
(long)shm_struct->processid, shm_struct->logtext);
    }
}

int main(int argc, char *argv[]) {
    struct sigaction act;
    sigset_t set;
    sigset_t setsuspend;
    pid_t pid, ppid;
    char message[MAX_MSG];
    int i, fd_shm;
       int n, m;
       int ret = EXIT_FAILURE;

       if (argc < 3) {
              fprintf(stderr,"usage: %s <n_process> <n_logs> \n",argv[0]);
              return ret;
       }

       n = atoi(argv[1]);
       m = atoi(argv[2]);

    ppid = getpid();

    /* DEFINIMOS LAS MÁSCARAS Y EL MANEJADOR */

    sigemptyset(&(act.sa_mask));
    act.sa_flags = 0;
    act.sa_handler = manejador;
    if (sigaction(SIGUSR1, &act, NULL) < 0) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }

    sigemptyset(&set);
    sigaddset(&set, SIGUSR1);
    if (sigprocmask(SIG_BLOCK, &set, NULL) == -1) {
        perror("sigprocmask");
        exit(EXIT_FAILURE);
    }

    sigfillset(&setsuspend);
    sigdelset(&setsuspend, SIGUSR1);
```

```
    /* CREAMOS LA MEMORIA */

    fd_shm = shm_open(SHM_NAME, O_RDWR | O_CREAT | O_EXCL, S_IRUSR | S_IWUSR);
    if (fd_shm == -1) {
        fprintf(stderr, "Error creating the shared memory segment\n");
        exit(EXIT_FAILURE);
    }

    if (ftruncate(fd_shm, sizeof(ClientLog)) == -1) {
        fprintf(stderr, "Error resizing the shared memory segment\n");
        shm_unlink(SHM_NAME);
        exit(EXIT_FAILURE);
    }

    shm_struct = mmap(NULL, sizeof(*shm_struct), PROT_READ | PROT_WRITE,
MAP_SHARED, fd_shm, 0);
    if (shm_struct == MAP_FAILED) {
        fprintf(stderr, "Error mapping the shared memory segment\n");
        shm_unlink(SHM_NAME);
        exit(EXIT_FAILURE);
    }

    close(fd_shm);

    /* CREAMOS LOS HIJOS */

    for(i = 0; i < n; i++) {
        if ((pid = fork()) == -1) {
            perror("fork");
            exit(EXIT_FAILURE);
        }
        if (!pid)
            break;
    }

    /* CÓDIGO DEL HIJO */

    if(pid == 0) {
        for(i = 0; i < m; i++) {
            usleep((rand()%801+100)*1000);

            shm_struct->processid = getpid();
            shm_struct->logid++;
            sprintf(message, "Soy el proceso %ld a las ", (long)getpid());
            getMilClock(message+strlen(message));
            memcpy(shm_struct->logtext, message, sizeof(message));

            if (kill(ppid, SIGUSR1) == -1) {
                perror("kill");
                exit(EXIT_FAILURE);
            }
        }

        munmap(shm_struct, sizeof(*shm_struct));

        exit(EXIT_SUCCESS);
    }
```

```
    /* CÓDIGO DEL PADRE */

    else {
        shm_struct->logid = -1;

        do {
            sigsuspend(&setsuspend);
        } while(shm_struct->logid < n*m - 1);

        for(i = 0; i < n; i++) {
            wait(NULL);
        }

        munmap(shm_struct, sizeof(*shm_struct));
        shm_unlink(SHM_NAME);

        exit(EXIT_SUCCESS);
    }

    return ret;
}
```

Primero realizamos los preparativos del programa, estableciendo las máscaras y el manejador, set es la máscara de los procesos y setsuspend la máscara que le pasaremos a la función sigsuspend. A continuación creamos la memoria compartida, la trucamos al tamaño adecuado y la mapeamos en nuestro programa.
Creamos el número de hijos que se pasa como argumento, como la memoria ya está mapeada, no hace falta hacerlo en cada uno de los hijos.
Cada hijo ejecutará un bucle for m veces (que también se pasa como argumento), en este bucle, dormirá un tiempo aleatorio entre 100 y 900 ms, a continuación escribirá su pid en el campo processid de shm_struct, sumará 1 al campo logid, y escribira el mensaje “Soy el proceso <PID> a las HH:MM:SS:mm” en el campo logtext. Por último, enviará la señal SIGUSR1 al padre y finalizará liberando la memoria.
El padre inicializa el campo logid a -1, a continuación se suspende a la espera de la señal SIGUSR1, esto lo hará mientras el campo logid de shm_struct sea menor que n*m-1 (es decir, mientras no hayan escrito todos los hijos), a continuación esperará la finalización de todos sus hijos y acabará liberando la memoria asociada.

b) Explicar claramente en qué falla el planteamiento del ejercicio. *(0,25 ptos.)*

El principal problema por el que el ejercicio falla es debido a la falta de concurrencia. Esto se debe a que a la hora de escribir los hijos en la estructura de ClientLog se pueden dar los casos en los que dos de ellos estén accediendo a la vez. Por ejemplo, mientras un hijo incrementa el valor de logid el otro está realizando el printf por lo que imprimirá un dato erróneo. Además de que se pueden producir errores a la hora de mostrar bien los datos, nuestro diseño del programa emplea la variable logid para controlar cuántas veces se tiene que realizar el sigsuspend en el padre. Por lo tanto, si dos hijos pretenden incrementar dicha variable a la vez y el padre quiere realizar la comprobación habrá casos en los que el padre no haga suficiente

veces el sigsuspend o incluso peor, que no termine el bucle del padre debido a que algún hijo haya sobreescrito erróneamente la variable logid y no se haya incrementado correctamente.

c) Implementar un mecanismo para solucionar este problema en un nuevo programa "shm_concurrence_solved.c", que incluya un semáforo sin nombre dentro de la estructura ClientInfo. *(1,00 ptos.)*

El programa, con el problema de concurrencia solucionado sería el siguiente:

```
void manejador (int sig) {
        if (sig == SIGUSR1) {
         sem_wait(&(shm_struct->sem));
               printf ("Log %ld: Pid %ld: %s\n",shm_struct->logid,
(long)shm_struct->processid, shm_struct->logtext);
         sem_post(&(shm_struct->sem));
    }
}

int main(int argc, char *argv[]) {
    struct sigaction act;
    sigset_t set;
    sigset_t setsuspend;
    pid_t pid, ppid;
    char message[MAX_MSG];
    int i, fd_shm;
       int n, m;
       int ret = EXIT_FAILURE;

       if (argc < 3) {
               fprintf(stderr,"usage: %s <n_process> <n_logs> \n",argv[0]);
               return ret;
       }

       n = atoi(argv[1]);
       m = atoi(argv[2]);

    ppid = getpid();

    /* DEFINIMOS LAS MÁSCARAS Y EL MANEJADOR */

    sigemptyset(&(act.sa_mask));
    act.sa_flags = 0;
    act.sa_handler = manejador;
    if (sigaction(SIGUSR1, &act, NULL) < 0) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }

    sigemptyset(&set);
    sigaddset(&set, SIGUSR1);
    if (sigprocmask(SIG_BLOCK, &set, NULL) == -1) {
        perror("sigprocmask");
        exit(EXIT_FAILURE);
    }
```

```
    sigfillset(&setsuspend);
    sigdelset(&setsuspend, SIGUSR1);

    /* CREAMOS LA MEMORIA */

    fd_shm = shm_open(SHM_NAME, O_RDWR | O_CREAT | O_EXCL, S_IRUSR | S_IWUSR);
    if (fd_shm == -1) {
        fprintf(stderr, "Error creating the shared memory segment\n");
        exit(EXIT_FAILURE);
    }

    if (ftruncate(fd_shm, sizeof(ClientLog)) == -1) {
        fprintf(stderr, "Error resizing the shared memory segment\n");
        shm_unlink(SHM_NAME);
        exit(EXIT_FAILURE);
    }

    shm_struct = mmap(NULL, sizeof(*shm_struct), PROT_READ | PROT_WRITE,
MAP_SHARED, fd_shm, 0);
    if (shm_struct == MAP_FAILED) {
        fprintf(stderr, "Error mapping the shared memory segment\n");
        shm_unlink(SHM_NAME);
        exit(EXIT_FAILURE);
    }

    close(fd_shm);

    /* CREAMOS EL SEMAFORO */

    if (sem_init(&(shm_struct->sem), 1, 0) == -1) {
        perror("sem_init");
        exit(EXIT_FAILURE);
    }

    /* CREAMOS LOS HIJOS */

    for(i = 0; i < n; i++) {
        if ((pid = fork()) == -1) {
            perror("fork");
            exit(EXIT_FAILURE);
        }
        if (!pid)
            break;
    }

    /* CÓDIGO DEL HIJO */

    if(pid == 0) {
        for(i = 0; i < m; i++) {
            usleep((rand()%801+100)*1000);

            sem_wait(&(shm_struct->sem));

            shm_struct->processid = getpid();
            shm_struct->logid++;
            sprintf(message, "Soy el proceso %ld a las ", (long)getpid());
            getMilClock(message+strlen(message));
```

```
            memcpy(shm_struct->logtext, message, sizeof(message));

            sem_post(&(shm_struct->sem));

            if (kill(ppid, SIGUSR1) == -1) {
                perror("kill");
                exit(EXIT_FAILURE);
            }
        }

        munmap(shm_struct, sizeof(*shm_struct));

        exit(EXIT_SUCCESS);
    }

    /* CÓDIGO DEL PADRE */

    else {
        shm_struct->logid = -1;

        sem_post(&(shm_struct->sem));

        do {
            sigsuspend(&setsuspend);
        } while(shm_struct->logid < n*m - 1);

        for(i = 0; i < n; i++) {
            wait(NULL);
        }
        sem_destroy(&(shm_struct->sem));

        munmap(shm_struct, sizeof(*shm_struct));
        shm_unlink(SHM_NAME);

        exit(EXIT_SUCCESS);
    }

    return ret;
}
```

Empleamos el semáforo mutex en varias ocasiones asegurando así la exclusión mútua a la hora de acceder a la estructura ClientInfo:
En primer lugar, el padre hace un up del semáforo tras inicializar el campo logid a -1, como el semáforo está inicialmente a 0 y dado que los hijos hacen un down antes de modificar los campo de la estructura ClientInfo, los hijos no pueden empezar a escribir hasta que se haya inicializado todo correctamente.
Los hijos antes de realizar cualquier modificación realizan un down del semáforo y, tras haberlo hecho, hacen un up. Nos aseguramos de esta manera que la escritura de los datos no va a interferir entre los procesos hijos.
Cuando el padre entra en el manejador, también realizamos un down antes de escribir por pantalla los campos de la estructura (y un up tras realizarlo), esto lo hacemos para asegurarnos de que nos campos de la estructura no cambien a mitad del printf del padre.

Por último cabe destacar que no utilizamos semáforos para la comprobación del bucle while del sigsuspend, esto es así porque sólo utilizamos el campo logid para comprobar si ha alcanzado el número que queremos (si ha alcanzado cierta cota), y no puede haber ningún problema de concurrencia debido a esta comprobación.

<u>Ejercicio 4: Problema del Productor–Consumidor.</u> *(2,75 ptos.)* **Se pretende implementar el problema del productor–consumidor. Para ello se escribirán dos programas en C: "shm_producer.c" y "shm_consumer.c".**
El productor deberá crear la memoria compartida para almacenar una cola circular de enteros, junto con los semáforos que se consideren necesarios para gestionarla. Acto seguido, deberá generar N números entre 0 a 9 (N es el primer parámetro del programa) y los inyectará uno a uno en la cola. Para finalizar incluirá el número −1 y terminará borrando la memoria. Los números que se van a inyectar pueden ser generados aleatoriamente o en secuencia. Esto lo indicará el segundo argumento del programa: '0' para números aleatorios y '1' para secuencia.
El consumidor, se conectará a la memoria y semáforos sin crearlos y leerá de la cola, generando un histograma que refleje la cantidad de cada número que ha sido recibido, imprimiendo un listado con cada número y las veces que ha sido leído. Cuando lea el número −1, finalizará.

a) **Implementar los programas propuestos.** *(2,75 ptos.)*

PROGRAMA PRODUCTOR:

```
typedef struct {
    sem_t mutex;
    sem_t empty;
    sem_t fill;
    int stock[TAM_ALMACEN];
} Almacen;

int main(int argc, char **argv) {
    int i, fd_shm, N, R;
    Almacen *almacen_struct;

    if (argc < 3) {
        fprintf(stdout, "Se esperaban 2 argumentos de entrada\n");
        return 1;
    }

    N = atoi(argv[1]);
    R = atoi(argv[2]);

    /* CREAMOS LA MEMORIA */

    fd_shm = shm_open(SHM_NAME, O_RDWR | O_CREAT | O_EXCL, S_IRUSR | S_IWUSR);
    if (fd_shm == -1) {
        perror("shm_open");
        return 1;
    }

    if (ftruncate(fd_shm, sizeof(Almacen)) == -1) {
```

```
        perror("ftruncate");
        shm_unlink(SHM_NAME);
        return 1;
    }

    almacen_struct = mmap(NULL, sizeof(*almacen_struct), PROT_READ | PROT_WRITE,
MAP_SHARED, fd_shm, 0);
    if (almacen_struct == MAP_FAILED) {
        perror("mmap");
        shm_unlink(SHM_NAME);
        return 1;
    }

    close(fd_shm);

    /* CREAMOS LOS SEMÁFOROS */

    if (sem_init(&(almacen_struct->mutex), 1, 1) == -1) {
        perror("sem_init");
        shm_unlink(SHM_NAME);
        return 1;
    }

    if (sem_init(&(almacen_struct->empty), 1, TAM_ALMACEN) == -1) {
        perror("sem_init");
        shm_unlink(SHM_NAME);
        return 1;
    }

    if (sem_init(&(almacen_struct->fill), 1, 0) == -1) {
        perror("sem_init");
        shm_unlink(SHM_NAME);
        return 1;
    }

    /* PRODUCIMOS LOS N NÚMEROS */

    if (R == 0) {
        for(i = 0; i < N; i++) {
            sem_wait(&(almacen_struct->empty));
            sem_wait(&(almacen_struct->mutex));

            almacen_struct->stock[i%TAM_ALMACEN] = rand()%10;

            sem_post(&(almacen_struct->mutex));
            sem_post(&(almacen_struct->fill));
        }
    }

    else if (R == 1) {
        for(i = 0; i < N; i++) {
            sem_wait(&(almacen_struct->empty));
            sem_wait(&(almacen_struct->mutex));

            almacen_struct->stock[i%TAM_ALMACEN] = i%10;

            sem_post(&(almacen_struct->mutex));
            sem_post(&(almacen_struct->fill));
```

```
        }
    }

    sem_wait(&(almacen_struct->empty));
    sem_wait(&(almacen_struct->mutex));

    almacen_struct->stock[i%TAM_ALMACEN] = -1;

    sem_post(&(almacen_struct->mutex));
    sem_post(&(almacen_struct->fill));

    munmap(almacen_struct, sizeof(*almacen_struct));

    return 0;
}
```

PROGRAMA CONSUMIDOR:

```
typedef struct {
    sem_t mutex;
    sem_t empty;
    sem_t fill;
    int stock[TAM_ALMACEN];
} Almacen;

int main(int argc, char **argv) {
    Almacen* almacen_struct;
    int num[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
    int num_leido, fd_shm;
    int i = 0;

    /* ABRIMOS LA MEMORIA */

    fd_shm = shm_open(SHM_NAME, O_RDWR , 0);
    if (fd_shm == -1) {
        perror("shm_open");
        return 1;
    }

    almacen_struct = mmap(NULL, sizeof(*almacen_struct), PROT_READ | PROT_WRITE,
MAP_SHARED, fd_shm, 0);
    if (almacen_struct == MAP_FAILED) {
        perror("mmap");
        shm_unlink(SHM_NAME);
        return 1;
    }

    close (fd_shm);

    /* CONSUMIMOS LOS N NÚMEROS */

    while(1) {
        sem_wait(&(almacen_struct->fill));
        sem_wait(&(almacen_struct->mutex));

        num_leido = almacen_struct->stock[i%TAM_ALMACEN];
        if (num_leido == -1)
```

```
            break;
        num[num_leido]++;

        sem_post(&(almacen_struct->mutex));
        sem_post(&(almacen_struct->empty));

        i++;
    }

    for (i = 0; i < 10; i++) {
        fprintf(stdout, "%d : %d\n", i, num[i]);
    }

    /* LIBERAMOS LOS RECURSOS Y FINALIZAMOS */

    sem_destroy(&(almacen_struct->mutex));
    sem_destroy(&(almacen_struct->fill));
    sem_destroy(&(almacen_struct->empty));
    munmap(almacen_struct, sizeof(*almacen_struct));

    shm_unlink(SHM_NAME);

    return 0;
}
```

A la hora de la implementación hemos optado por crear una estructura en a cual almacenaremos los semáforos y la cola circular con los elementos. Esta estructura estará presente en los códigos tanto de productor como de consumidor pues en ambos reservaremos un segmento de memoria para compartir dicha estructura y que ambos programas puedan acceder a los datos.

En el código del productor inicializamos los semáforos, después dependiendo del número R que nos hayan pasado como argumento ejecutaremos un segmento de código u otro para que los números que metamos en la cola sean aleatorios o una secuencia. A la hora de acceder a la cola para insertar los números esta estará protegida por un semáforo mutex asegurando la concurrencia. Además usaremos otros dos semáforos para indicar que hemos añadido un elemento a la cola y para comprobar si está llena y no podemos introducir más.
Una vez hayamos insertado los N elementos en la cola accederemos una última vez para escribir el -1 que leerá el consumidor para saber que ya no quedan más elementos.

En el código del consumidor a parte de crear el segmento de memoria y enlazarlo, va a consistir principalmente en un bucle que se va a repetir hasta que se lea de la cola el -1.
Mientras esté leyendo los datos los meterá en una estructura de array de 10 elementos que representan cada número desde el 0 hasta el 9. Cada vez que se lea un número irá a esa posición del array y sumará uno indicando que ha aparecido en la cola y lo ha leído. Al igual que en el productor, a la hora de leer elementos de la cola deberemos proteger esta sección con semáforos para asegurar la concurrencia y usaremos los otros dos para comprobar que la cola no esté vacía y se pueda

seguir leyendo. Una vez se hayan almacenado en el array todas las veces que se ha leído cada elemento se imprimirán estos datos por pantalla.
Finalmente se liberan los recursos en ambos programas y se finaliza.

b) (Opcional) Realizar los cambios mínimos necesarios en los programas anteriores de manera que la cola esté en un fichero, en lugar de en memoria compartida, que también se enlazará al proceso con mmap. Nombrar estos nuevos programas "shm_producer_file.c" y "shm_consumer_file.c". *(Opcional; 0,50 ptos.)*

Por los pocos cambios realizados a los programas no adjuntamos el código en la memoria para no repetir (los programas funcionales están en la entrega realizada), sin embargo explicamos aquí los cambios realizados:
Cambiamos la macro `#define SHM_NAME "/shm_eje4"` por `#define FILE "almacen.txt"` y el el int fd_shm por file (simples cambios de notación).
A continuación, cambiamos las funciones shm_open por open, de tal manera que creamos el que el fichero "almacen.txt" no tiene por qué crearse en RAM (como nos asegura la función shm_open), por último, cambiamos las funciones shm_unlink por la función unlink (indicando que el fichero a eliminar no está creado en el directorio `/dev/shm` si no que es el fichero "almacen.txt").
El resto del código es igual, incluyendo truncar el tamaño del programa al que se adapte a nuestras necesidades y mapear el fichero creado en la memoria del propio programa.

<u>Ejercicio 5: Colas de Mensajes.</u> Dado el siguiente código en C: *(0,30 ptos.)*

a) Ejecutar el código del emisor, y después el del receptor. ¿Qué sucede? ¿Por qué? *(0,10 ptos.)*

Al ejecutar primero el emisor y después el receptor, el receptor recibe correctamente el mensaje imprimiendo por pantalla:

```
29: Hola a todos
```

Esto se debe a que el emisor envía el mensaje correspondiente con mq_send y, una vez enviado, el receptor lo recibe tras ejecutar mq_receive e imprime por pantalla el mensaje mostrado.

b) Ejecutar el código del receptor, y después el del emisor. ¿Qué sucede? ¿Por qué? *(0,10 ptos.)*

El código del receptor emite la solicitud de recibir un mensaje tras haber abierto la cola de mensajes, como aún no hay ninguno se queda bloqueado a la espera de recibir alguno.
Al ejecutar el emisor, esté tras abrir la cola de mensajes emite uno nuevo y el receptor que estaba bloqueado sale del bloque y lo recibe, imprimiendo por pantalla:

```
29: Hola a todos
```

Ambos esperan la introducción de cualquier tecla y finalizaran correcatamente.

c) **Repetir las pruebas anteriores creando la cola de mensajes como no bloqueante. ¿Que sucede ahora?** *(0,10 ptos.)*

Incluimos el flag `O_NONBLOCK` en la creación de la cola (tanto en el emisor como en el receptor) y ejecutamos de nuevo.
Al ejecutar primero el emisor y después el receptor no cambia nada, y de nuevo imprime el mensaje correspondiente. Esto se debe a que, aunque la cola de mensajes sea no bloqueante, cuando el receptor ejecuta mq_receive el mensaje ya ha sido enviado y por tanto puede imprimir su contenido correctamente.
Sin embargo, al ejecutar primero el receptor y después el emisor el mensaje mostrado es:

```
Error receiving message
```

La explicación es sencilla, pues el receptor ejecuta la función mq_receive, pero todavía no se ha enviado ningún mensaje, debido a que la función es no bloqueante, continúa su ejecución imprimiendo por pantalla el mensaje de error.

<u>Ejercicio 6: Pool de Trabajadores.</u> *(2,70 ptos.)* **Se pretende diseñar e implementar un pool de trabajadores que procesarán cada uno de los mensajes recibidos a través de una cola de mensajes. Aunque en este caso la tarea a realizar por los trabajadores es muy sencilla (contar la aparición de un carácter), el modelo de pool es muy utilizado y totalmente extrapolable a otros casos.**
El sistema consistirá en un programa en C, "mq_injector.c", que enviará el contenido de un fichero, y otro programa, "mq_workers_pool.c", que instanciará los procesos trabajador que procesarán los mensajes que van llegando.
En concreto, el programa "mq_injector.c" abre un fichero, cuyo nombre recibe como primer argumento, y escribe en la cola de mensajes, cuyo nombre recibe como segundo argumento, trozos del fichero de longitud máxima 2 kB, por ejemplo:
$ ./ injector fichero_entrada . txt / nombre_cola
Una vez terminado, enviará un mensaje de finalización para notificarlo.
Por otro lado, el programa "mq_workers_pool.c" recibe tres argumentos: el número de trabajadores del pool N (entre 1 y 10), el nombre de la cola de mensajes a leer, y el carácter a contar, por ejemplo:
$ ./ workers_pool 5 / nombre_cola c
Este programa satisfará los siguientes requisitos:

- **Instanciará los N trabajadores como procesos hijo.**
- **Esperará una señal SIGUSR2 que provocará la finalización de los procesos hijo mediante una señal SIGTERM.**
- **Esperará la finalización correcta de los hijos.**
- **Cerrará y liberará convenientemente todos los recursos.**

Por último, cada proceso hijo (cada trabajador):

- **Inicializará su contador a 0.**
- **Leerá el siguiente mensaje de la cola.**
    - **Si el mensaje es de finalización, enviará una señal SIGUSR2 al proceso padre.**

    - **Si no, incrementará su contador con el número de apariciones del carácter a buscar.**
- **Al recibir la señal SIGTERM, mostrará una estadística que indique el número de mensajes procesados, y el valor de su contador del carácter a buscar.**
- **Cerrará y liberará convenientemente todos los recursos.**

a) Implementar los programas propuestos. *(2,70 ptos.)*

PROGRAMA INJECTOR:

```
int main(int argc, char **argv) {
    struct mq_attr attributes = {
        .mq_flags = 0,
        .mq_maxmsg = 10,
        .mq_msgsize = TAM_MSG,
        .mq_curmsgs = 0
    };
    int file;
    ssize_t ret;
    mqd_t queue;
    char *buffer;

    buffer = malloc(TAM_MSG);
    if (buffer == NULL) {
        perror("malloc");
        return 1;
    }

    if (argc < 3) {
        fprintf(stdout, "Se esperaban 2 argumentos de entrada\n");
        free(buffer);
        return 1;
    }

    /* ABRIMOS EL FICHERO */

    file = open(argv[1], O_RDONLY, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
    if (file == -1) {
        perror("open");
        free(buffer);
        return 1;
    }

    /* CREAMOS LA COLA DE MENSAJES */

    queue = mq_open(argv[2], O_CREAT | O_WRONLY, S_IRUSR | S_IWUSR, &attributes);
    if (queue == (mqd_t)-1) {
        perror("mq_open");
        free(buffer);
        close(file);
        return 1;
    }

    /* LEEMOS DEL FICHERO Y ENVIAMOS LOS MENAJES */

    do {
```

```
        ret = read(file, buffer, TAM_MSG);
        if (ret == -1) {
            perror("read");
            free(buffer);
            close(file);
            mq_close(queue);
            mq_unlink(argv[2]);
            return 1;
        }

        if (mq_send(queue, buffer, ret, 1) == -1) {
            perror("mq_send");
            free(buffer);
            close(file);
            mq_close(queue);
            mq_unlink(argv[2]);
            return 1;
        }
    } while(ret != 0);

    /* ENVIAMOS EL MENSAJE DE FINALIZACIÓN Y TERMINAMOS */

    if (mq_send(queue, "\r", sizeof("\r"), 1) == -1) {
        perror("mq_send");
        free(buffer);
        close(file);
        mq_close(queue);
        mq_unlink(argv[2]);
        return 1;
    }

    free(buffer);
    close(file);
    mq_close(queue);
    mq_unlink(argv[2]);

    return 0;
}
```

PROGRAMA WORKERS_POOL:

```
int cuentamensaje, cuentacaracter;
char caracter;
mqd_t queue;
pid_t *cpid;
char *buffer;
char *name;

void manejador_SIGUSR2(int sig) {
    return;
}

void manejador_SIGTERM(int sig) {
    fprintf(stdout, "<%ld>: %d mensajes, %d contador\n", (long)getpid(),
cuentamensaje, cuentacaracter);
    fflush(stdout);
    free(buffer);
```

```
    free(cpid);
    mq_close(queue);
    mq_unlink(name);
    exit(EXIT_SUCCESS);
}

int main(int argc, char **argv) {
    struct mq_attr attributes = {
        .mq_flags = 0,
        .mq_maxmsg = 10,
        .mq_msgsize = TAM_MSG,
        .mq_curmsgs = 0
    };
    struct sigaction act;
    sigset_t set;
    sigset_t setsuspend;
    pid_t pid;
    pid_t ppid;
    int i, N;

    buffer = malloc(TAM_MSG);
    if (buffer == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }

    if (argc < 4) {
        fprintf(stdout, "Se esperaban 3 argumentos de entrada\n");
        free(buffer);
        exit(EXIT_FAILURE);
    }

    N = atoi(argv[1]);
    name = argv[2];
    caracter = argv[3][0];

    cpid = malloc(N*sizeof(pid_t));
    if (cpid == NULL) {
        perror("malloc");
        free(buffer);
        exit(EXIT_FAILURE);
    }

    /* DEFINIMOS LAS MÁSCARAS Y EL MANEJADOR */

    sigemptyset(&(act.sa_mask));
    act.sa_flags = 0;
    act.sa_handler = manejador_SIGUSR2;
    if (sigaction(SIGUSR2, &act, NULL) < 0) {
        perror("sigaction");
        free(buffer);
        free(cpid);
        exit(EXIT_FAILURE);
    }
    act.sa_handler = manejador_SIGTERM;
    if (sigaction(SIGTERM, &act, NULL) < 0) {
        perror("sigaction");
        free(buffer);
```

```
    free(cpid);
    exit(EXIT_FAILURE);
}

sigemptyset(&set);
sigaddset(&set, SIGUSR2);
if (sigprocmask(SIG_BLOCK, &set, NULL) == -1) {
    perror("sigprocmask");
    free(buffer);
    free(cpid);
    exit(EXIT_FAILURE);
}

sigfillset(&setsuspend);
sigdelset(&setsuspend, SIGUSR2);

/*ABRIMOS LA COLA DE MENSAJES*/

queue = mq_open(name, O_CREAT | O_RDONLY , S_IRUSR | S_IWUSR, &attributes);
if(queue == (mqd_t)-1) {
    perror("mq_open");
    free(buffer);
    free(cpid);
    exit(EXIT_FAILURE);
}

/* CREAMOS LOS HIJOS */

ppid = getpid();

for(i = 0; i < N; i++) {
    if ((pid = fork()) == -1) {
        perror("fork");
        free(buffer);
        free(cpid);
        exit(EXIT_FAILURE);
    }
    if (!pid)
        break;
    cpid[i] = pid;
}

/* CÓDIGO DEL HIJO */

if(!pid) {
    cuentacaracter = 0;
    cuentamensaje = 0;

    /* SOLICITUD DE MENSAJE */

    while(1) {
        if (mq_receive(queue, buffer, TAM_MSG, NULL) == -1) {
            perror("mq_receive");
            free(buffer);
            free(cpid);
            mq_close(queue);
            mq_unlink(name);
            exit(EXIT_FAILURE);
```

```
            }

            cuentamensaje++;

            if(buffer[0] != '\r') {
                for (i = 0; i < strlen(buffer); i++) {
                    if(buffer[i] == caracter)
                        cuentacaracter++;
                }
            }

            else {
                if (kill(ppid, SIGUSR2) == -1) {
                    perror("kill");
                    free(buffer);
                    free(cpid);
                    mq_close(queue);
                    mq_unlink(name);
                    exit(EXIT_FAILURE);
                }
            }
        }
    }

    /* CÓDIGO DEL PADRE */

    else {
        sigsuspend(&setsuspend);

        for (i=0; i<N; i++) {
            if (kill(cpid[i], SIGTERM) == -1) {
                perror("kill");
                free(buffer);
                free(cpid);
                mq_close(queue);
                mq_unlink(name);
                exit(EXIT_FAILURE);
            }
        }
        for (i=0; i<N; i++) {
            wait(NULL);
        }
    }

    exit(EXIT_SUCCESS);
}
```

El programa mq_injector se encargará de abrir el fichero y crear la cola de mensajes con la cual se comunicará con el programa mq_workers_pool.c
El injector irá leyendo el archivo de texto que le pasemos como argumento y almacenándolo en un buffer inicializado con un tamaño de 2kB. Cada vez que realice una lectura solicitará enviar un mensaje al programa receptor con el contenido que acabemos de guardar en el buffer. Realizará este proceso hasta que acabe de leer completamente el archivo de texto. Finalmente se enviará el mensaje

que indica que ha acabado de leer el documento para notificarlo a el programa receptor. Cerrará y liberará convenientemente todos los recursos para concluir.

En el programa mq_workers_pool.c abriremos la cola de mensajes cuyo nombre se nos pase como argumento y crearemos tantos hijos como nos indiquen mediante otra parámetro de la función. Los hijos realizarán un bucle en el que soliciten la llegada de un mensaje de la cola y lo almacenarán en un buffer previamente inicializado con un tamaño de 2kB. Aumentarán el contador de mensajes leídos y contarán cuántas veces aparece en el fragmento de texto recibido el carácter que se les ha pasado como argumento. Si lo que leen es el mensaje que indica el final del fichero, mandarán una señal SIGUSR2 al padre.

El padre por otro lado esperará suspendido hasta recibir la señal SIGUSR2 enviada por uno de los hijos. Cuando la reciba seguirá ejecutando su programa y enviará una señal SIGTERM a todos sus hijos. Estos al recibirla ejecutarán el conveniente manejador, el cual se encargará de imprimir los datos almacenados por cada hijo y liberar las variables usadas. El padre esperará a que todos sus hijos terminen correctamente, y acabará cerrando y liberando todos los recursos para así finalizar el programa.

b) (Opcional) Realizar los cambios mínimos necesarios en "mq_workers_pool.c" de manera que, si un trabajador no consigue leer un mensaje de la cola durante 100 ms, imprima un mensaje anunciando que no se le necesita y termine liberando todos los recursos. Nombrar este nuevo programa "mq_workers_pool_timed.c". *(Opcional; 0,50 ptos.)*

Por los pocos cambios realizados con respecto al programa mq_workers_pool.c sólo incluimos el fragmento de código y explicamos los cambios realizados.
En el fragmento de código en el que se ejecuta el mq_receive dentro del bucle, lo hemos cambiado por la función mq_timedreceive.
Esta función, además de los argumentos que se pasan a mq_receive, tiene uno adicional que es un puntero a una estructura timespec, en ella se indica el tiempo que la función tiene que esperar por un mensaje, devolviendo error una vez pasado ese tiempo.
El código de la nueva función es el siguiente:

```
if (mq_timedreceive(queue, buffer, TAM_MSG, NULL, &time) == -1) {
        if (errno == ETIMEDOUT) {
                fprintf(stdout, "<%ld>: No se me necesita\n", (long)getpid());
                fflush(stdout);
                free(buffer);
                free(cpid);
                mq_close(queue);
                mq_unlink(name);
                exit(EXIT_SUCCESS);
        }
        perror("mq_receive");
        free(buffer);
        free(cpid);
```

```
        mq_close(queue);
        mq_unlink(name);
        exit(EXIT_FAILURE);
}
```

Previamente se ha especificado en la estructura timespec time, el atributo `time.tv_nsec = 100*10^6` (pues se pasa en nanosegundos), de tal manera que si a los 100 ms no se a recibido ningún mensaje, devuelve -1. Hacemos una comprobación adicional, si `errno == ETIMEDOUT`, es decir, si el error es por finalización de tiempo, entonces lanzamos el mensaje “No se me necesita” y finalizamos liberando los recursos.