Sistemas Operativos II

Práctica 2 - Informe

Sincronización de procesos con semáforos

Xiana Carrera Alonso
Curso 2021-2022

28/03/2022

ÍNDICE

1 Introducción

1.1 Estructura de la práctica

Esta práctica está destinada al análisis de los conceptos de carreras críticas, regiones críticas, exclusión mutua entre procesos e hilos y, especialmente, en las variables semáforo y su uso a la hora de impedir la ocurrencia de carreras críticas.

Tal estudio se realiza a través del problema del productor-consumidor, un ejemplo clásico en el área de comunicación entre procesos ([**AfterAcademy**]).

La práctica se divide en 3 ejercicios:

Ejercicio 1

Esta primera cuestión requiere una implementación preliminar del problema empleando procesos y espera activa para la sincronización de estos. Corresponde a una variante con respecto al algoritmo visto en clase, que se puede consultar en [**Tanenbaum**].

A través de él se analizará en qué consiste una carrera crítica en el problema del productor-consumidor, cuál es la región crítica asociada y en qué condiciones tiene lugar.

Los detalles de implementación (véase la estructura del buffer compartido entre procesos, el tipo de dato que este almacena, etc.) se dejan a elección del alumno.

Ejercicio 2

Este apartado da solución a la situación presentada en el ejercicio 1 mediante el empleo de tres semáforos. Un uso controlado y sincronizado de ellos por los procesos productor y consumidor permite evitar la aparición de carreras críticas.

En este caso, se requiere un seguimiento más pautado de las condiciones de implementación. Se exige la utilización de una cola FIFO para representar el buffer, así como la definición de diferentes funciones y el uso de *sleep()* en puntos específicos del programa.

Ejercicio 3

En este apartado se implementa una variante del ejercicio 2 utilizando hilos en lugar de procesos, probando que esta opción también es posible.

1.2 Descripción del problema del productor-consumidor

```
#define N 100                                   /* número de ranuras en el búfer */
int cuenta = 0;                                 /* número de elementos en el búfer */

void productor(void)
{
        int elemento;

        while (TRUE) {                          /* se repite en forma indefinida */
                elemento = producir_elemento();         /* genera el siguiente elemento */
                if (cuenta == N) sleep();               /* si el búfer está lleno, pasa a inactivo */
                insertar_elemento(elemento);            /* coloca elemento en búfer */
                cuenta = cuenta + 1;                    /* incrementa cuenta de elementos en búfer */
                if (cuenta == 1) wakeup(consumidor);    /* ¿estaba vacío el búfer? */
        }
}


void consumidor(void)
{
        int elemento;

        while (TRUE) {                          /* se repite en forma indefinida */
                if (cuenta == 0) sleep();               /* si búfer está vacío, pasa a inactivo */
                elemento = quitar_elemento();           /* saca el elemento del búfer */
                cuenta = cuenta – 1;                    /* disminuye cuenta de elementos en búfer */
                if (cuenta==N–1) wakeup(productor);     /* ¿estaba lleno el búfer? */
                consumir_elemento(elemento);            /* imprime el elemento */
        }
}
```

***Figura 1**. Implementación original del problema por [**Tanenbaum**].*

En este planteamiento se consideran dos procesos, un productor y un consumidor, que comparten un *buffer* de tamaño fijo (N). El productor se encarga de introducir nuevos elementos en el *buffer*, mientras que el consumidor es responsable de eliminarlos. Supondremos que dicha región se trata de una pila LIFO.

Para evitar que el productor introduzca un número de elementos que sobrepase el tamaño del *buffer*, o que el consumidor trate de eliminar ítems cuando este está vacío, se debe garantizar que el primero quede bloqueado cuando el número de ítems, que se representa mediante la variable *cuenta*, es exactamente N. Asimismo, el consumidor quedará bloqueado cuando *cuenta* valga 0.

En el código que indica *[**Tanenbaum**]*, este control se establece a través de funciones *sleep()* y *wakeup()*, de implementación no especificada. Los procesos se bloquearían a sí mismos utilizando *sleep()* y serían responsables de despertarse mutuamente empleando *wakeup()*, pues cada uno de ellos rompe la condición de bloqueo del otro.

El control del número de elementos presentes es el *buffer* es el responsable de la ocurrencia de carreras críticas. La actualización de *cuenta* (que el productor aumenta en 1 en cada iteración, y que el consumidor disminuye en 1) no es atómica con respecto a las funciones *insertar_elemento()* y *quitar_elemento()*. Por ende, puede darse la siguiente situación:

Supongamos que *cuenta* es 3, N es 6 y el productor acaba de ejecutar *insertar_elemento()*, escribiendo un nuevo ítem en la posición 3 del *buffer* (supondremos que se deja como "A B C D X X" y que está numerado de 0 a 5). En ese punto, justo al ejecutar la línea "cuenta = cuenta + 1", lee cuenta, lo incrementa y no llega a escribir el resultado, sino que tiene lugar un cambio de contexto y el control de la CPU pasa al consumidor. Este no verá que *cuenta* es 4, pues aún no está actualizada, sino que leerá *"cuenta = 3"*. Por tanto, cuando ejecute *quitar_elemento()*, eliminará el

que se encuentre en la posición 2 del *buffer*, en lugar de el que acaba de introducir el productor en la posición 3. El *buffer* quedaría entonces como "A B X D X X". El consumidor podría entonces reducir *cuenta* a 2, pero en cuanto el control de la CPU vuelva al productor, este sobreescribirá dicho valor como el que había guardado localmente, 4.

Por consiguiente, quedará registrado que hay 4 elementos en el *buffer*, cuando solo hay 3 y además se tiene un espacio en blanco del que tanto productor como consumidor son desconocedores. No solo es una situación incorrecta, sino que puede dar lugar a fallos críticos cuando alguno de los procesos trate de manejar la posición 2, en función del tipo de datos empleado.

La condición de carrera crítica para el consumidor es análoga, cambiando *insertar_elemento()* por *quitar_elemento().* Partiendo de la situación anterior e invirtiendo los papeles de los procesos, quedaría una situación como "A B X D X X", esta vez con *cuenta = 2*, de modo que el elemento D guardado por el productor acabaría siendo sobreescrito más adelante por él mismo.

En el código propuesto por ***[Tanenbaum]*** es posible que el productor o el consumidor queden dormidos indefinidamente, si tiene lugar una interrupción entre la lectura de *cuenta* en *if (cuenta == N)* para el productor (o *if (cuenta == 0)* para el consumidor) y la llamada a *sleep().* Si por ejemplo el productor ve que *cuenta* es N y entonces hay una interrupción, tras la cual el consumidor retira un ítem y ejecuta un *wakeup()* sin efecto, el productor quedará inmediatamente dormido cuando retome el control. Eventualmente, el consumidor retirará todos los elementos del *buffer* y también quedará bloqueado indefinidamente.

Nótese que en la implementación desarrollada en el ejercicio 1 los procesos no quedarán dormidos indefinidamente por el uso de exclusión mutua, que obliga a comprobar periódicamente la condición de bloqueo y saca a los procesos del *while* asociado en cuanto esta no se satisface.

No obstante, es evidente que aquí se presentan igualmente problemas de sincronización entre procesos. En la siguiente sección, destinada al ejercicio 1, analizaremos los resultados experimentales obtenidos.

2 Ejercicio 1

Como se mencionaba en el apartado anterior, en esta implementación no tendremos problemas de bloqueos incontrolados, al emplear espera activa. Por consiguiente, la región crítica del código se restringe a las secciones indicadas en las figuras 2 y 3.

***Figuras 2 y 3**. Regiones críticas del productor y del consumidor, respectivamente.*

Cabe destacar que en la consideración de la región crítica habría que incluir también las líneas intermedias de impresión en un *log* y el posible *sleep()*. Sin embargo, dado que estas líneas son superfluas, sirven únicamente para control del usuario y no se consideran en el problema original, podrían ignorarse. En tal caso, únicamente hay dos líneas conflictivas en cada proceso, las marcadas en rojo.

En efecto, en las salidas del código podemos comprobar que tienen lugar las carreras críticas descritas (obsérvese que el *buffer* se ha implementado como una pila):

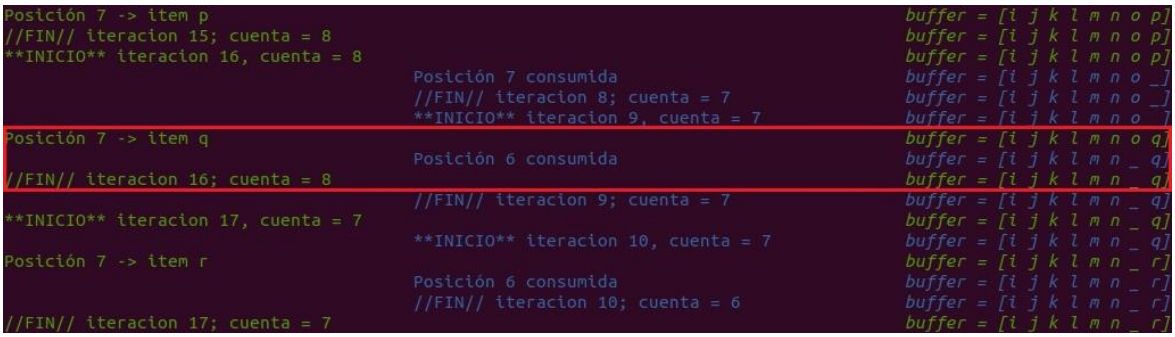


***Figura 4**. Ejemplo de carrera crítica en el ejercicio 1.*

En el caso de las líneas destacadas en la figura 4, vemos que lo que está ocurriendo es que el productor (verde) introduce un nuevo elemento, q, pero no llega a actualizar *cuenta* a 8 instantáneamente, sino que el turno pasa al consumidor (azul). Este lee *cuenta = 7* y elimina el elemento de la posición 6, o. Una vez el turno vuelve al productor, este escribe *cuenta = 8*, pero el daño ya ha sido ocasionado: ahora el *buffer* cuenta con un espacio inutilizado.

La acumulación de este tipo de situaciones puede llevar a finales de ejecución como el de la figura 5, cuando el número de iteraciones totales es alto. Vemos que el número de huecos es grande y el *buffer* está en una situación totalmente incorrecta.

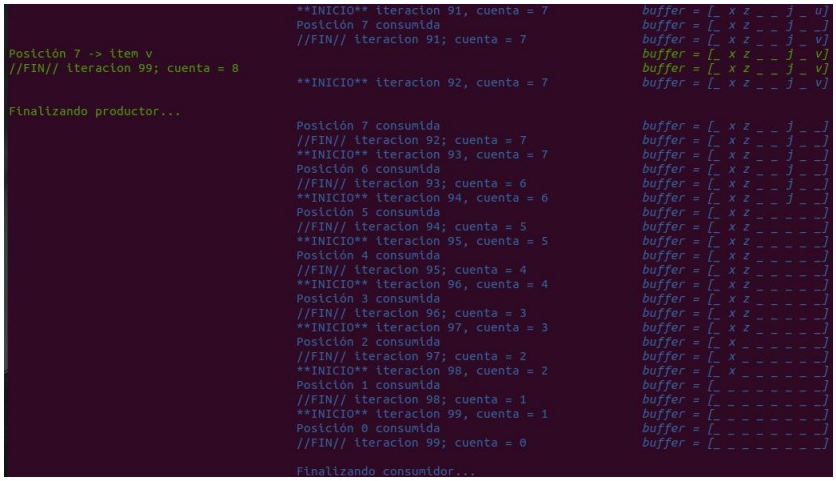


***Figura 5**. Ejemplo de un final de ejecución en el ejercicio 1.*

Cabe destacar la importancia del número de iteraciones y del proceso hijo que comienza a ejecutarse antes. En los ejemplos anteriores se estaban realizando 100 iteraciones y era el consumidor el que comenzaba. De este modo, aseguramos que ambos procesos hijos empiecen sus labores aproximadamente a la vez, pues el consumidor tendrá que esperar a que se cree el productor y este inserte elementos para comenzar a actuar. Si comienza antes el productor, ya habrá insertado un gran número de elementos para cuando se cree el consumidor. Si el número de iteraciones es bajo, puede que ni siquiera lleguen a confluir, como ocurre en la figura 6:

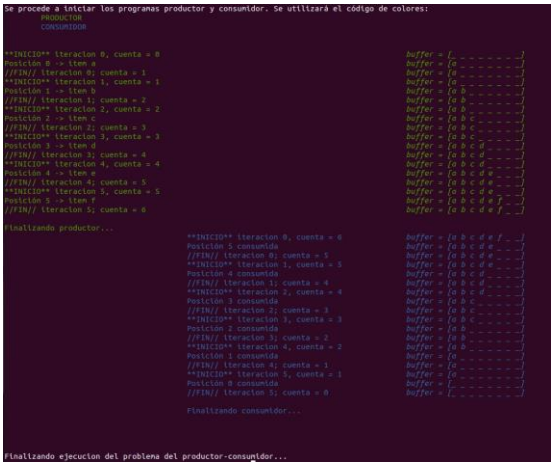


***Figura 6**. Adelantamiento del productor con número bajo de iteraciones.*

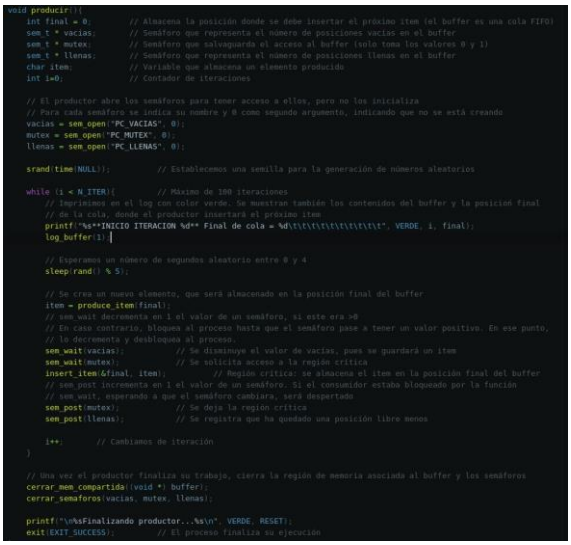
En este tipo de situaciones, lo más recomendable para apreciar carreras críticas es ralentizar uno de los procesos o ambos mediante llamadas a *sleep()*, de forma que se alargue la duración de la región crítica y aumente la probabilidad de que se produzcan interrupciones en ellas. Este control se realiza mediante el argumento *ralentizar* en las funciones del productor y del consumidor (véanse las figuras 2 y 3), que representa el número de segundos que debe durar el *sleep().*

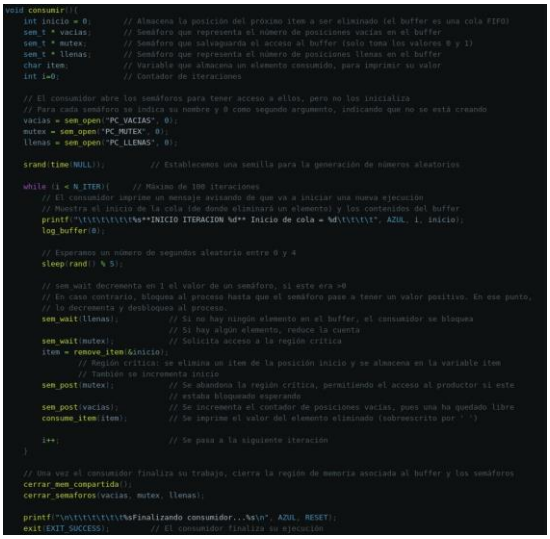
Observamos que cuando se duerme a los procesos sí aumenta la probabilidad de carreras críticas. No obstante, cuando uno de ellos es más rápido de manera fija (si su

sleep() dura menos), el *buffer* tenderá a estar siempre casi lleno (si el más rápido es el productor) o casi vacío (si el más rápido es el consumidor), que puede no resultar demasiado esclarecedor. Si, en cambio, son igual de rápidos, el *buffer* tenderá a variar únicamente en 1 o 2 posiciones. Por tanto, se recomienda generar un número de segundos de espera aleatorio, como en el ejercicio 2.

Cabe destacar que en este ejercicio las variables compartidas son *cuenta* y *buffer*. Esta compartición se implementa a través de la reserva de un área de memoria con *mmap(),* que almacena un entero (*cuenta*) y N caracteres.

3 Ejercicio 2





***Figuras 7 y 8**. Códigos del productor y consumidor con semáforos.*

En el ejercicio 2 se da solución al problema mediante el uso de semáforos, variables que toman valores enteros positivos y que gestionan el acceso del productor y del consumidor a la región crítica, además de servir a modo de contadores.

Uso de semáforos

Los semáforos empleados son 3:

- *vacias*, que toma valores de 0 a N y lleva la cuenta del número de posiciones vacías en el *buffer*. Puede impedir el acceso para el productor.
- *mutex*, que toma los valores 0 y 1 y restringe el acceso a la región crítica. Puede bloquear tanto al productor como al consumidor.
- *llenas*, que toma valores de 0 a N y lleva la cuenta del número de posiciones ocupadas en el *buffer*. Puede impedir el acceso al consumidor.

Los semáforos se gestionan mediante las funciones *sem_wait()* y *sem_post()*.

sem_wait() comprueba si el valor del semáforo pasado como argumento es mayor que 0 y, si es así, disminuye su valor. En caso contrario (es 0), deja bloqueado al proceso hasta que el valor sea positivo, y entonces lo disminuye y despierta al proceso. La clave de esta función es que es atómica, de modo que ningún otro proceso puede acceder al semáforo y/o modificarlo mientras la función esté en ejecución. Esta es la base del funcionamiento de los semáforos, que provoca que se puedan evitar las carreras críticas.

sem_post() simplemente aumenta en 1 el valor del semáforo pasado como argumento. Esta acción podrá desbloquear a otros procesos que se encuentren bloqueados por un *sem_wait()* asociado a ese semáforo. En concreto, si hay varios en esa situación, el sistema operativo seleccionaría uno al azar y lo haría finalizar el *sem_wait()* y despertarse.

Los semáforos deben crearse mediante la operación *sem_open(),* que les asigna un nombre identificativo, les da permisos y guarda en ellos un valor inicial (*vacías,* N; *mutex,* 1; *llenas,* 0). Esta acción, en la implementación realizada para el ejercicio 2, la realiza el proceso padre. Después, los procesos hijos solicitan acceso a los semáforos volviendo a ejecutar *sem_open(),* pero esta vez sin indicar la opción de creación.

Una vez se dejan de utilizar los semáforos, cada proceso los cierra mediante *sem_close().* Cuando finaliza su uso de forma definitiva, se destruyen con *sem_unlink()* (lo realiza el padre al final de la ejecución y también al principio, para eliminarlos si ya existían previamente).

Control de la región crítica

En primer lugar, cabe destacar que ahora el *buffer* será una cola FIFO. El uso de una variable global *cuenta* se ve sustituido por dos variables locales, *inicio* para el productor y *fin* para el consumidor, que indican las posiciones del *buffer* donde se deben realizar la próxima inserción y eliminación, respectivamente. La gestión de los límites del *buffer* se realiza a través de estas variables y de los semáforos.

Con este nuevo planteamiento, la región crítica del productor pasa a estar contenida en la función *insert_item(),* y la del consumidor, en *remove_item().* Para protegerlas, cada proceso solicita acceso a ellas antes de entrar ejecutando *sem_wait()* sobre el semáforo *mutex,* de forma que se le impide el acceso si el otro proceso está ejecutando su región crítica. *mutex* se desbloquea mediante el *sem_post()* que ejecuta cada proceso al salir de la región crítica, de forma que siempre valdrá 0 o 1.

Además, el productor se asegura de que el *buffer* no esté lleno ejecutando *sem_wait()* sobre *vacias*. Si el número de posiciones vacías era 0, quedará esperando a que se liberen huecos. Análogamente, el consumidor se asegura de que el *buffer* no esté vacío comprobando *llenas.* Al salir de la región crítica, el productor incrementa *llenas*, desbloqueando al consumidor si procede, y el consumidor hace lo mismo con *vacias.*

Resultados

Con esta implementación, observamos que, en efecto, dejan de producirse carreras críticas, incluso aunque realicemos llamadas a *sleep()* de forma aleatoria (en el código de las figuras aparece una única llamada, pero también se ha probado con varias, con resultados similares). Se puede ver en la figura 9 una prueba de ejecución (sin dormir a los procesos, para llenar más el *buffer*):

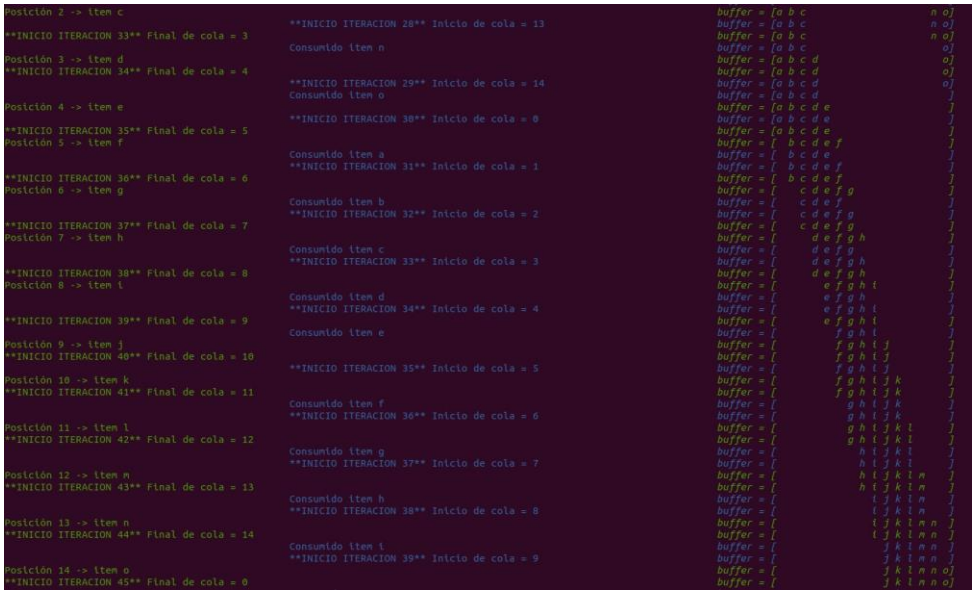


***Figura 9**. Ejecución del ejercicio 2.*

Cambios en la velocidad de los procesos dan lugar a ejecuciones con el *buffer* más o menos lleno.

4 Ejercicio 3

Para este ejercicio se ha realizado una adaptación del código del ejercicio 2 para hilos en lugar de procesos. Las bases teóricas y la mayoría de detalles de implementación son los mismos. Se comentan a continuación las puntualizaciones más significativas:

- Las funciones de creación, espera, etc. de procesos (como *fork()* y *waitpid())* se sustituyen por las análogas de hilos (*pthread_create(), pthread_join(), etc.*).
- Los semáforos deben seguir abriéndose y cerrándose en cada hilo. No es posible acceder directamente a ellos sin ejecutar *sem_open()* desde los hilos hijos.
- Dado que los hilos comparten espacio de direcciones, ya no es necesario reservar un área de memoria compartida para el *buffer*. En esta implementación se ha optado por reservar memoria dinámica a través de *malloc().*
- Únicamente es necesario fijar una semilla para la generación de números aleatorios una vez, desde el hilo padre, en lugar de para cada proceso.

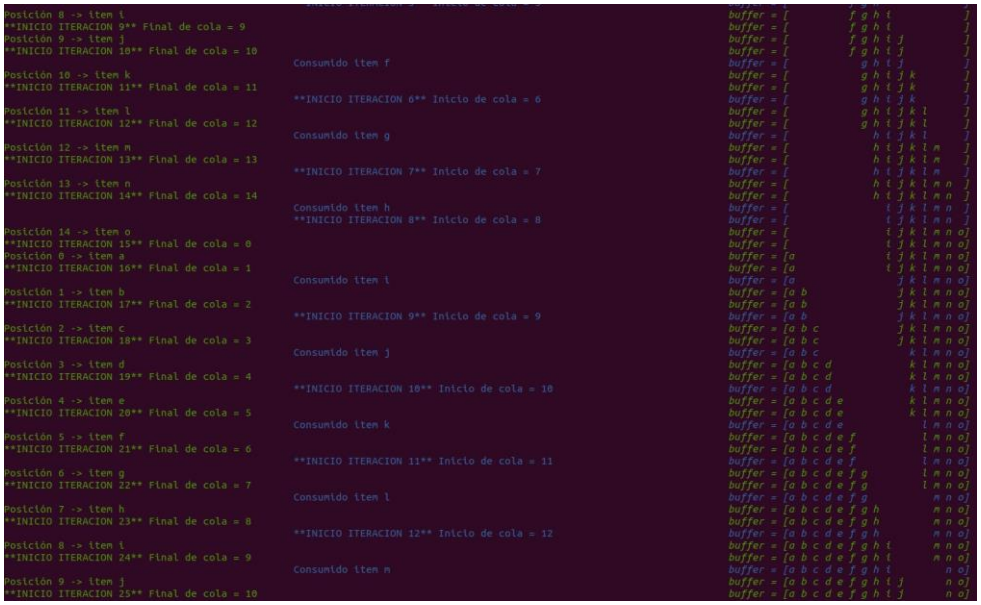


***Figura 10**. Ejecución del ejercicio 3. Obsérvese que por el uso de un segundo sleep() en el consumidor, este es más lento, y el buffer se va llenando progresivamente.*

5 Bibliografía

[**Tanenbaum**] Tanenbaum, Andrew S. *Sistemas Operativos Modernos.* Editorial Prentice-Hall, 3ª edición (2009).

[**AfterAcademy**] The producer-consumer problem in Operating System. *After Academy*. https://afteracademy.com/blog/the-producer-consumer-problem-in-operating-system [online] última visita 28 de marzo de 2022