**Centro Universitario de Ciencias Exactas e Ingenerias**

**Departamento de Ciencias Computacionales**

**Seminario de Sistemas Operativos**



**Violeta del Rocio Becerra Velazquez**

**Edgar Agustin Martinez Gonzalez**

**220286695**

**Ingeneria en Computacion**

**D01**

**Programa 6. Productor-Consumidor**

**06/11/2022**

**Introduccion**

**El problema**

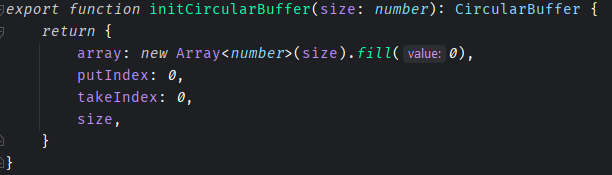
El programa describe dos procesos, productor y consumidor, ambos comparten un buffer de tamaño finito. La tarea del productor es generar un producto, almacenarlo y comenzar nuevamente; mientras que el consumidor toma (simultáneamente) productos uno a uno. El problema consiste en que el productor no añada más productos que la capacidad del buffer y que el consumidor no intente tomar un producto si el buffer está vacío.

**Soluciones**

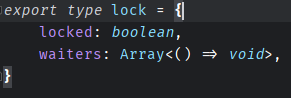
La idea para la solución es la siguiente, ambos procesos (productor y consumidor) se ejecutan simultáneamente y se “despiertan” o “duermen” según el estado del buffer. Concretamente, el productor agrega productos mientras quede espacio y en el momento en que se llene el buffer se pone a “dormir”. Cuando el consumidor toma un producto notifica al productor que puede comenzar a trabajar nuevamente. En caso contrario, si el buffer se vacía, el consumidor se pone a dormir y en el momento en que el productor agrega un producto crea una señal para despertarlo. Se puede encontrar una solución usando mecanismos de comunicación de interprocesos, generalmente se usan semáforos.

**Desarrollo**

El buffer circular a usar almacena un array del tamano que necesitemos, dos indices para la posicion en donde se insertan y en donde se toman elementos del buffer y tambien guardamos el tamano para poder ciclar el buffer de manera circular.

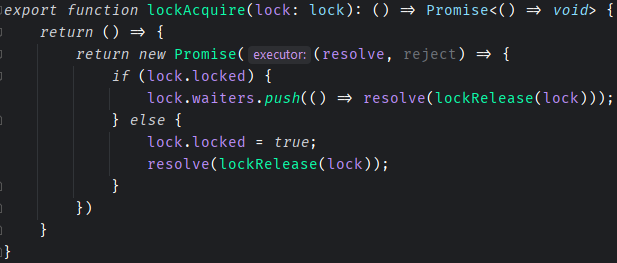


El semaforo que usaremos nos permite bloquear un proceso si el semaforo ha sido tomado y reanudar el proceso hasta que se libere.

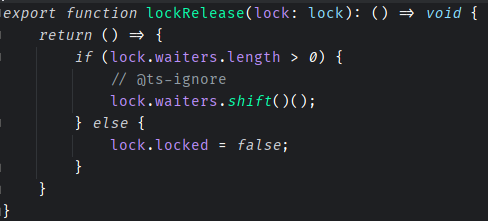


En la funcion *acquire* para tomar el semaforo se regresa una promesa pendiente en caso de no estar disponible y agregamos dicha promesa a la lista de espera, de esta forma el proceso que haya intentado tomar el semaforo podra esperarse hasta que la promesa se resuelva y entonces seria seguro acceder al buffer.

Si el semaforo no esta tomado entonces se marca como tomado y se regresa una promesa ya resuelta que haria que inmediatemente el proceso continue sabiendo que es seguro usar el buffer.



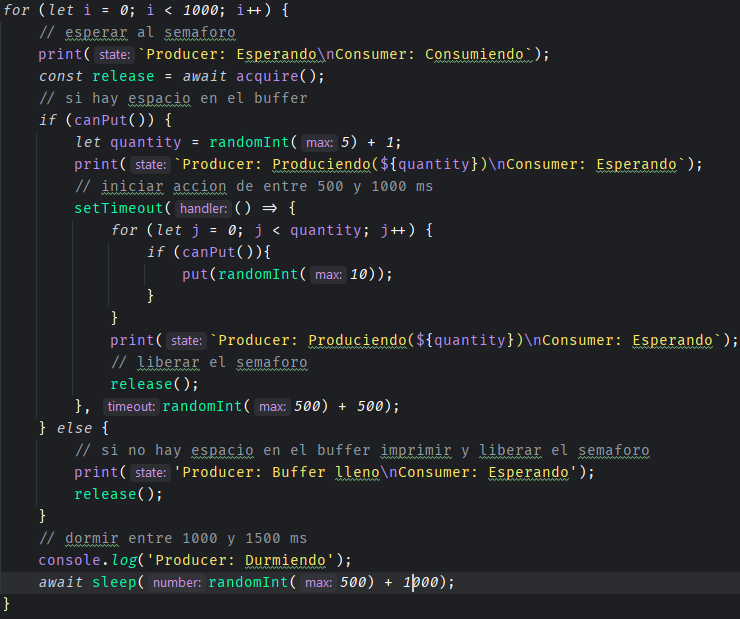
La promesa que se obtiene al tomar el semaforo incluye el metodo *release* para soltar dicho semaforo. Cuando esta funcion se llama el semaforo tomara la siguiente promesa de la lista de espera y la resolvera, indicando que es el turno del proceso esperando en dicha promesa para acceder al buffer de forma segura



Finalmente, en el producer y consumer se utilizan dos metodos distintos donde su unica diferencia relevante es que el producer utiliza el metodo put y canPut para realizar su accion y el consumer utiliza el metodo take y canTake.

Cada uno entrara en un ciclo donde intentan tomar el semaforo, se marca com esperando hasta tomarlo y entonces si puede realizar su accion lo realiza con entre 1 a 5 elementos de la lista y esto le toma entre .5 a 1 segundo, una vez que termina suelta el semaforo y se duerme entre .5 a 1.5 segundos.

Si no hay espacio o elementos para hacer su accion segun sea el caso, entonces inmediatamente libera el semaforo y se va a dormir.



**Conclusion**

El semaforo es un concepto muy ingenioso y sencillo de implementar para evitar las condiciones de carrera en un programa. No es un metodo que haga al programa 100% seguro contra estas condiciones por que a un programador se le podria olvidar el esperar o incluso tomar el semaforo antes de usar un recurso compartido y el programa igual funcionaria pero eventualmente se tendria un error, de cualquier forma si se tiene un conocimiento claro de los semaforos es mas intuitivo encontrar estos errores por la sencillez del concepto.