Informe Del Ejercicio 2

1. Introducción.

Para solucionar el clásico problema del productor-consumidor, en el que dos procesos (uno productor y otro consumidor) comparten un buffer compartido de tamaño limitado, generando problemas al haber carreras críticas, ya que acceden los dos de manera simultánea sin ningún control, se utilizan mecanismos de sincronización que eliminan estas carreras críticas, en nuestro caso utilizamos los semáforos.

En 1965, E.W. Dijkstra sugirió utilizar una variable entera que permite que dos procesos se coordinen, la cual se llamó semáforos. Los semáforos tienen dos operaciones: “down”, disminuye el valor del semáforo y bloquea el proceso si su valor es 0, y “up”, incrementa el valor del semáforo y despierta un proceso que está bloqueado, si lo hay. Todas estas acciones que realizan los semáforos son operaciones atómicas indivisibles, lo que garantiza que una vez que una operación de un semáforo empieza, ningún proceso podrá acceder al semáforo hasta que se complete o se bloquee.

En esta implementación el productor genera elementos y los mete en el buffer solo si hay sitio, y el consumidor retira elementos del buffer solo si el buffer no está vacío. Al usar los semáforos nos evitamos la pérdida de datos, o que sean incorrectos, al tener los procesos con memoria compartida sincronizados, garantizando un acceso seguro y ordenado, resolviendo así que aparezcan carreras críticas.

2. Ejecución.

La implementación en C está dividida en dos programas: prod_sem.c (el productor implementando semáforos) y cons_sem.c (el consumidor implementando semáforos). Generando los ejecutables compilando ambos programas con estos comandos:

```
gcc -o prod_sem prod_sem.c
gcc -o cons_sem cons_sem.c
```

Ejecutando primero el productor y luego el consumidor, en diferentes terminales, de esta manera:

```
./prod_sem
```

```
./cons_sem
```

3. Funcionamiento.

El productor genera letras aleatorias y las almacena en la pila de la memoria compartida, antes de añadir un nuevo elemento a la pila comprueba que haya espacio disponible mediante el semáforo VACÍAS, comprueba que la región crítica no está ocupada mediante el semáforo MUTEX. Una vez se inserta el elemento se incrementa el índice del tope de la pila. El programa realiza 60 iteraciones (sustituyendo al lazo infinito del ejercicio anterior), en las cuales realiza las siguientes operaciones de manera ordenada: general una letra aleatoria, espera un tiempo aleatorio (entre 0 y 3 segundos, para generar situaciones en las que el buffer se llena o se vacía), espera empleando semáforos a que haya espacio disponible (usando:

sem_wait(vacias)), espera empleando semáforos hasta poder acceder a la región crítica (memoria compartida) (usando: sem_wait(mutex)), introduce el elemento en la pila y actualiza el valor del tope, indica usando semáforos al consumidor que abandona la región crítica (usando: sem_post(mutex)) y, por último, indica al consumidor con semáforos que hay un nuevo elemento en el buffer. En este programa se maneja correctamente la inicialización y eliminación de los semáforos para evitar fugas.

El consumidor retira elementos de la pila, antes de retirar comprueba que hay elementos disponibles (LLENAS) y consigue el acceso a la región crítica usando (MUTEX). Este programa también tiene 60 iteraciones en las que realiza las siguientes acciones: espera un tiempo aleatorio (entre 0 y 3 segundos, para generar situaciones en las que el buffer se llena o se vacía), espera usando semáforos a que haya algún elemento en la pila (usando: sem_wait(llenas)), espera usando semáforos para acceder a la región crítica (usando: sem_wait(mutex)), retira el elemento del buffer y cambia el valor del tope, indica usando semáforos que se a retirado un elemento de la pila (usando: sem_post(mutex)) y, al final, almacena este elemento en una cadena local.

En esta ejecución hemos reducido el número de iteraciones de 60 a 20, ya que se puede ver bien su funcionamiento y es más visible:

```
joel@joel-Victus:~/Escritorio/uni/Sistemas Operativos II/practica2/codigos$ ./prod_sem
Introduciendo T en la posicion 0
Introduciendo O en la posicion 1
Introduciendo F en la posicion 0
Introduciendo N en la posicion 0
Introduciendo T en la posicion 0
Introduciendo S en la posicion 0
Introduciendo L en la posicion 0
Introduciendo V en la posicion 1
Introduciendo Y en la posicion 1
Introduciendo E en la posicion 1
Introduciendo N en la posicion 1
Introduciendo J en la posicion 1
Introduciendo H en la posicion 2
Introduciendo T en la posicion 1
Introduciendo E en la posicion 2
Introduciendo V en la posicion 2
Introduciendo B en la posicion 3
Introduciendo Q en la posicion 4
Introduciendo H en la posicion 5
Introduciendo L en la posicion 5
Contenido de str: TOFNTSLVYENJHTEVBQHL
```

```
joel@joel-Victus:~/Escritorio/uni/Sistemas Operativos II/practica2/codigos$ ./cons_sem
Retirando O de la posicion 1
Retirando T de la posicion 0
Retirando F de la posicion 0
Retirando N de la posicion 0
Retirando T de la posicion 0
Retirando S de la posicion 0
Retirando V de la posicion 1
Retirando Y de la posicion 1
Retirando E de la posicion 1
Retirando N de la posicion 1
Retirando H de la posicion 2
Retirando J de la posicion 1
Retirando E de la posicion 2
Retirando H de la posicion 5
Retirando L de la posicion 5
Retirando Q de la posicion 4
Retirando B de la posicion 3
Retirando V de la posicion 2
Retirando T de la posicion 1
Retirando L de la posicion 0
Contenido de str: OTFNTSVYENHJEHLQBVTL
```

Al ejecutar los dos procesos en paralelo se puede observar un correcto funcionamiento del intercambio de los elementos entre ambos procesos. Confirmándose que la sincronización utilizando semáforos funciona correctamente, evitando los problemas del ejercicio 1, asegurando una correcta producción y consumo de los elementos.

4. Conclusión.

Este ejercicio nos muestra que el uso de semáforos de POSIX (específicamente usando las funciones sem_wait() y sem_post(), y los semáforos VACIAS, LLENAS y MUTEX) nos permite resolver los problemas que se generaban con las carreras críticas del problema del productor-consumidor, haciendo que los procesos estén correctamente sincronizados con consistencia en los datos.