Practica 3: Sincronización de procesos con semáforos

Manuel José Corral Pazos, Grupo 1 (Jueves)

Notas previas: En esta práctica, se han creado dos códigos para uno de los ejercicios, uno correspondiente al rol de productor y el otro al de consumidor. Además, se utilizará un archivo de texto para cada uno de los ejercicios. Por ejemplo, al ejercicio 1 le corresponden los archivos 1consumidor.c, 1productor.c y 1txt.txt. En este archivo de texto se guardará una cadena de números. Estos números serán la representación del buffer en el que el productor y el consumidor realizarán sus respectivos trabajos. Para la realización de las prácticas, se tomará la siguiente notación:

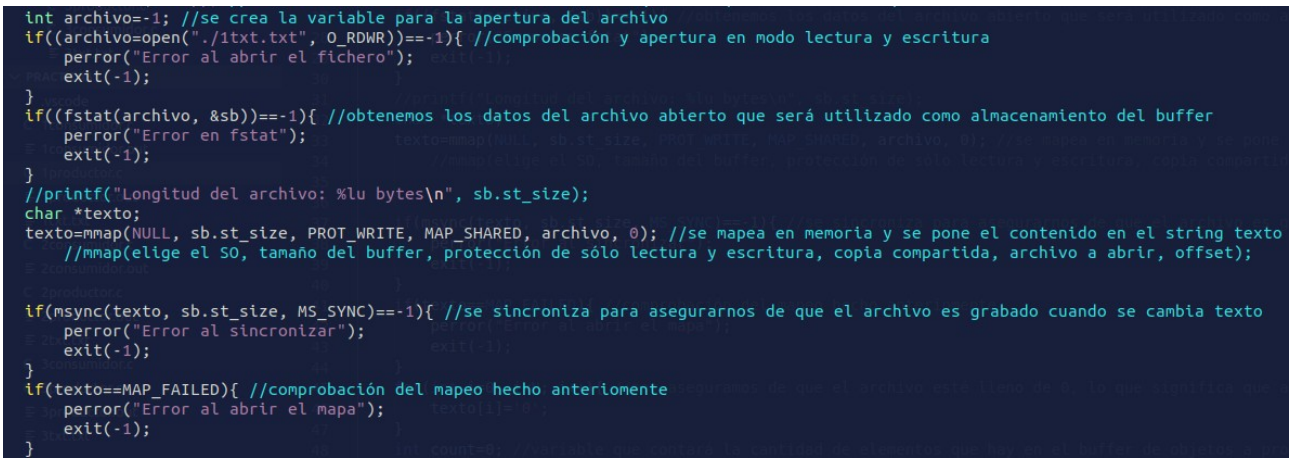
- 0 para los elementos que aún no han sido ni producidos ni consumidos
- 1 para los elementos producidos y que están presentes en el buffer
- 2 en las posiciones en las que un elemento ha sido consumido y todavía no se ha introducido ninguno por parte del productor.

Además, en el ejercicio 1 el primer número del buffer se corresponderá con la variable cuenta, ya que debido a la no utilización de semáforos este número también ha de ser compartido entre los procesos utilizando el archivo de texto.

Ejercicio 1:

En este ejercicio no se utilizarán semáforos, sino que los procesos llamarán a la función sleep en determinados puntos del código, siendo eventualmente despertados por el otro proceso. Dichos sleep se ejecutarán dependiendo de la variable “cuenta”. El productor dormirá si el buffer está lleno de objetos, mientras que el consumidor dormirá si el buffer está vacío. Para el envío de señales entre los procesos, será necesario que cada uno de los procesos conozca el PID del otro para así, utilizando la función kill, enviarle señales para que salga del sleep.

Para comenzar el programa, se abre el fichero 1txt.txt, que contendrá el buffer de elementos producidos/consumidos. El contenido de dicho archivo será almacenado por ambos programas en una variable de tipo string llamada texto, que será la que los programas actualicen. Además, se utilizará la función msync para sincronizar la actualización de la variable con la escritura del fichero.



```
int archivo=-1; //se crea la variable para la apertura del archivo
if((archivo=open("./1txt.txt", O_RDWR))==-1){ //comprobación y apertura en modo lectura y escritura
    perror("Error al abrir el fichero");
    exit(-1);
}
if((fstat(archivo, &sb))==-1){ //obtenemos los datos del archivo abierto que será utilizado como almacenamiento del buffer
    perror("Error en fstat");
    exit(-1);
}
//printf("Longitud del archivo: %lu bytes\n", sb.st_size);
char *texto;
texto=mmap(NULL, sb.st_size, PROT_WRITE, MAP_SHARED, archivo, 0); //se mapea en memoria y se pone el contenido en el string texto
    //mmap(elige el SO, tamaño del buffer, protección de sólo lectura y escritura, copia compartida, archivo a abrir, offset);


if(msync(texto, sb.st_size, MS_SYNC)==-1){ //se sincroniza para asegurarnos de que el archivo es grabado cuando se cambia texto
    perror("Error al sincronizar");
    exit(-1);
}
if(texto==MAP_FAILED){ //comprobación del mapeo hecho anteriomente
    perror("Error al abrir el mapa");
    exit(-1);
}
```

Tras esto, el proceso productor se asegurará de que el buffer inicial esté lleno de ceros, indicando que dichos elementos aún no han sido producidos ni consumidos en ningún momento.

```
for(int i=0; i<N; i++){ //nos aseguramos de que el archivo esté lleno de 0, lo que significa que aún no se ha producido ni consumido nada
    texto[i]='0';
}
```

Una vez hecho esto, se pide el PID del otro proceso y se comienzan las operaciones.
El código correspondiente al productor es el siguiente:

```
while(1){ //se entra en el bucle del programa, se debe comentar esta línea en caso de querer un bucle finito
//for(int i=0; i<3; i++){  //se debe comentar esta línea en caso de querer un bucle finito
    dormir=drand48()*2.0; //se genera un numero aleatorio con decimales entre 0.0 y 1.0 (excluidos), y se multiplica por 2
    count=(texto[0])-'0'; //se mira cuánto vale la variable cuenta al principio de la iteracion
    if(count==N) sleep(400); //en caso de que cuenta sea N, o sea, que el buffer esté lleno, se duerme y se espera a ser despertado por el consumidor
    count=(texto[0])-'0'; //se vuelve a mirar cuando vale la variable, puesto que ha sido actualizada por el consumidor
    printf("Productor:\n\tBuffer antiguo: ");
    for(int i=1; i<N; i++){ //se imprime cuánto vale el buffer cuando el productor lo lee
        printf("%c ", texto[i]);
    }
    //insertarObjeto(item);
    count=count+1; //se aumenta la cuenta en uno, puesto que se va a insertar un elemento
    texto[count]='1'; //se inserta el elemento en su sitio corresponiente, con un 1 indicando que el objeto está producido
        //el sitio en el que hay que insertar el elemento, lo da la variable count, que es la cantidad de elementos del buffer
    texto[0]=count + '0'; //se actualiza la variable cuenta en el archivo de texto
    printf("\n\tBuffer nuevo:   "); //se imprime por pantalla el resultado de la producción del elemento
    for(int i=1; i<N; i++){
        printf("%c ", texto[i]);
    }
    printf("\n\tCuenta actualizada: %d->%d\n", count-1, count); //se imprime la cuenta actualizada
    sleep(dormir); //se duerme el valor aleatorio fijado anteriormente
    if(count==1) kill(consumidor, SIGUSR1); //si la cuenta vale uno, se envía una señal al consumidor para que despierte
}
```

En primer lugar, se mira cuánto vale la variable cuenta, o sea, cuántos elementos hay producidos en el buffer. En caso de que dicha variable sea N, significará que el buffer está completamente lleno, por lo que el proceso dormirá una cantidad suficientemente grande de tiempo como para que el consumidor vacíe algún hueco. En caso de no ser así, se cambia el valor del elemento en la posición “cuenta” y se pone a uno, indicando que dicho elemento está producido y disponible para su consumición. Es importante recalcar que se cambia el elemento de la posición cuenta y no cualquier otro producido, ya que así se mantiene un orden con respecto a los elementos producidos y consumidos. Durante el proceso, se ejecutan ciertas operaciones de impresión del buffer y de la variable cuenta para ir dejando constancia de los valores implicados. En caso de que en el buffer hubiera un único elemento después de la inserción, significaría que estaba vacío antes de empezar la operación, por lo que el consumidor estaría durmiendo y sería necesario despertarlo. Por ello, se le manda la señal con la ayuda de la función kill.

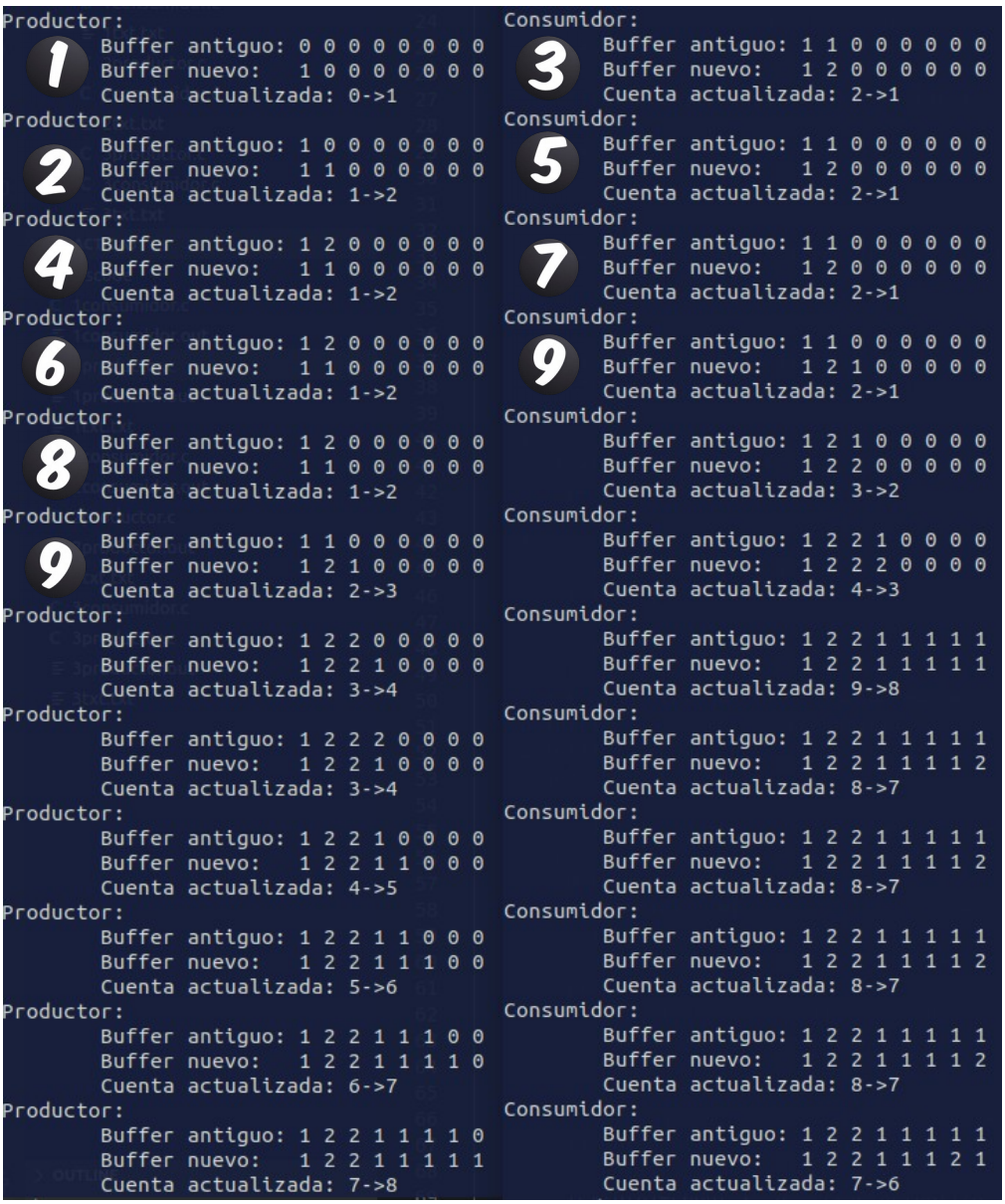
Por otra parte, el código del consumidor será el siguiente:

```
while(1){ //se entra en el bucle del programa, se debe comentar esta línea en caso de querer un bucle finito
//for(int i=0; i<3; i++){  //se debe comentar esta línea en caso de querer un bucle finito
    dormir=drand48()*2.0; //se genera un numero aleatorio con decimales entre 0.0 y 1.0 (excluidos), y se multiplica por 2
    count=(texto[0])-'0'; //se mira cuánto vale la variable cuenta
    if(count==0) sleep(400); //en caso de que valga cero, o sea, que el buffer esté vacío, se duerme hasta que el productor nos despierte
    count=(texto[0])-'0'; //se vuelve a leer la variale cuenta, puesto que ha sido modificada por el productor
    printf("Consumidor:\n\tBuffer antiguo: "); //se imprime cuánto vale el buffer antes de la consumición
    for(int i=1; i<N; i++){
        printf("%c ", texto[i]);
    }
    //item=obtenerObjeto();
    texto[count]='2'; //se marca el objeto como consumido, indicándolo con un 2
    //el sitio en el que hay que insertar el elemento, lo da la variable count, que es la cantidad de elementos del buffer
    count=count-1; //se actualiza la variable cuenta
    texto[0]=count + '0';
    printf("\n\tBuffer nuevo:   "); //se imprime el buffer después de haber consumido un objeto
    for(int i=1; i<N; i++){
        printf("%c ", texto[i]);
    }
    printf("\n\tCuenta actualizada: %d->%d\n", count+1, count); //se imprime el valor actualizado de count
    sleep(dormir); //se duerme
    if(count==N-1) kill(productor, SIGUSR1); //en caso de que al buffer le falte un elemento por llenarse, se manda una señal al productor
    // asi, el productor despertará
    //imprimirObjeto(item);
}
```

El comienzo es completamente análogo al del productor, solo que en este caso el proceso dormirá en caso de que el buffer esté vacío, es decir, si la variable cuenta vale 0. Por este motivo, a la hora de ejecutar los programas, el programa del consumidor se ejecutará antes que el del productor, porque así el consumidor estará esperando dormido a que el productor produzca algo. En caso de hacerlo a la inversa, ejecutar el productor antes que el consumidor, es posible que el productor tenga “ventaja”, puesto que se podría poner a producir incluso antes de que el consumidor estuviese en ejecución.

Una vez comienza la consumición, se cambia el elemento del buffer correspondiente a la posición cuenta (por el mismo motivo que el productor) por un 2, indicando que en esa posición ha sido consumido un elemento. Al finalizar, comprueba que si la variable cuenta vale N-1 después de consumir, significa que el buffer estaba lleno antes de realizar la operación, por lo que el productor está durmiendo. En caso de que eso ocurriese, se le enviaría una señal para que despertase.

Entrando ya en la ejecución, se introducen los PID de los respectivos procesos (primero el del productor en la ejecución del consumidor por la razón expuesta anteriormente) y se comienzan las operaciones, teniendo como salida:



En esta misma salida (donde los números indican el orden de las impresiones), se puede observar cómo se produce una carrera crítica sobre la variable “cuenta” en la salida correspondiente al número 9. En este paso, vemos cómo tanto el productor como el consumidor leen el buffer con valor “1 1 0 0 ...”. El consumidor, actualiza el valor del buffer y resulta en “1 2 1 0 0...”, mientras que el productor lo hace con valor “1 2 1 0 0...”. Esto se debe a una carrera crítica entre los procesos, puesto que los dos quieren actualizar los mismos datos al mismo tiempo. Como resultado, se tienen incoherencias en los resultados.

Ejercicio 2:
En este ejercicio se incluye la utilización de semáforos. El tamaño del buffer en este caso será igual a 10 por la especificación del enunciado de la práctica. Incluidas las librerías pertinentes, se declaran los nombres de los semáforos que se utilizarán.

```
#define mutexC "MUTEX" //definimos los nombres de los semáforos
#define emtpyC "EMTPY"
#define fullC "FULL"
```

En el caso del productor, se utiliza la función sem_unlink sobre dichos nombres, asegurándonos de que no haya semáforos en el kernel con dichos nombres antes de crearlos. Además, se abren los semáforos con los valores que a cada uno le corresponden, con sus comprobaciones de error al abrir.

```
sem_unlink(mutexC); //está indicado en el enunciado que es buena práctica borrar los semáforos antes de volver a crearlos
sem_unlink(emtpyC); //de esta forma, nos aseguramos de que no haya semáforos en el kernel con los nombres que elegimos
sem_unlink(fullC);

sem_t *mutex, *emtpy, *full; //creación de las variables de los semáforos
int intMutex=0, intEmtpy=0, intFull=0; //algunos de estos enteros se utilizarán con la función sem_getvalue;

//ahora abrimos los semáforos por primera vez con sus diferentes comprobaciones de que hayan sido abiertos correctamente
//sem_open(nombre del semáforo, opción para crear, permisos, valor incial)
//en caso de error, la función devuelve la constante SEM_FAILED
if((mutex=sem_open(mutexC, O_CREAT, 0700, 1))==SEM_FAILED)
    perror("Error al abrir mutex");
if((full=sem_open(fullC, O_CREAT, 0700, 0))==SEM_FAILED)
    perror("error al abrir full");
if((emtpy=sem_open(emtpyC, O_CREAT, 0700, N))==SEM_FAILED);
    perror("error al abrir emtpy");
```

En el caso del consumidor, simplemente se inicializan los semáforos con el mismo nombre que se les dio en el anterior programa.

```
sem_t *mutex, *emtpy, *full;
int intMutex=0, intEmtpy=0, intFull=0; // comentarios análogos al productor
//en este caso, se le pasan menos argumentos a la función sem_open porque los semáforos ya están creados en el productor
//simplemente se inicializan en este proceso, porque ya fueron creados
if((mutex=sem_open(mutexC, 0))==SEM_FAILED)
    perror("Error al abrir mutex");
if((full=sem_open(fullC, 0))==SEM_FAILED)
    perror("error al abrir full");
if((emtpy=sem_open(emtpyC, 0))==SEM_FAILED)
    perror("error al abrir emtpy");
```

En ambos programas se abre el archivo de igual modo que en el ejercicio anterior, y el productor pone ceros en el buffer.

Añadiendo semáforos, el código del productor quedaría de la siguiente forma:

```
    for(int i=0; i<100; i++){ //bucle del programa, sustituyendo el while(1) del ejercicio 1 por un bucle finito de 100 iteraciones
        dormir=drand48()*4.0; //se genera un numero aleatorio con decimales entre 0.0 y 1.0 (excluidos), y se multiplica por 4
        sleep(dormir); //se duerme fuera de la región crítica ese número aleatorio de segundos
        printf("Productor:\n\tBuffer antiguo: "); //se imprime el contenido del buffer antes de producir
        for(int i=0; i<N; i++){
            printf("%c ", texto[i]);
        }
        sem_wait(emtpy); //se baja el semáforo emtpy, indicando que hay una posición vacía menos en el buffer
        sem_wait(mutex); //se baja el semáforo mutex, indicando que vamos a entrar en la sección crítica
        // de esta forma, cualquier otro proceso con semáforos estará esperando hasta que volvamos a subir el semáforo mutex, obteniendo
exclusin mutua
        sem_getvalue(full, &intFull); //se lee el valor del semáfoor full

        texto[intFull]='1'; //se indica que el elemento correspondiente a la primera posición vacía/consumida se produce

        sem_post(mutex); //se levanta el semáforo mutex, indicando que se sale de la sección crítica
        sem_post(full); //se aumenta el semáforo full, indicando que se ha creado un nuevo elemento
        printf("\n\tBuffer nuevo:   "); //se imprime el buffer nuevo después de la producción de un elemento
        for(int i=0; i<N; i++){
            printf("%c ", texto[i]);
        }
        printf("\n\tCuenta actualizada: %d->%d\n", intFull, intFull+1); //se imprime el valor actualizado de la cuenta de elementos prod
ucidos que hay en el buffer
    }
```

En este caso, se emplean los semáforos para actualizar los valores de emtpy y full, indicando la cantidad de huecos y objetos que hay en el buffer, además de para el semáforo mutex, utilizado en la exclusión mutua. Siendo el código del consumidor el siguiente:

```
for(int i=0; i<100; i++){
    dormir=drand48()*4.0; //se genera un numero aleatorio con decimales entre 0.0 y 1.0 (excluidos), y se multiplica por 2
    sleep(dormir);
    printf("Consumidor:\n\tBuffer antiguo: ");
    for(int i=0; i<N; i++){
        printf("%c ", texto[i]);
    }
    sem_wait(full); //en este caso, se disminuye el semáforo de la cuenta de elementos, ya que se consumirá un elemento
    sem_wait(mutex); //se baja el semáforo del mutex, indicando que se entra en la región crítica de igual modo que el productor
    sem_getvalue(full, &intFull);
    texto[intFull]='2'; //se actualiza el último valor disponible para consumir del buffer, indicando con un 2 que ha sido consumido
    sem_post(mutex); //se levanta mutex
    sem_post(emtpy); //se aumenta el semáforo que indica la cantidad de elementos vacios/consumidos en el buffer
    printf("\n\tBuffer nuevo:   "); //se imprime de nuevo el buffer tras la consumición
    for(int i=0; i<N; i++){
        printf("%c ", texto[i]);
    }
    printf("\n\tCuenta actualizada: %d->%d\n", intFull, intFull-1); //se imprime el valor actualizado de los elementos que hay en el buffer
}
```

, se ve cómo ambos procesos utilizan la función sem_wait() sobre la variable mutex antes de entrar en la región crítica. Así, si uno de los dos procesos ha entrado, el otro esperará hasta que el de dentro invoque sem_post() sobre mutex, asegurando la corrección de carreras críticas. Además, el productor disminuye emtpy cuando comienza su operación, y aumenta full cuando la termina, siguiendo así con la lógica del programa; mientras que el consumidor disminuye full cuando comienza y aumenta emtpy cuando termina. De esta forma, la salida es análoga a la del apartado uno, pero no aparecen incoherencias entre las salidas de productor y consumidor.

Una vez acabado el bucle de 100 iteraciones indicado en el enunciado, se libera el mapeado y se borran los semáforos.

```
munmap(texto, sb.st_size); //se libera el mapeo de memoria
close(archivo); //se cierra el archivo y los semáforos
sem_close(mutex);
sem_close(empty);
sem_close(full);

sem_unlink(mutexC); //se eliminan los semáforos
sem_unlink(emptyC);
sem_unlink(fullC);
```

Ejercicio 3:
Similar al ejercicio 2, simplemente añadiendo lo siguiente:
Al comienzo del programa, tomamos el segundo argumento pasado como el número de productores/consumidores que deben existir.

```
int main(int argc, char *argv[]){

    int productores=0;
    if(argc==2){ //se guardan en productores la cantidad de productores a crear
        productores=atoi(argv[1]);
    }
    else{
        printf("Número de argumentos no válido. Prueba ./3productor.out <numero productores>\n");
    }
```

Para dicho número introducido, se hace un bucle de fork() para generar dicha cantidad de procesos hijos, los cuales ejecutarán la labor de productor/consumidor:

```
    pid_t pids[productores]; //vector de pids de los hijos productores

    for (int i=0;i<productores;i++) { //para cada uno de los productores deseados,se hace un proceso hijo que llevará a cabo dicha funci
ón de productor
        if((pids[i]=fork())==0) {
            for(int j=0; j<5; j++){
                dormir=drand48()*4.0; //se genera un numero aleatorio con decimales entre 0.0 y 1.0 (excluidos), y se multiplica por 2
                sleep(dormir);
                printf("%d\tBuffer antiguo: ", getpid());
                for(int i=0; i<N; i++){
                    printf("%c ", texto[i]);
                }
                sem_wait(emtpy);
                sem_wait(mutex);
                sem_getvalue(full, &intFull);

                texto[intFull]='1';

                sem_post(mutex);
                sem_post(full);
                printf("\n%d\tBuffer nuevo:   ", getpid());
                for(int i=0; i<N; i++){
                    printf("%c ", texto[i]);
                }
                //sem_getvalue(full, &intFull);
                printf("\n%d\tCuenta actualizada: %d->%d\n", getpid(),intFull+1, intFull);
            }
            exit(i);
        }
```

Además, el proceso padre espera por los hijos para liberar el mapeado de memoria, cerrar el archivo y borrar los semáforos.

```
    int status;
    pid_t pid;
    int c=productores;
    while (c > 0) {
        pid = wait(&status);
         c--;
    }

    munmap(texto, sb.st_size);
    close(archivo);
    sem_close(mutex);
    sem_close(emtpy);
    sem_close(full);

    sem_unlink(mutexC);
    sem_unlink(emtpyC);
    sem_unlink(fullC);
```