Sistemas Operativos II

Práctica 4 - Informe

Sincronización de procesos con paso de mensajes

Xiana Carrera Alonso

Curso 2021-2022

27/04/2022

ÍNDICE

1 Introducción

1.1 Estructura de la práctica

Esta práctica está destinada al estudio del mecanismo de pase de mensajes para la solución de carreras críticas. En particular, se implementa una resolución al problema del productor-consumidor empleando las colas de mensajes de la librería *Realtime Extensions Library*.

El programa está planteado en dos versiones:

- FIFO, en la que el consumidor retira los mensajes de su buffer de recepción como si este fuera una cola FIFO, es decir, en orden First-In-First-Out. Esta versión aprovecha la implementación predeterminada de las funciones de manejo de colas de mensajes y no utiliza sistemas de prioridades.
- LIFO, en la que el consumidor retira los mensajes de su buffer de recepción como si este fuera una pila LIFO, es decir, en orden Last-In-First-Out. Esta versión requiere una sincronización especial de los dos procesos a través del sistema de prioridades.

Este informe se estructurará en dos apartados diferenciados: uno de explicación teórica acerca del funcionamiento del método de paso de mensajes, y otro de revisión de las implementaciones, mostrando su planteamiento general, las funciones empleadas, las diferencias entre versiones y los resultados obtenidos.

2 Explicación teórica

Es habitual trabajar con procesos que se encuentran en el mismo equipo y que, por tanto, comparten el kernel y la jerarquía de memoria. En consecuencia, pueden configurar directamente zonas de memoria compartida entre ellos a través de la cual enviarse información entre sí y controlar su sincronización.

No obstante, podríamos plantearnos una situación en la que quisiéramos comunicar procesos de equipos distintos que, por ejemplo, estuvieran conectados por una red, pero sin emplear memoria compartida. En este caso ya no habría la posibilidad de utilizar directamente semáforos o mutexes. Sin embargo, existen soluciones alternativas, como el paso de mensajes.

La idea fundamental consiste en la creación de buffers para la mensajería. Cada proceso tendrá un buffer de recepción y/o uno de envío. Cuando quiera enviar un mensaje a otro, lo guardará en el buffer de envío. Este será gestionado periódicamente por el sistema, que se encargará de retirar los mensajes y pasarlos al otro proceso, almacenándolos en su respectivo buffer de recepción. Cuando el segundo proceso eventualmente acceda al buffer de recepción, retirará los mensajes y leerá su contenido.

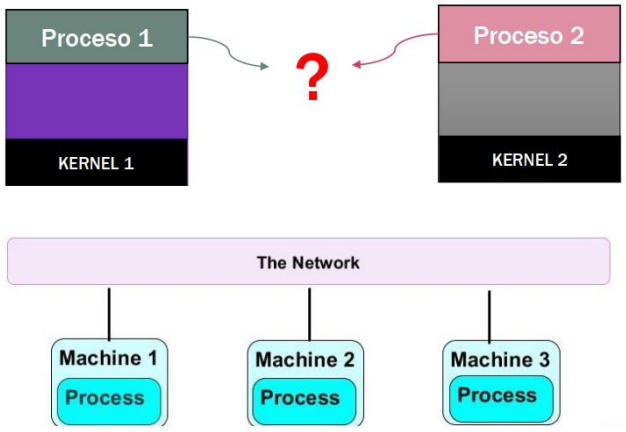


Figura 1. Situación habitual de uso de paso de mensajes con conexión por red.

Este método tiene una fuerte componente de abstracción, pues delega los aspectos referentes a la conexión a un sistema externo. Por ejemplo, en el caso de la conexión por red, los pasos de revisión de los buffers de envío en busca de mensajes pendientes, envío de estos a través de la red (con los mecanismos de seguridad, autenticación, etc. apropiados) e introducción en los buffers de recepción se gestionarían habitualmente mediante un driver accedido desde el kernel mediante llamadas a las subrutinas adecuadas.

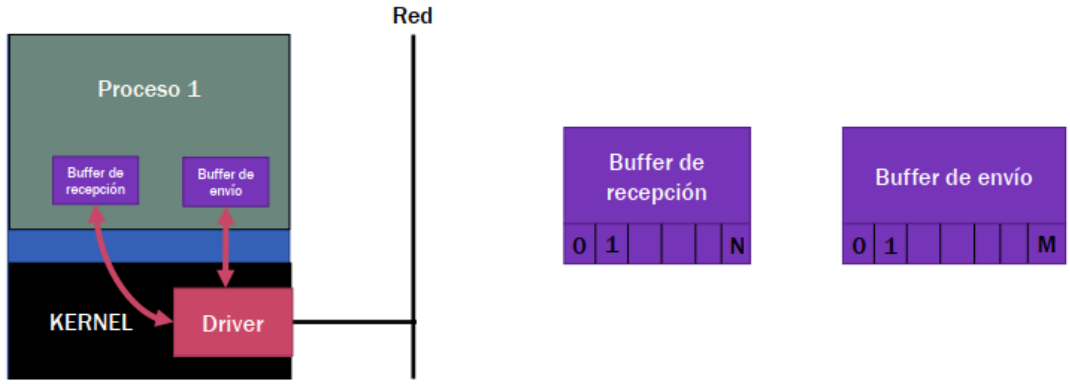


Figura 2. Estructura básica del paso de mensajes.

Nótese que el paso de mensajes también tiene utilidad en sí mismo para procesos de un mismo equipo, como es el caso de la implementación de esta práctica. Como ocurre con los semáforos o los mutexes, permite la solución de problemas de carreras críticas, si bien es cierto que reduce la velocidad de la comunicación al requerir un control especial en el envío y recepción de datos.

3 Aplicación al problema del productor-consumidor

3.1 Solución de Tanenbaum

La solución propuesta por Tanenbaum se basa en la creación de dos buffers o buzones de igual tamaño tanto para el productor como para el consumidor: uno de recepción (de solo lectura) y otro de envío (de solo escritura). Denotaremos el buzón de recepción del productor como *buz_ordenes*; y el del consumidor, como *buz_items*.

La cuestión fundamental por gestionar es el número de mensajes transmitidos por cada proceso. Esto es debido a que, si su número de envíos sobrepasase el límite de tamaño del buffer de recepción del otro, se perderían datos.

La idea de la propuesta de Tanenbaum es que el consumidor envíe un mensaje vacío al productor por cada slot que haya libre en *buz_items*. Cuando el productor recoge un mensaje recibido a través de *buz_ordenes*, interpreta que el consumidor tiene la capacidad de recibir un nuevo ítem. En consecuencia, genera uno y lo introduce en *buz_items* para transmitirlo.

Cuando el mensaje llegue al consumidor, este lo leerá, extrayéndolo de *buz_items*, y enviará un nuevo ítem al productor a través de *buz_ordenes* para avisarle de que vuelve a haber espacio disponible. A continuación, consumirá el elemento leído. El proceso se repite iterativamente.

Así, en un momento dado la suma del total de ítems presentes en *buz_ordenes* y *buz_items* más los ítems que se estén transmitiendo es igual al tamaño (número de slots) de un buffer. Como este coincide para todos los buzones, nunca habrá en circulación más mensajes de los que quepan en cualquier buffer, de forma que podrían llenarse pero no desbordarse.

Un aspecto fundamental es que la función de recepción debe ser bloqueante, de forma que si no hay ningún ítem en el buffer de recepción, el proceso quedará dormido hasta que llegue alguno (o lo despierte una señal). Si por el contrario la función retornase inmediatamente, se perdería el control sobre los turnos de envío.

3.2 Funciones y elecciones de implementación

Para el manejo de los buffers se emplean las siguientes funciones de POSIX:

- *mq_open(),* que permite:
    - Crear una cola de mensajes, indicando su nombre, las opciones, los permisos que tendrá la cola y su configuración de atributos. Como opciones debe especificarse O_CREAT y si se utilizará en modo de lectura (O_RDONLY), escritura (O_WRONLY) o ambos (O_RDWR).
    - Abrir una cola de mensajes previamente creada. En este caso solo se indica su nombre y el modo (lectura, escritura o ambos).
- *mq_unlink(),* que elimina una cola de mensajes del sistema.

- *mq_close(),* que cierra una cola para el proceso actual.
- *mq_send(),* que se utiliza para enviar mensajes a una cola. Toma como argumentos:
  - La cola de destino, una estructura *mqd_t.*
  - El mensaje (*char* *).
  - El tamaño del mensaje (*size_t*).
  - La prioridad que tendrá el mensaje (*unsigned int*).
- *mq_receive(),* que retira un mensaje de una cola. Si no hay ninguno, el proceso queda bloqueado hasta que llegue algún ítem. Si hay varios, se tomará el de mayor prioridad y, en caso de empate, el más antiguo. Toma los siguientes argumentos:
  - La cola de recepción (*mqd_t*).
  - El puntero a la variable donde se guardará el ítem leído (*char* *).
  - El tamaño del ítem (*size_t*).
  - Un puntero a un *unsigned_int* donde se guardará la prioridad del ítem.

En los atributos de los buzones se especificará su número de ranuras totales (*MAX_BUFFER*, que valdrá 5) y el tamaño de cada mensaje (*tam_msg*).

En la implementación, será el productor el que cree y elimine las colas. El consumidor solamente las abrirá. Por consiguiente, se debe lanzar el ejecutable del productor antes que el del consumidor.

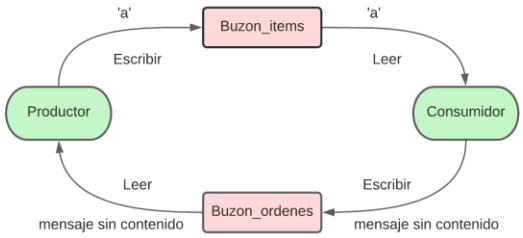


Figura 3. Intercambio de ítems.

Los datos enviados por el productor serán caracteres y, por la especificación de la función *producir_elemento(),* serán en concreto las letras ‘a’, ‘b’, ‘c’, ‘d’ y ‘e’, que se tomarán de forma cíclica según la iteración actual. El consumidor enviará los mismos mensajes de vuelta, sin realmente vaciarlos, ya que su contenido no es usado por el productor.

El número de iteraciones de cada programa será *DATOS_A_PRODUCIR* para el productor y *DATOS_A_CONSUMIR* para el consumidor, aunque estos valores coincidirán entre sí.

El consumidor comenzará su ejecución llenando el buffer de recepción del productor para indicarse que *buz_items* está completamente vacío. El productor no podrá comenzar a actuar hasta haber recibido al menos un envío.

Como puntualización, nótese que el paso de mensajes definido por POSIX se basa en la creación de los buzones sobre regiones de memoria compartidas. Por consiguiente, no hay una transmisión “real” de datos y solamente hay 2 buzones reales, en lugar de los 4 que se definen de forma teórica (**[Auckland]**).

3.3 Versión FIFO y LIFO

El orden en el que el consumidor lea los mensajes se configurará a través de las prioridades que marque el productor.

Dado que a igualdad de prioridad el orden de lectura predeterminado es FIFO, en la primera versión basta que el productor marque la misma prioridad para todos los mensajes.

Para la segunda versión, el productor tendrá que seguir una secuencia de prioridades creciente entre envíos. En particular, se ha optado por tomar una prioridad igual al contador de iteraciones efectuadas. De esta forma, de entre los mensajes disponibles en *buz_items*, el consumidor tomará siempre el que se haya enviado en una iteración posterior, aunque hayan llegado desordenados.

En ambas versiones, el consumidor envía sus mensajes con prioridad 0, pues al productor solamente le interesa el hecho de que lleguen mensajes, y no su contenido.

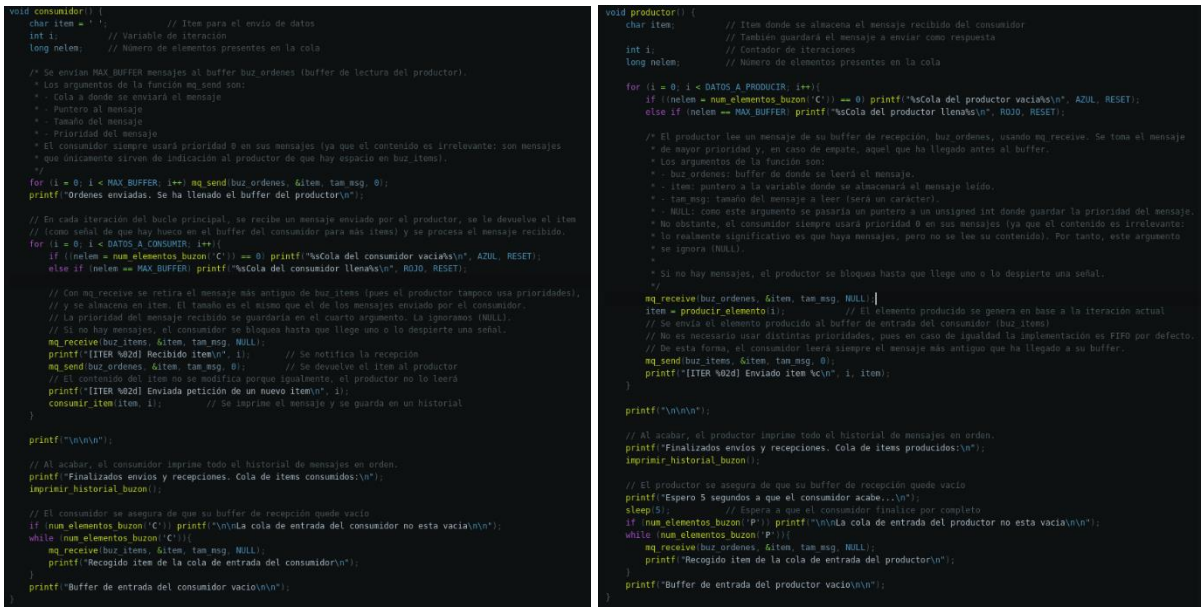


Figura 4. Programas del consumidor y del productor en su versión FIFO.

```c
mq_receive(buz_items, &item, tam_msg, &prio);
printf("[ITER %02d] Recibido item\n", i);
mq_send(buz_ordenes, &item, tam_msg, 0);   // Se devuelve el item al productor
// El contenido del item no se modifica porque igualmente, el productor no lo leerá
printf("[ITER %02d] Enviada petición de un nuevo item\n", i);
consumir_item(item, i, prio);              // Se imprime el mensaje y se guarda en un historial
```

```c
mq_receive(buz_ordenes, &item, tam_msg, 0);
item = producir_elemento(i);        // El elemento producido se genera en base a la iteración actual
/* El mensaje es enviado al buzón de entrada del consumidor (buz_items) con prioridad igual a la iteración
 * actual. Esto asegura que el consumidor siempre leerá el elemento de la iteración más reciente que haya
 * presente en el buffer, de forma que funciona como una pila LIFO.
 */
mq_send(buz_items, &item, tam_msg, i);
printf("[ITER %02d] Enviado item %c\n", i, item);
```

Figura 5. Adaptación del bucle principal a la versión LIFO.

3.4 Sleeps

Para forzar el vaciado y llenado de los buzones, se realiza una llamada a sleep en cada proceso. El número de elementos de los buffers se comprueba al iniciar cada nueva iteración y en las funciones *producir_item()* y *consumir_item().*

La duración de la espera es aleatoria para ambos procesos, pudiendo ser de 0, 1 o 2 segundos, y se efectúa dentro de las funciones *producir_item()* y *consumir_item().*

Se observa que aún así, no tienen lugar carreras críticas, pues sigue imponiéndose el control de turnos y los datos se transmiten por estructuras separadas.

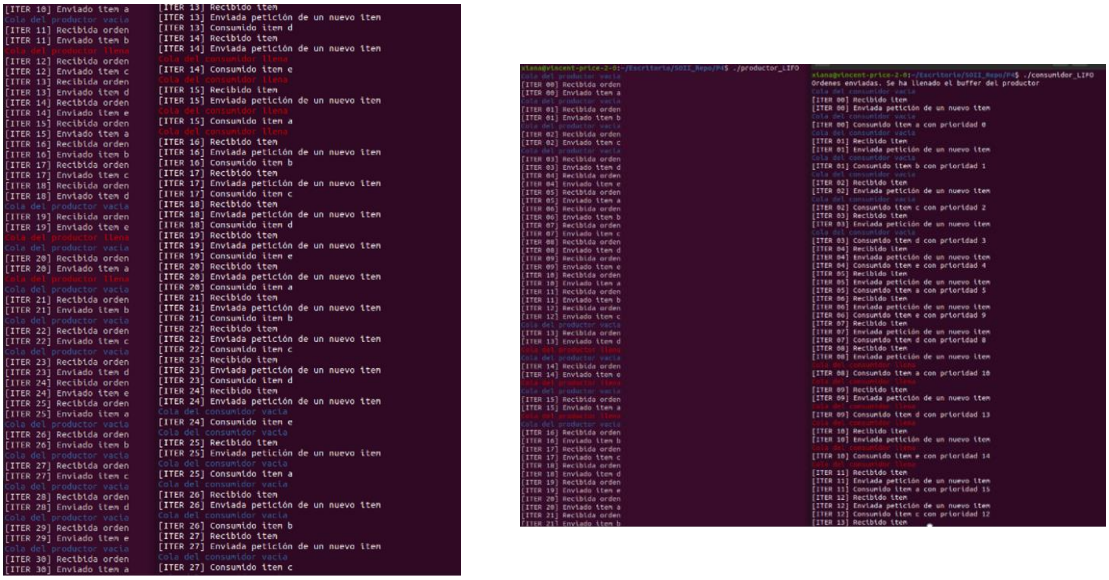
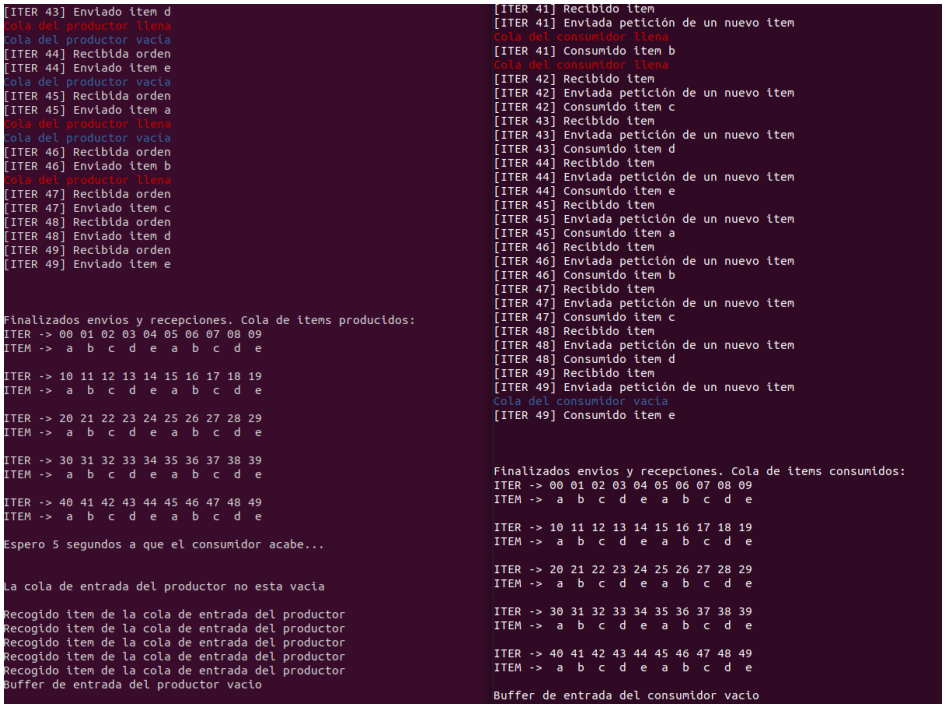


Figura 6. Vaciado y llenado de las colas en la versión FIFO y LIFO, respectivamente. En cada imagen, el proceso de la izquierda es el productor y el de la derecha, el consumidor.

3.5 Resultados y solución de carreras críticas

Los resultados son los esperados: en la versión FIFO, el productor lee los mensajes en exactamente el mismo orden en el que el productor los fue generando y enviando. Al asegurarnos de que no se sobrepasan los límites de los buffers de recepción, no hay carreras críticas.

En la versión LIFO, los mensajes aparecen desordenados ya que el consumidor va leyendo datos en paralelo al trabajo del productor. Cuando el primero se retrasa, se aprecian secuencias de prioridades crecientes, pues van llegando datos cada vez más recientes. Cuando se adelanta, pueden verse períodos en los que empieza a leer ítems que habían quedado atrás y las prioridades decrecen. No obstante, las secuencias son solo momentáneas.



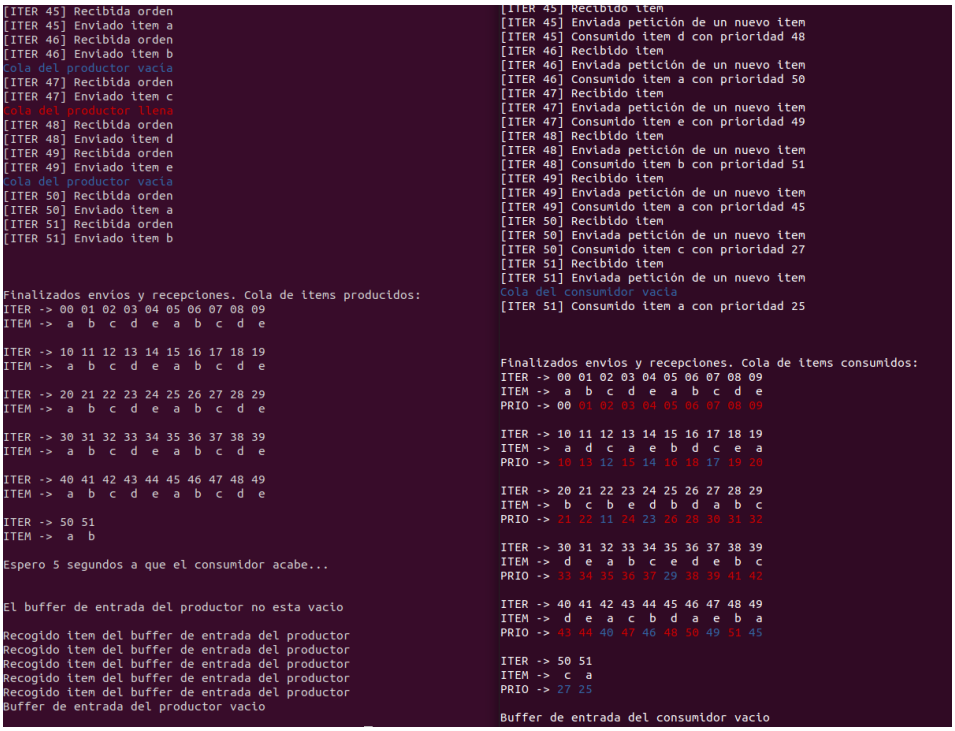


Figura 7. Resultados de las ejecuciones para las versiones FIFO y LIFO. En la LIFO se imprime por colores la evolución de las secuencias de prioridades (rojo para secuencias crecientes; azul para decrecientes). En ningún caso hay errores en el orden de los datos y no se observan carreras críticas.

3.6 Vaciado total de las colas

Por construcción del programa, al haber un número de iteraciones fijo e igual entre ambos procesos, *buz_items* acaba estando vacía al finalizar el consumidor su bucle, pues debe recuperar todos los elementos enviados por el productor para alcanzar *DATOS_A_CONSUMIR* iteraciones.

En contraposición, *buz_ordenes* queda lleno, pues al leer los últimos 5 mensajes el consumidor transmitirá sendos ítems avisando de los nuevos huecos. No obstante, el productor no llegará a verlos, pues serían iteraciones extra a *DATOS_A_PRODUCIR*. En consecuencia, tendrá que retirarlos "manualmente". Nótese que además se realiza un sleep previo a esta comprobación, ya que el consumidor puede ir algo retrasado por el efecto de los *sleep()* aleatorios y llegar a mandar sus últimos mensajes cuando el productor ya haya finalizado.

4 Bibliografía

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

[**Auckland**] Digital Unix Documentation Library. *University of Auckland*. https://www.cs.auckland.ac.nz/references/unix/digital/APS33DTE/DOCU_011.HTM [online] última visita 26 de abril de 2022