

# Sistemas Operativos II

## **Práctica 4 - Informe**

*Sincronización de procesos con paso de mensajes*

Xiana Carrera Alonso

Curso 2021-2022

<b>SOII</b>	Sincronización de procesos con paso de mensajes	Práctica 4
	Xiana Carrera Alonso	

# ÍNDICE

<b>1. INTRODUCCIÓN .....</b>	<b>1</b>
<b>1.1 ESTRUCTURA DE LA PRÁCTICA .....</b>	<b>1</b>
<b>2. EXPLICACIÓN TEÓRICA.....</b>	<b>1</b>
<b>3. APLICACIÓN AL PROBLEMA DEL PRODUCTOR-CONSUMIDOR..3</b>	
<b>3.1 SOLUCIÓN DE TANENBAUM.....</b>	<b>3</b>
<b>3.2 FUNCIONES Y ELECCIONES DE IMPLEMENTACIÓN .....</b>	<b>3</b>
<b>3.3 VERSIÓN FIFO Y LIFO.....</b>	<b>5</b>
<b>3.4 SLEEPS .....</b>	<b>6</b>
<b>3.5 RESULTADOS Y SOLUCIÓN DE CARRERAS CRÍTICAS ..</b>	<b>6</b>
<b>3.6 VACIADO TOTAL DE LAS COLAS.....</b>	<b>8</b>
<b>4. BIBLIOGRAFÍA .....</b>	<b>8</b>

<b>SOII</b>	<b>Sincronización de procesos con paso de mensajes</b>	<b>Práctica 4</b>
	<b>Xiana Carrera Alonso</b>	

## 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.

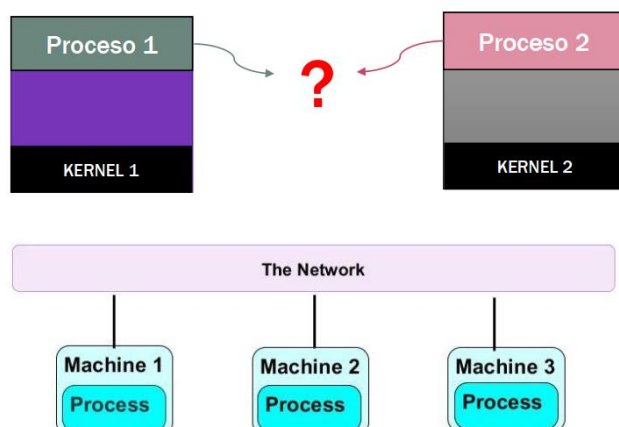
Este informe se estructurará en dos apartados diferenciados: uno de explicación teórica acerca del funcionamiento del método de paso de mensajes, y otro de revisión de las implementaciones, mostrando su planteamiento general, las funciones empleadas, las diferencias entre versiones y los resultados obtenidos.

## 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.

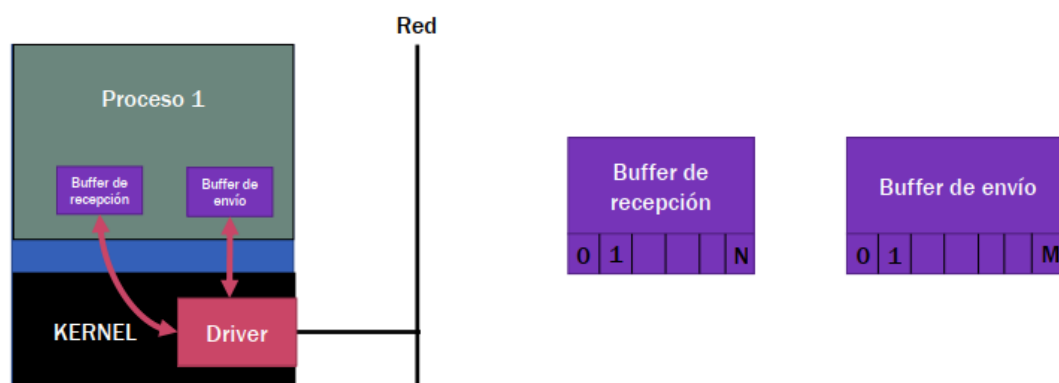
No obstante, podríamos plantearnos una situación en la que quisiéramos comunicar procesos de equipos distintos que, por ejemplo, estuvieran conectados por una red, pero sin emplear memoria compartida. En este caso ya no habría la posibilidad de utilizar directamente semáforos o mutexes. Sin embargo,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

<b>SOII</b>	Sincronización de procesos con paso de mensajes	Práctica 4
	Xiana Carrera Alonso	

### 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:

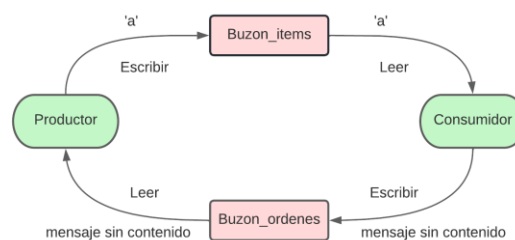
- *mq\_open()*, que permite:
  - Crear una cola de mensajes, indicando su nombre, las opciones, los permisos que tendrá la cola y su configuración de atributos. Como opciones debe especificarse *O\_CREAT* y si se utilizará en modo de lectura (*O\_RDONLY*), escritura (*O\_WRONLY*) o ambos (*O\_RDWR*).
  - 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.

<b>SOII</b>	<b>Sincronización de procesos con paso de mensajes</b>	<b>Práctica 4</b>
	<b>Xiana Carrera Alonso</b>	

- *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.

```

void consumidor() {
    char item = ' '; // Item para el envío de datos
    int i; // Variable de iteración
    long nlelem; // Número de elementos presentes en la cola

    /* Se envían MAX_BUFFER mensajes al buffer buz_ordenes (buffer de lectura del productor).
    * Los argumentos de la función mq_send son:
    * - Cola a donde se enviará el mensaje
    * - Puntero al mensaje
    * - Tamaño del mensaje
    * - Prioridad del mensaje
    * El consumidor siempre usará prioridad 0 en sus mensajes (ya que el contenido es irrelevante; son mensajes
    * que únicamente sirven de indicación al productor de que hay espacio en buz_items).
    */
    for (i = 0; i < MAX_BUFFER; i++) mq_send(buz_ordenes, &item, tam_msg, 0);
    printf("Ordenes enviadas. Se ha llenado el buffer del productor\n");

    // En cada iteración del bucle principal, se recibe un mensaje enviado por el productor, se le devuelve el item
    // (como señal de que hay hueco en el buffer del consumidor para más items) y se procesa el mensaje recibido.
    for (i = 0; i < DATOS_A_CONSUMIR; i++) {
        if ((nlelem = num_elementos_buzon('C')) == 0) printf("La cola del consumidor vacía\n", AZUL, RESET);
        else if (nlelem == MAX_BUFFER) printf("La cola del consumidor llena\n", ROJO, RESET);

        /* Con mq_receive se retira el mensaje más antiguo de buz_items (pues el productor tampoco usa prioridades),
        // y se almacena en item. El tamaño es el mismo que el de los mensajes enviados por el consumidor.
        // La prioridad del mensaje recibido se guardaría en el cuarto argumento, la ignoramos (NULL).
        // Si no hay mensajes, el consumidor se bloquea hasta que llegue uno o lo despierte una señal.
        mq_receive(buz_items, &item, tam_msg, NULL);
        printf("ITER %02d Recibido item\n", i); // Se notifica la recepción
        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); // Se imprime el mensaje y se guarda en un historial
    }

    printf("\n\n");

    // Al acabar, el consumidor imprime todo el historial de mensajes en orden.
    printf("Finalizados envíos y recepciones. Cola de items consumidos:\n");
    imprimir_historial_buzon();

    // El consumidor se asegura de que su buffer de recepción quede vacío
    if (num_elementos_buzon('C')) printf("La cola de entrada del consumidor no está vacía\n");
    while (num_elementos_buzon('C')) {
        mq_receive(buz_items, &item, tam_msg, NULL);
        printf("Recogido item de la cola de entrada del consumidor\n");
    }
    printf("Buffer de entrada del consumidor vacío\n");
}

void productor() {
    char item; // Item donde se almacena el mensaje recibido del consumidor
    // También guardará el mensaje a enviar como respuesta
    int i; // Contador de iteraciones
    long nlelem; // Número de elementos presentes en la cola

    for (i = 0; i < DATOS_A_PRODUCIR; i++) {
        if ((nlelem = num_elementos_buzon('C')) == 0) printf("La cola del productor vacía\n", AZUL, RESET);
        else if (nlelem == MAX_BUFFER) printf("La cola del productor llena\n", ROJO, RESET);

        /* El productor lee un mensaje de su buffer de recepción, buz_ordenes, usando mq_receive. Se toma el mensaje
        * de mayor prioridad y, en caso de empate, aquel que ha llegado antes al buffer.
        * Los argumentos de la función son:
        * - buz_ordenes: buffer de donde se leerá el mensaje.
        * - item: puntero a la variable donde se almacenará el mensaje leído.
        * - tam_msg: tamaño del mensaje a leer (será un carácter).
        * - NULL: como este argumento se pasaría un puntero a un unsigned int donde guardar la prioridad del mensaje.
        * No obstante, el consumidor siempre usará prioridad 0 en sus mensajes (ya que el contenido es irrelevante).
        * Su realmente significativo es que haya mensajes, pero no se lee su contenido. Por tanto, este argumento
        * se ignora (NULL).
        * Si no hay mensajes, el productor se bloquea hasta que llegue uno o lo despierte una señal.
        */
        mq_receive(buz_ordenes, &item, tam_msg, NULL);
        item = producir_elemento(i); // El elemento producido se genera en base a la iteración actual
        // Se envía el elemento producido al buffer de entrada del consumidor (buz_items)
        // No es necesario usar distintas prioridades, pues en caso de igualdad la implementación es FIFO por defecto
        // De esta forma, el consumidor leerá siempre el mensaje más antiguo que ha llegado a su buffer.
        mq_send(buz_items, &item, tam_msg, 0);
        printf("ITER %02d Enviado item %c\n", i, item);

        printf("\n\n");

        // Al acabar, el productor imprime todo el historial de mensajes en orden.
        printf("Finalizados envíos y recepciones. Cola de items producidos:\n");
        imprimir_historial_buzon();

        // El productor se asegura de que su buffer de recepción quede vacío
        printf("Espero 5 segundos a que el consumidor acabe...\n");
        sleep(5);
        if (num_elementos_buzon('P')) printf("La cola de entrada del productor no está vacía\n");
        while (num_elementos_buzon('P')) {
            mq_receive(buz_ordenes, &item, tam_msg, NULL);
            printf("Recogido item de la cola de entrada del productor\n");
        }
        printf("Buffer de entrada del productor vacío\n");
    }
}

```

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

```

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

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.

[illegible]

**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.



```

[ITER 43] Enviado ítem d
Cola del productor llena
Cola del productor vacía
[ITER 44] Recibida orden
[ITER 44] Enviado ítem e
Cola del productor vacía
[ITER 45] Recibida orden
[ITER 45] Enviado ítem a
Cola del productor llena
Cola del productor vacía
[ITER 46] Recibida orden
[ITER 46] Enviado ítem b
Cola del productor vacía
[ITER 47] Recibida orden
[ITER 47] Enviado ítem c
[ITER 48] Recibida orden
[ITER 48] Enviado ítem d
[ITER 49] Recibida orden
[ITER 49] Enviado ítem e

Finalizados envíos y recepciones. Cola de ítems producidos:
ITER -> 00 01 02 03 04 05 06 07 08 09
ITEM -> a b c d e a b c d e

ITER -> 10 11 12 13 14 15 16 17 18 19
ITEM -> a b c d e a b c d e

ITER -> 20 21 22 23 24 25 26 27 28 29
ITEM -> a b c d e a b c d e

ITER -> 30 31 32 33 34 35 36 37 38 39
ITEM -> a b c d e a b c d e

ITER -> 40 41 42 43 44 45 46 47 48 49
ITEM -> a b c d e a b c d e

Espero 5 segundos a que el consumidor acabe...

La cola de entrada del productor no esta vacía

Recogido ítem de la cola de entrada del productor
Recogido ítem de la cola de entrada del productor
Recogido ítem de la cola de entrada del productor
Recogido ítem de la cola de entrada del productor
Buffer de entrada del productor vacío

[ITER 41] Recibido ítem
[ITER 41] Enviada petición de un nuevo ítem
Cola del consumidor vacía
[ITER 41] Consumido ítem b
Cola del consumidor vacía
[ITER 42] Recibido ítem
[ITER 42] Enviada petición de un nuevo ítem
[ITER 42] Consumido ítem c
[ITER 43] Recibido ítem
[ITER 43] Enviada petición de un nuevo ítem
[ITER 43] Consumido ítem d
[ITER 44] Recibido ítem
[ITER 44] Enviada petición de un nuevo ítem
[ITER 44] Consumido ítem e
[ITER 45] Recibido ítem
[ITER 45] Enviada petición de un nuevo ítem
[ITER 45] Consumido ítem a
[ITER 46] Recibido ítem
[ITER 46] Enviada petición de un nuevo ítem
[ITER 46] Consumido ítem b
[ITER 47] Recibido ítem
[ITER 47] Enviada petición de un nuevo ítem
[ITER 47] Consumido ítem c
[ITER 48] Recibido ítem
[ITER 48] Enviada petición de un nuevo ítem
[ITER 48] Consumido ítem d
[ITER 49] Recibido ítem
[ITER 49] Enviada petición de un nuevo ítem
Cola del consumidor vacía
[ITER 49] Consumido ítem e

Finalizados envíos y recepciones. Cola de ítems consumidos:
ITER -> 00 01 02 03 04 05 06 07 08 09
ITEM -> a b c d e a b c d e

ITER -> 10 11 12 13 14 15 16 17 18 19
ITEM -> a b c d e a b c d e

ITER -> 20 21 22 23 24 25 26 27 28 29
ITEM -> a b c d e a b c d e

ITER -> 30 31 32 33 34 35 36 37 38 39
ITEM -> a b c d e a b c d e

ITER -> 40 41 42 43 44 45 46 47 48 49
ITEM -> a b c d e a b c d e

Buffer de entrada del consumidor vacío

[ITER 45] Recibida orden
[ITER 45] Enviado ítem a
[ITER 46] Recibida orden
[ITER 46] Enviado ítem b
Cola del productor vacía
[ITER 47] Recibida orden
[ITER 47] Enviado ítem c
Cola del productor vacía
[ITER 48] Recibida orden
[ITER 48] Enviado ítem d
[ITER 49] Recibida orden
[ITER 49] Enviado ítem e
Cola del productor vacía
[ITER 50] Recibida orden
[ITER 50] Enviado ítem a
[ITER 51] Recibida orden
[ITER 51] Enviado ítem b

Finalizados envíos y recepciones. Cola de ítems producidos:
ITER -> 00 01 02 03 04 05 06 07 08 09
ITEM -> a b c d e a b c d e

ITER -> 10 11 12 13 14 15 16 17 18 19
ITEM -> a b c d e a b c d e

ITER -> 20 21 22 23 24 25 26 27 28 29
ITEM -> a b c d e a b c d e

ITER -> 30 31 32 33 34 35 36 37 38 39
ITEM -> a b c d e a b c d e

ITER -> 40 41 42 43 44 45 46 47 48 49
ITEM -> a b c d e a b c d e

ITER -> 50 51
ITEM -> a b

Espero 5 segundos a que el consumidor acabe...

El buffer de entrada del productor no esta vacío

Recogido ítem del buffer de entrada del productor
Recogido ítem del buffer de entrada del productor
Recogido ítem del buffer de entrada del productor
Recogido ítem del buffer de entrada del productor
Buffer de entrada del productor vacío

[ITER 45] Recibido ítem
[ITER 45] Enviada petición de un nuevo ítem
[ITER 45] Consumido ítem d con prioridad 48
[ITER 46] Recibido ítem
[ITER 46] Enviada petición de un nuevo ítem
[ITER 46] Consumido ítem a con prioridad 50
[ITER 47] Recibido ítem
[ITER 47] Enviada petición de un nuevo ítem
[ITER 47] Consumido ítem e con prioridad 49
[ITER 48] Recibido ítem
[ITER 48] Enviada petición de un nuevo ítem
[ITER 48] Consumido ítem b con prioridad 51
[ITER 49] Recibido ítem
[ITER 49] Enviada petición de un nuevo ítem
[ITER 49] Consumido ítem a con prioridad 45
[ITER 50] Recibido ítem
[ITER 50] Enviada petición de un nuevo ítem
[ITER 50] Consumido ítem c con prioridad 27
[ITER 51] Recibido ítem
[ITER 51] Enviada petición de un nuevo ítem
Cola del consumidor vacía
[ITER 51] Consumido ítem a con prioridad 25

Finalizados envíos y recepciones. Cola de ítems consumidos:
ITER -> 00 01 02 03 04 05 06 07 08 09
ITEM -> a b c d e a b c d e
PRIO -> 00 01 02 03 04 05 06 07 08 09

ITER -> 10 11 12 13 14 15 16 17 18 19
ITEM -> a d c a e b d c e a
PRIO -> 10 13 12 15 14 16 18 17 19 20

ITER -> 20 21 22 23 24 25 26 27 28 29
ITEM -> b c b e d b d a b c
PRIO -> 21 22 11 24 23 20 28 30 31 32

ITER -> 30 31 32 33 34 35 36 37 38 39
ITEM -> d e a b c e d e b c
PRIO -> 33 34 35 36 37 29 38 39 41 42

ITER -> 40 41 42 43 44 45 46 47 48 49
ITEM -> d e a c b d a e b a
PRIO -> 43 44 40 47 46 48 50 49 51 45

ITER -> 50 51
ITEM -> c a
PRIO -> 27 25

Buffer de entrada del consumidor vacío

```

**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.

<b>SOII</b>	Sincronización de procesos con paso de mensajes	Práctica 4
	Xiana Carrera Alonso	

### 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](https://www.cs.auckland.ac.nz/references/unix/digital/APS33DTE/DOCU_011.HTM)  
 [online] última visita 26 de abril de 2022