

Área de Arquitectura y Tecnología de Computadores

Universidad Carlos III de Madrid



SISTEMAS OPERATIVOS

Práctica 3. Programación multi-hilo.

Grado de Ingeniería en Informática

Curso 2019/2020

Índice

Enunciado de la Práctica.....	2
Descripción de la práctica.....	3
Gestor de cálculo de costes en centro de computación.....	4
N-productores - 1-consumidor.....	5
Cola sobre un buffer circular.....	6
Corrector automático.....	7
Entrega.....	8
Plazo de entrega.....	8
Procedimiento de entrega de las prácticas.....	8
2.3 Documentación a Entregar.....	8
Normas.....	10
Anexos.....	11
5.1 man function.....	11
Bibliografía.....	12

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 procesos de fabricación incluyendo varios procesos encargados de gestionar distintas fases de la fábrica, y un proceso que realiza la planificación de las fases.

Descripción de la práctica

El objetivo de esta práctica es codificar un sistema multi-hilo concurrente que calcule el coste de utilización de las máquinas de un centro de procesamiento. Dado un fichero con un formato específico, se debe calcular cuánto se debe cobrar al cliente según el tipo de máquinas que quiere utilizar, así como el tiempo que las quiere utilizar.

Para la realización de la funcionalidad, se recomienda implementar dos funciones básicas, que representan los roles del programa:

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

De esta forma, el programa tendrá el siguiente comportamiento:

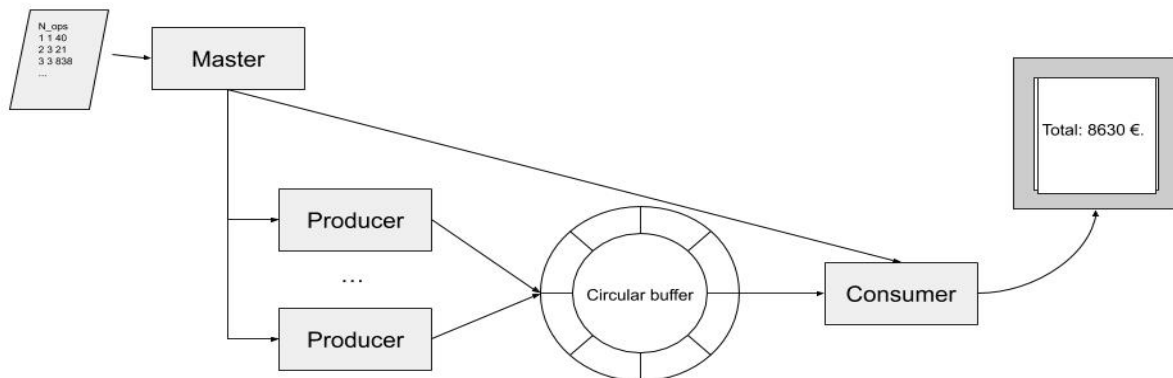


Ilustración 1. Ejemplo de funcionamiento con 1 productor, 1 consumidor y un buffer.

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 de la carga del fichero entre el número de hilos productores indicado.
 - d) Lanzar los n productores y el consumidor.
 - e) Esperar la finalización de todos los hilos y mostrar el coste total calculado (resultado de esperar a la finalización del hilo consumidor).
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 productores, así como el consumidor. 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 al consumidor para que los extraiga; luego dar paso al siguiente productor, etc.).
3. El **hilo consumidor** deberá:
- a) Obtener, de forma concurrente, los elementos insertados en la cola.
 - b) Cada elemento extraído representa un tipo de máquina y el tiempo de uso, por lo que deberá calcular el coste y acumularlo hasta que todos los elementos se hayan procesado.
 - c) Una vez procesados todos los elementos, el hilo finalizará su ejecución retornando al hilo principal el coste total calculado.

Gestor de cálculo de costes en centro de computación

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

`./calculator <file_name> <num. Producers> <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. 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
1 1 4
2 2 12
3 1 100
4 3 45
...
```

La primera línea del fichero representa el número de operaciones que se quieren calcular. 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 líneas del fichero representan una operación: **<id> <tipo de máquina> <tiempo de uso>**. Son tres valores enteros separados por un espacio y finalizado en un salto de línea. El id es consecutivo: se va incrementando con cada operación. El tipo de máquina representa si es un nodo común (coste 1€/minuto), nodo de cómputo (coste 3€/minuto), o un super-computador (coste

10€/minuto). El último valor representa el tiempo de uso. Con todo ello, el **coste total** que debe calcular el consumidor es: $\Sigma(\text{tipo} \cdot \text{tiempo})$.

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 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:

```
$> ./calculator input_file 5 10
Total: 234234 €.
$>
```

N-productores - 1-consumidor

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 del hilo consumidor se recomienda seguir un esquema similar al anterior por simplicidad:

1. Extraer elemento de la cola.
2. Procesar el coste de la operación y acumularlo.
3. Cuando se hayan procesado todas las operaciones, finalizar el hilo con *pthread_exit* devolviendo el coste total calculado.

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 ésta no está vacía. Si ésta 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:

	<p>Departamento de Informática Grado en Ingeniería Informática Sistemas Operativos (2019-2020)</p> <p>Práctica 3 – Programación Multi-hilo</p>	
---	---	---

- **int type:** 1 si es un nodo común; 2 si es un nodo de cómputo; 3 si es un super computador.
- **Int time:** representa el tiempo que se va a utilizar la máquina con las características definidas por *type*.

Corrector automático

Junto al código inicial se adjunta el corrector automático de la práctica. Para ejecutar el corrector es necesario comprimir los ficheros fuente solicitados en la entrega en un ZIP y darle el nombre apropiado. Posteriormente hay que ejecutar el script de corrección: `./corrector_ssoo_p3.sh <zip_file>`

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

Entrega

Plazo de entrega

La fecha límite de entrega de la práctica en AULA GLOBAL será el **viernes 8 de mayo de 2020 (hasta las 23:55h)**.

Procedimiento de entrega de las prácticas

La entrega de las prácticas ha de realizarse de forma electrónica. En AULA GLOBAL se habilitarán unos enlaces a través de los cuales 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

En el entregador para el código se debe entregar un archivo comprimido en formato zip con el nombre `ssoo_p3_AAAA_BBBB_CCCC.zip` donde A...A, B...B y C...C son los NIAs de los integrantes del grupo. El archivo debe contener, además de los ficheros que genere el alumno:

- `costCalculator.c`
- `queue.c`
- `queue.h`
- `Makefile`
- `Autores.txt`: fichero en formato csv con un autor en cada línea: NIA, Apellidos, Nombre.

En el entregador TURNITIN deberá entregarse el fichero `memoria.pdf`. La memoria tendrá que contener al menos los siguientes apartados:

- **Portada:** con los nombres completos de los autores, NIAs, grupo al que pertenecen 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:
 - Que un programa compile correctamente y sin advertencias (*warnings*) no es garantía de que funcione correctamente.
 - 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.
 - Evitar el uso de capturas de pantalla de terminales y código.
- **Conclusiones**, problemas encontrados, cómo se han solucionado, y valoraciones 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.

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: El entregador de la memoria permite una **única entrega**. La valoración de esta entrega es la única válida y definitiva.

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 copiadore) perderán las calificaciones obtenidas por evaluación continua.
- 3) Los programas deben compilar sin warnings.
- 4) 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 los laboratorios de informática de la universidad o en el servidor `guernika.lab.inf.uc3m.es`. Si el código presentado no compila o no funciona sobre estas plataformas la implementación no se considerará correcta.
- 5) Un programa no comentado, obtendrá una calificación de 0.
- 6) 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.
- 7) Se debe respetar en todo momento el formato de la entrada y salida que se indica en cada programa a implementar.
- 8) Se debe realizar un control de errores en cada uno de los programas.

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

Anexos

5.1 *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.

Para salir de la página mostrada, basta con pulsar la tecla 'q'.

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.

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.

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)