Área de Arquitectura y Tecnología de Computadores

Universidad Carlos III de Madrid

SISTEMAS OPERATIVOS

Práctica 3. Programación multi-hilo

Grado de Ingeniería en Informática
Grado en Matemática Aplicada y Computación
Doble Grado en Ingeniería Informática y Administración de Empresas

Curso 2023/2024

Índice

1 Enunciado de la Práctica

Esta práctica permite al alumno familiarizarse con los servicios para la gestión de procesos que proporciona POSIX.

Para la gestión de procesos ligeros (hilos), se utilizarán las llamadas al sistema `pthread_create`, `pthread_join`, `pthread_exit`, y para la sincronización de éstos, **mutex** y **variables condicionales**:

- **pthread_create**: crea un nuevo hilo que ejecuta una función que se le indica como argumento en la llamada.
- **pthread_join**: realiza una espera por un hilo que debe terminar y que está indicado como argumento de la llamada.
- **pthread_exit**: finaliza la ejecución del proceso que realiza la llamada.

El alumno deberá diseñar y codificar, en lenguaje C y sobre el sistema operativo UNIX / Linux, un programa que actúe como gestor de beneficios y de *stock* de una tienda.

1.1 Descripción de la Práctica

El objetivo de esta práctica es codificar un sistema multi-hilo concurrente que calcule el beneficio y el *stock* de una tienda. Dado un fichero con un formato específico, se debe calcular el beneficio (diferencia entre el coste de adquirir los productos y los ingresos por su venta) así como el *stock* restante de cada producto.

Para la realización de la funcionalidad, se recomienda implementar dos funciones básicas, que representan los roles del programa (siguiendo el comportamiento de la **Figura 1**):

- **Productor**: Será la función que ejecuten los hilos encargados de agregar elementos en la cola circular compartida.
- **Consumidor**: Será la función que ejecuten los hilos encargados de extraer elementos de la cola circular compartida.

1. El **hilo principal** será el encargado de:
   (a) Leer los argumentos de entrada
   (b) Cargar los datos del fichero proporcionado en memoria
   (c) Hacer un reparto equitativo de la carga del fichero entre el número de hilos productores indicado
   (d) Lanzar los **N** productores y los **M** consumidores
   (e) Esperar la finalización de todos los hilos y mostrar el beneficio y el *stock* de cada producto al finalizar al finalizar las operaciones.

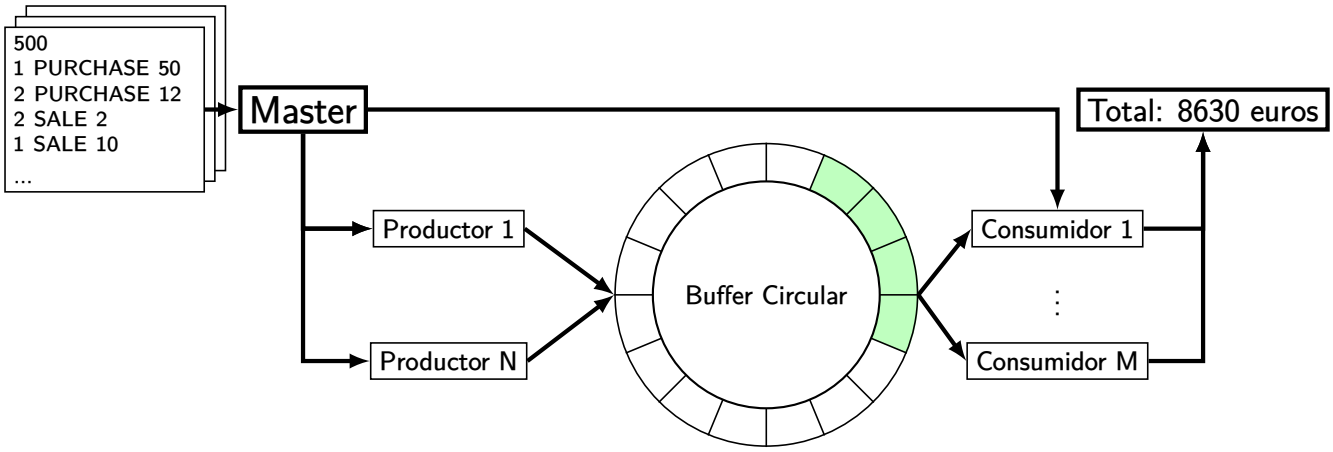

Fig. 1: Ejemplo de funcionamiento con N productores, M consumidores y un buffer.

2. Cada **hilo productor** deberá:
   (a) Obtener los datos extraídos del fichero que le correspondan e insertarlos, **uno a uno** en la cola circular.
   (b) **Esta tarea debe realizarse de forma concurrente con el resto de los productores, así como los consumidores**. En ningún caso se pueden quedar los hilos bloqueados **manualmente** o forzar un orden entre ellos (por ejemplo, esperar que un hilo inserte todos sus elementos, o los que quepan en la cola, y dar paso a los consumidores para que los extraiga; luego dar paso al siguiente productor, etc.).
3. Cada **hilo consumidor** deberá:
   (a) Obtener los elementos insertados en la cola.
   (b) Cada elemento extraído representa una operación con un producto. Estas operaciones consisten en la adquisición de un número determinado de productos y la venta de un número determinado de productos. Por lo tanto, se deberá calcular los beneficios de realizar dichas operaciones, así como el *stock* de cada producto una vez realizadas las operaciones.
   (c) Una vez procesados todos los elementos, cada hilo finalizará su ejecución y **retornara al proceso principal el beneficio y *stock* parcial calculado**.

1.1.1 Gestor de cálculo de costes y *stock* de una tienda

El programa principal se encargará de importar los argumentos y datos del fichero indicado. Para ello, se debe tener en cuenta que la ejecución del programa será de la siguiente manera:

```
./store_manager <file_name><num_producers><num_consumers><buff_size>
```

La etiqueta **file_name** se corresponde con el nombre del fichero que se quiere importar. La etiqueta **num_producers** es un entero que representa el número de hilos productores que se quieren generar. La etiqueta **num_consumers** es un entero que representa el número de hilos consumidores que se quieren generar. Finalmente, la etiqueta **buff_size** es un entero que indica

el tamaño de la cola circular (número máximo de elementos que puede almacenar).

Por otro lado, el fichero de entrada debe tener el siguiente formato:

```
500 #Num max operaciones a procesar
1 PURCHASE 50
2 PURCHASE 12
2 SALE 2
1 SALE 10
...
```

La primera línea del fichero representa el número de operaciones que se quieren realizar. Puede haber más operaciones en el fichero, pero **únicamente se deben procesar las indicadas por este valor. En ningún caso puede haber menos operaciones en el fichero que operaciones indica el primer valor**. El resto de las líneas del fichero representan una operación:

<id_producto><tipo_operacion><unidades>

Son tres valores separados por un espacio y finalizado en un salto de línea. El **id_producto** es el identificador del producto y tiene asociado un coste de compra y un precio de venta por unidad:

id_producto	**Coste de compra** (€/unidad)	**Precio de venta** (€/unidad)
1	2	3
2	5	10
3	15	20
4	25	40
5	100	125

Table 1: Coste de compra y precio de venta de cada producto.

El segundo valor, indica el tipo de operación que se va a realizar sobre el producto. Puede ser la adquisición de un producto (PURCHASE) o la venta de un producto (SALE). Por último, el ultimo valor indica el número de unidades que se han adquirido o se han vendido de un determinado producto.

El proceso principal debe cargar en memoria la información contenida en el fichero para su posterior procesamiento por los productores. Para ello se recomienda hacer uso de la función `scanf` y la reserva dinámica de memoria con `malloc` (y `free` para su posterior liberación). La idea es:

1. Obtener el número de operaciones (primer valor del fichero).
2. Reservar memoria para todas esas operaciones con `malloc`.
3. Almacenar las operaciones en el array.
4. Repartir las operaciones de **forma equitativa** entre los productores.

(a) Para simplificar la tarea, se recomienda hacer un reparto de las operaciones, de forma que cada hilo productor sepa en qué posición empezar a procesar y en qué posición terminar.

(b) De esta forma cada thread es consciente de cuándo debe finalizar su ejecución.

(c) Para ello, se pueden pasar argumentos a cada thread productor en el momento del lanzamiento con **pthread_create**.

5. Tras el procesamiento de los threads, liberar la memoria reservada con `free`.

NOTA

Para almacenar los datos desde el fichero se puede generar un array de estructuras. También se recomienda utilizar una estructura para el paso de parámetros a los threads.

A continuación se muestra un ejemplo de la salida del programa:

```
$> ./store_manager input_file 5 2 10
Total: 234234 euros
Stock:
    Product 1: 10 units
    Product 2: 2 units
    Product 3: 50 units
    Product 4: 22 units
    Product 5: 33 units
$>
```

1.1.2 N-productores / M-consumidores

El problema que se pide implementar es un ejemplo clásico de sincronización de procesos: al compartirse una cola compartida (**buffer circular**), hay que realizar un control sobre la concurrencia a la hora de depositar objetos en la misma, y a la hora de extraerlos.

Para la implementación de los hilos productores se recomienda que la función siga el siguiente esquema por simplicidad:

1. Obtención de los índices a los que debe acceder de los datos del fichero. Se recomienda hacer paso de parámetros al thread.
2. Bucle desde el inicio hasta el fin de las operaciones que debe procesar:
   (a) Obtener los datos de la operación.
   (b) Crear un elemento con los datos de la operación para insertar en la cola.
   (c) Insertar elemento en la cola.
3. Finalizar el hilo con **pthread_exit**.

Para la implementación de los hilos consumidores se recomienda seguir un esquema similar al anterior por simplicidad:

1. Extraer elemento de la cola.
2. Calcular el beneficio y el *stock* y acumularlo.
3. Cuando se hayan procesado todas las operaciones, finalizar el hilo con **pthread_exit devolviendo el beneficio y *stock* parcial calculado por cada uno**.

NOTA

Para el control de la concurrencia hay que utilizar **mutex** y **variables condición**. La concurrencia se puede gestionar en las funciones de productor y consumidor, o en el código de la cola circular (en **queue.c**). La elección es del grupo de prácticas.

Cola sobre un buffer circular La comunicación entre los productores y los consumidores se realizará mediante una cola circular. Debe crearse una cola circular compartida por los productores y el consumidor. Dado que constantemente se van a producir modificaciones sobre este elemento, se deben implementar mecanismos de control de la concurrencia para los procesos ligeros.

La cola circular y sus funciones deben estar implementadas en un fichero denominado **queue.c**, y debe contener, al menos, las siguientes funciones:

- **queue* queue_init (int num_elements)**: función que crea la cola y reserva el tamaño especificado como parámetro.
- **int queue_destroy (queue* q)**: función que elimina la cola y libera todos los recursos asignados.
- **int queue_put (queue* q , struct element * ele)**: función que inserta elementos en la cola si hay espacio disponible. Si no hay espacio disponible, debe esperar hasta que pueda ser realizada la inserción.
- **struct element * queue_get (queue* q)**: función que extrae elementos de la cola si no está vacía. Si está vacía, se debe esperar hasta que haya un elemento disponible.
- **int queue_empty (queue* q)**: función que consulta el estado de la cola y determina si está vacía (return 1) o no (return 0).
- **int queue_full (queue* q)**: función que consulta el estado de la cola y determina si está llena (return 1) o aún dispone de posiciones disponibles (return 0).

La implementación de esta cola debe realizarse de forma que no haya problemas de concurrencia entre los threads que están trabajando con ella. Para ello se deben utilizar los mecanismos propuestos de **mutex** y **variables condición**.

El objeto que debe almacenarse y extraerse de la cola circular debe corresponderse con una estructura definida con los siguientes campos, al menos:

- **int product**: representa el identificador del producto.
- **int op**: representa el tipo de operación.
- **int units**: representa el número de unidades implicadas en la operación.

1.2 Código Fuente de Apoyo

Para facilitar la realización de esta práctica se dispone del fichero:

`ssoo_p3_multihilo_2024.zip`

Que contiene código fuente de apoyo. Para extraer su contenido ejecutar lo siguiente:

```
unzip ssoo_p3_multihilo_2024.zip
```

Al extraer su contenido, se crea el directorio ssoo_p3_multihilo_2024/, donde se debe desarrollar la práctica. Dentro de este directorio se habrán incluido los siguientes ficheros:

- **Makefile**
  NO debe ser modificado. Fichero fuente para la herramienta `make`. Con él se consigue la recompilación automática sólo de los ficheros fuente que se modifiquen. Utilice *make* para compilar los programas, y *make clean* para eliminar los archivos compilados.
- **store_manager.c**
  Debe modificarse. Fichero fuente C donde los alumnos deberán codificar el gestor de beneficios y *stock* de una tienda.
- **queue.h**
  Debe modificarse. Fichero de cabeceras donde los alumnos deberán definir las estructuras de datos y funciones utilizadas para la gestión de la cola circular
- **queue.c**
  Debe modificarse. Fichero fuente C donde los alumnos deberán implementar las funciones que permiten gestionar la cola circular.
- **probador_ssoo_p3.sh**
  NO debe ser modificado. Shell-script que realiza una auto-corrección guiada de la práctica. En ella se muestra, una por una, las instrucciones de prueba que se quieren realizar, así como el resultado esperado y el resultado obtenido por el programa del alumno. Al final se da una nota tentativa del código de la práctica (sin contar revisión manual y la memoria). Para ejecutarlo se debe dar permisos de ejecución al archivo mediante:

  ```
  chmod +x probador_ssoo_p3.sh
  ```

  Y ejecutar con:

  ```
  ./probador_ssoo_p3.sh <fichero_zip_código>
  ```

- **autores.txt**
  Debe modificarse. Fichero `txt` donde incluir los autores de la práctica.
- **file.txt**
  Fichero de apoyo.

1.3 Comprobador de resultados

Junto al código de apoyo se adjunta un fichero **checker.xlsx** que es un documento excel para comprobar si los resultados obtenidos son correctos o no. El fichero **file.txt** proporcionado, contiene los mismos datos que el primero, pero sin fórmulas y comprobaciones (tiene el formato especificado para poder procesarlo). Si se modifica el número de operaciones que se quieren procesar en *file.txt*, se puede comprobar el resultado obtenido modificando el número de operaciones en el documento *checker.xlsx*.

NOTA

También se pueden generar nuevos ficheros (**recomendable**), y sobre ellos realizar nuevas pruebas.

2 Entrega

2.1 Plazo de Entrega

La fecha límite de entrega de la práctica en AULA GLOBAL será el **10 de mayo de 2024 (hasta las 23:55h)**

2.2 Procedimiento de entrega de las prácticas

La entrega de las prácticas ha de realizarse de forma electrónica y por **un único integrante del grupo**. En AULA GLOBAL se habilitarán unos enlaces a través de los cuales se podrá realizar la entrega de las prácticas. En concreto, **se habilitará un entregador para el código de la práctica, y otro de tipo TURNITIN** para la memoria de la práctica.

2.3 Documentación a Entregar

Se debe entregar un archivo comprimido en formato zip con el nombre

`ssoo_p3_AAAAAAAAA_BBBBBBBBB_CCCCCCCC.zip`

Donde A...A, B...B y C...C son los NIAs de los integrantes del grupo. En caso de realizar la práctica en solitario, el formato será **ssoo_p3_AAAAAAAAA.zip**. **El archivo zip se entregará en el entregador correspondiente al código de la práctica**. El archivo debe contener:

- **store_manager.c**
- **queue.c**
- **queue.h**
- **Makefile**
- **autores.txt: Fichero de texto en formato csv con un autor por línea. El formato es: NIA, Apellidos, Nombre**

NOTA

Para comprimir dichos ficheros y ser procesados de forma correcta por el probador proporcionado, se recomienda utilizar el siguiente comando:

```
zip ssoo_p3_AAA_BBB_CCC.zip Makefile store_manager.c queue.c queue.h
autores.txt
```

La memoria se entregará en formato PDF en un fichero llamado:

`ssoo_p3_AAAAAAAAA_BBBBBBBBB_CCCCCCC.pdf`

Solo se corregirán y calificarán memorias en formato pdf. Tendrá que contener al menos los siguientes apartados:

- **Portada** con los nombres completos de los autores, NIAs y direcciones de correo electrónico.
- **Índice** con opciones de navegación hasta los apartados indicados por los títulos.
- **Descripción del código** detallando las principales funciones implementadas. **NO** incluir código fuente de la práctica en este apartado. Cualquier código será automáticamente ignorado.
- **Batería de pruebas** utilizadas y resultados obtenidos. Se dará mayor puntuación a pruebas avanzadas, casos extremos, y en general a aquellas pruebas que garanticen el correcto funcionamiento de la práctica en todos los casos. Hay que tener en cuenta:
  1. Que un programa compile correctamente y sin advertencias (**warnings**) no es garantía de que funcione correctamente.
  2. Evite pruebas duplicadas que evalúan los mismos flujos de programa. La puntuación de este apartado no se mide en función del número de pruebas, sino del grado de cobertura de las mismas. Es mejor pocas pruebas que evalúan diferentes casos a muchas pruebas que evalúan siempre el mismo caso.
- **Conclusiones**, problemas encontrados, cómo se han solucionado, y opiniones personales.

Se puntuará también los siguientes aspectos relativos a la **presentación** de la práctica:

- Debe contener portada, con los autores de la práctica y sus NIAs.
- Debe contener índice de contenidos navegable.
- La memoria debe tener números de página en todas las páginas (menos la portada).
- El texto de la memoria debe estar justificado.

El archivo pdf se entregará en el entregador correspondiente a la memoria de la práctica (entregador TURNITIN). La longitud de la memoria no deberá superar las **15 páginas** (portada e índice incluidos). Es imprescindible aprobar la memoria para aprobar la práctica, por lo que no debe descuidar la calidad de la misma.

NOTA

Es posible entregar el código de la práctica tantas veces como se quiera dentro del plazo de entrega, siendo la última entrega realizada la versión definitiva. **LA MEMORIA DE LA PRÁCTICA ÚNICAMENTE SE PODRÁ ENTREGAR UNA ÚNICA VEZ A TRAVÉS DE TURNITIN.**

3 Normas

1. Las prácticas que no compilen o que no se ajusten a la funcionalidad y requisitos planteados, obtendrán una calificación de 0.

2. Se prestará especial atención a detectar funcionalidades copiadas entre dos prácticas. En caso de encontrar implementaciones comunes en dos prácticas, los alumnos involucrados (copiados y copiadores) perderán las calificaciones obtenidas por evaluación continua.

3. **No se permite utilizar sentencias o funciones como `goto`.**

4. Los programas deben compilar sin **warnings**.

5. Los programas deberán funcionar bajo un sistema Linux, no se permite la realización de la práctica para sistemas Windows. Además, para asegurarse del correcto funcionamiento de la práctica, deberá chequearse su compilación y ejecución en máquina virtual con Ubuntu Linux o en las Aulas Virtuales proporcionadas por el laboratorio de informática de la universidad. Si el código presentado no compila o no funciona sobre estas plataformas la implementación no se considerará correcta.

6. Un programa no comentado, obtendrá una calificación muy baja.

7. La entrega de la práctica se realizará a través de Aula Global, tal y como se detalla en el apartado Entrega de este documento. No se permite la entrega a través de correo electrónico sin autorización previa.

8. Se debe respetar en todo momento el formato de la entrada y salida que se indica en cada programa a implementar.

9. Se debe realizar un control de errores en cada uno de los programas, más allá de lo solicitado explícitamente en cada apartado.

Los programas entregados que no sigan estas normas no se considerarán aprobados.

4 Anexo

4.1 man function

man es el paginador del manual del sistema, es decir permite buscar información sobre un programa, una utilidad o una función. Véase el siguiente ejemplo:

```
man 2 fork
```

Las páginas usadas como argumentos al ejecutar man suelen ser normalmente nombres de programas, utilidades o funciones. Normalmente, la búsqueda se lleva a cabo en todas las secciones de manual disponibles según un orden predeterminado, y sólo se presenta la primera página encontrada, incluso si esa página se encuentra en varias secciones.

Una página de manual tiene varias partes. Éstas están etiquetadas como NOMBRE, SINOPSIS, DESCRIPCIÓN, OPCIONES, FICHEROS, VÉASE TAMBIÉN, BUGS, y AUTOR. En la etiqueta de SINOPSIS se recogen las librerías (identificadas por la directiva *#include*) que se deben incluir en el programa en C del usuario para poder hacer uso de las funciones correspondientes. **Para salir de la página mostrada, basta con pulsar la tecla 'q'.**

Las formas más comunes de usar *man* son las siguientes:

- **man sección elemento**: Presenta la página de elemento disponible en la sección del manual.
- **man -a elemento**: Presenta, secuencialmente, todas las páginas de elemento disponibles en el manual. Entre página y página se puede decidir saltar a la siguiente o salir del paginador completamente.
- **man -k palabra-clave**: Busca la palabra-clave entre las descripciones breves y las páginas de manual y presenta todas las que casen.

5 Bibliografía

- El lenguaje de programación C: diseño e implementación de programas Félix García, Jesús Carretero, Javier Fernández y Alejandro Calderón. Prentice-Hall, 2002.
- The UNIX System S.R. Bourne Addison-Wesley, 1983.
- Advanced UNIX Programming M.J. Rochkind Prentice-Hall, 1985.
- Sistemas Operativos: Una visión aplicada Jesús Carretero, Félix García, Pedro de Miguel y Fernando Pérez. McGraw-Hill, 2001.
- Programming Utilities and Libraries SUN Microsystems, 1990.
- Unix man pages (`man function`)