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 2022/2023

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 un banco que proporciona operaciones sobre cuentas desde cajeros automáticos, de tal forma que el saldo de las cuentas sea siempre correcto.

1.1 Descripción de la Práctica

El objetivo de esta práctica es codificar un sistema multi-hilo concurrente que que actúe como un banco que proporciona operaciones sobre cuentas desde cajeros automáticos, de tal forma que el saldo de las cuentas sea siempre correcto. Se proporciona a los alumnos un fichero que indica:

- Una lista de operaciones sobre cuentas, que incluyen: CREAR, INGRESAR, RETIRAR, SALDO, TRASPASAR.

El programa banco debe leer el archivo y crear los threads de los cajeros y de los trabajadores. Además, deberá crear una cola circular sobre la que escriben las operaciones los cajeros y de las que las sacan los trabajadores para ejecutar la operación. Se deben ejecutar las operaciones sobre las cuentas que se solicite, asegurando que el saldo es correcto al final de todas las operaciones en cada cuenta, independientemente del orden de ejecución.

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**):

- **Cajero**: Será la función que ejecuten los hilos encargados de leer las operaciones del fichero y agregar elementos en la cola circular compartida.
- **Trabajador**: Será la función que ejecuten los hilos encargados de extraer elementos de la cola circular compartida y de hacer realmente las operaciones del banco.

1. El **banco** (hilo principal) será el encargado de:

   (a) Leer los argumentos de entrada y crear los cajeros y los trabajadores que se indiquen.

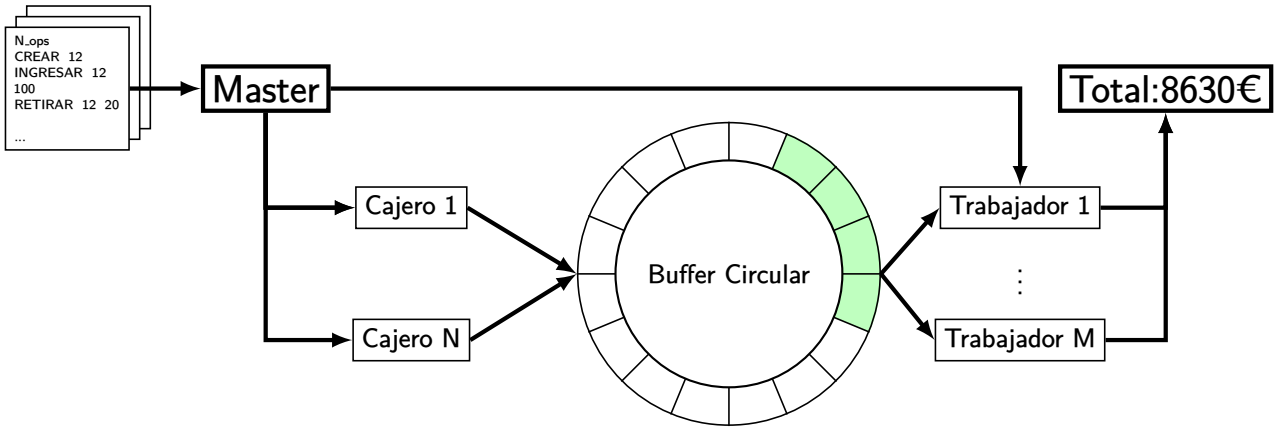

Fig. 1: Ejemplo de funcionamiento con N cajeros, M trabajadores y un buffer

(b) Crear tres variables globales: **client_numop**, que se inicia a cero y se incrementa cada vez que un cliente va a hacer una operación desde un cajero; **bank_numop**, que se inicia a cero y se incrementa cada vez que un trabajador va a hacer una operación extraída de la cola circular compartida con los cajeros; **global_balance**, que se inicia a cero y se actualiza con las operaciones que se realizan **(puede ser positivo o negativo)**.

(c) Cargar en memoria, en un vector, **list_client_ops**, la lista de las operaciones del fichero proporcionado.

(d) Lanzar los **N** cajeros y los **M** trabajadores.

(e) Esperar la finalización de todos los hilos.

2. Cada **cajero** (hilo productor) deberá:

(a) Obtener la siguiente operación de la lista **list_client_ops**, una a una, hasta que la lista esté vacía, incrementar el número de operaciones de cliente **client_numop**, e insertar la operación en la cola circular.

(b) **Esta tarea debe realizarse de forma concurrente con el resto de los cajeros, así como de los trabajadores**. 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.). El orden se hará con el número de operación (client_numop).

3. Cada **trabajador** (hilo consumidor) deberá:

(a) Incrementar el número de operación del banco (**bank_numop**) y extraer de la cola circular la operación con ese número.

(b) Para cada elemento extraído se realiza la operación bancaria indicada sobre las cuentas asociadas y además se actualiza el saldo global.

(c) Imprimir por pantalla el tipo de operación sus parámetros y el saldo resultante de hacerla.

(d) Una vez procesadas todas las operaciones de los clientes cada hilo finalizará su ejecución.

1.1.1 Programa banco

El programa principal (bank) 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:

```
./bank <nombre_fichero> <num_cajeros> <num_trabajadores> <max_cuentas>
                                                    <tam_buff>
```

La etiqueta **nombre_fichero** se corresponde con el path del fichero que se quiere importar. La etiqueta **num_cajeros** es un entero que representa el número de cajeros (hilos productores) que se quieren generar. La etiqueta **num_empleados** es un entero que representa el número de trabajadores (hilos consumidores) que se quieren generar. La etiqueta **max_cuentas** es un entero positivo que representa el número de cuentas máximo que puede tener el banco. Finalmente, la etiqueta **tam_buff** es un entero que indica el tamaño de la cola circular (número máximo de elementos que puede almacenar).

A continuación se muestra un ejemplo de contenido del fichero de entrada:

```
50 #Num max operaciones a procesar
CREAR 12
INGRESAR 12 100
RETIRAR 12 20
CREAR 25
TRASPASAR 12 25 30
SALDO 25
...
```

La primera línea indica el número de operaciones a realizar en esa sesión (**max_operaciones**). **Se debe comprobar que no se supere en número de operaciones indicadas en esta línea y que no supere las 200 operaciones máximas. Además, en ningún caso puede haber menos operaciones en el fichero que operaciones indica el primer valor**. Cada una de las líneas siguientes indica el tipo de operación y los argumentos, que son valores enteros separados por un espacio en blanco. Cabe destacar que solo hay una operación por línea y todas ellas tienen siempre el tipo de operación y el número de cuenta sobre el que realizar dicha operación.

El proceso principal debe cargar en memoria, en un vector o una cola, la información contenida en el fichero para su posterior procesamiento por los cajeros. Para leer el archivo se recomienda hacer uso de la función `scanf`. La idea es:

1. Reservar memoria para el máximo de operaciones permitidas (max_operaciones) con `malloc`.
2. Almacenar las operaciones en el array **list_client_ops**, una por elemento.
3. Pasar el argumento del array a los cajeros en el momento del lanzamiento con **pthread_create** para que lean la operación indicada en **client_numop**.
4. Tras el procesamiento de las operaciones 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.

Cada trabajador debe imprimir por pantalla las operaciones realizadas, indicando en primer lugar el orden de la operación (que debe ser global para todos los trabajadores y consecutivo), el tipo de operación, los argumentos, el saldo de la cuenta y el saldo global del banco en cada momento.

A continuación, se muestra un ejemplo de la salida del programa para el fichero de operaciones del ejemplo anterior:

```
$> ./bank input_file 5 3 50 20
1 CREAR 12 SALDO=0 TOTAL=0
2 INGRESAR 12 100 SALDO=100 TOTAL=100
3 RETIRAR 12 20 SALDO=80 TOTAL=80
4 CREAR 25 SALDO=0 TOTAL=80
5 TRASPASAR 12 25 30 SALDO=30 TOTAL=80
6 SALDO 25 SALDO=30 TOTAL=80
...
$>
```

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. Bucle desde el inicio hasta el fin de las operaciones bancarias que debe procesar:
   (a) Obtener los datos de la operación, aumentar la variable de operación de cliente (client_numop) de forma segura.
   (b) Crear un elemento con los datos de la operación para insertar en la cola, incluyendo siempre en primer lugar el número de operación
   (c) Insertar el elemento en la cola compartida.
2. Finalizar el hilo con **pthread_exit**.

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

1. Incrementar el contador global de operaciones del banco (bank_numop).
2. Extraer el elemento de la cola que se corresponde con el contador anterior.

3. Procesar la operación por el trabajador.
4. Actualizar el saldo de la cuenta correspondiente y el global del banco (bank_balance) por cada hilo consumidor para obtener el saldo total.
5. Cuando se hayan procesado todas las operaciones, finalizar el hilo con **pthread_exit**.

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.

Variables globales Además del buffer circular, es necesario manejar con los mutex y variables condición las variables globales siguientes:

- `list_client_ops`, array donde se cargan las operaciones del fichero y a la que todos los cajeros acceden.
- `client_numop`, que se inicia a cero y se incrementa concurrentemente cada vez que un cliente va a hacer una operación desde un cajero.
- `bank_numop`, que se inicia a cero y se incrementa concurrentemente cada vez que un trabajador va a hacer una operación extraída de la cola circular compartida con los cajeros.
- `global_balance`, que se modifica cada vez que se hace un ingreso o retirada de fondos.
- `saldo_cuenta[i]`, actualizar el saldo de la cuenta [i]. Se recomienda generar un vector en el que quepan el máximo de cuentas y mantener el saldo indexado por número de cuenta.

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** (struct element).

1.2 Código Fuente de Apoyo

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

`ssoo_p3_multihilo_2023.zip`

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

`unzip ssoo_p3_multihilo_2023.zip`

Al extraer su contenido, se crea el directorio ssoo_p3_multihilo_2023/, 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.
- **bank.c**
  Debe modificarse. Fichero fuente C donde los alumnos deberán codificar el programa pedido.
- **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.

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 **12 de mayo de 2023 (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:

- **bank.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 bank.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`)