PROGRAMACIÓN CONCURRENTE Y TIEMPO REAL
PRÁCTICA 1.2

JORGE LÓPEZ GÓMEZ
OUSSAMA BOLBAROUD

UCLM

ÍNDICE

INTRODUCCIÓN

En este documento se describen las especificaciones para desarrollar un sistema que consta de tres programas que simulen un escenario de procesamiento concurrente. Este sistema se compone de tres tipos de procesos: el Manager, los Procesadores y los Contadores.

El Manager es responsable de supervisar la creación, ejecución y finalización de los procesos Procesador y Contador. Se ingresa la ubicación de dos archivos: uno contiene texto y el otro contiene patrones. Utilizando el archivo de texto, se generan operaciones de Contador, a la vez que se realizan operaciones de Procesador con el archivo de patrones.

Los Procesadores tienen la responsabilidad de detectar patrones particulares en el texto que se encuentra en el archivo dado. Se encarga de verificar si un patrón dado corresponde con alguna palabra del archivo de texto.

Además, los Contadores deben contar cuántas palabras hay en cada línea del documento de texto y luego este dato debe de ser mostrado por pantalla.

El Manager debe asegurar que el programa termine cuando todos los Procesadores y Contadores hayan finalizado o si el usuario presiona Ctrl + C, de lo contrario si algunos procesos quedasen abiertos estaríamos perdiendo efectividad y eficiencia de computación.

Mediante una demostración práctica, se muestra cómo el Manager gestiona la creación, ejecución y finalización de los procesos Procesador y Contador, exhibiendo el resultado de su desempeño.

El sistema propuesto ofrece una herramienta para procesar texto de manera eficiente utilizando varios procesos concurrentes, mostrando un enfoque práctico para manejar procesos en entornos de programación concurrente y tiempo real.

```
[jorgelg@manjaro esqueleto]$ ./exec/manager data/solution.txt data/patrones.txt
[MANAGER] 6 procesos creados.
[CONTADOR 3390] La linea '0' tiene 10 palabras
[CONTADOR 3392] La linea '2' tiene 4 palabras
[CONTADOR 3391] La linea '1' tiene 10 palabras
[PROCESADOR 3395] Patron 'calvito' encontrado en la linea 1
[PROCESADOR 3393] Patron 'Pablito' encontrado en la linea 1
[PROCESADOR 3395] Patron 'calvito' encontrado en la linea 2
[PROCESADOR 3393] Patron 'Pablito' encontrado en la linea 2
[PROCESADOR 3393] Patron 'Pablito' encontrado en la linea 3
[PROCESADOR 3394] Patron 'clavito' encontrado en la linea 1
[PROCESADOR 3394] Patron 'clavito' encontrado en la linea 2
[PROCESADOR 3394] Patron 'clavito' encontrado en la linea 3
[MANAGER] Proceso CONTADOR terminado [3390]...
[MANAGER] Proceso CONTADOR terminado [3391]...
[MANAGER] Proceso CONTADOR terminado [3392]...
[MANAGER] Proceso PROCESADOR terminado [3393]...
[MANAGER] Proceso PROCESADOR terminado [3394]...
[MANAGER] Proceso PROCESADOR terminado [3395]...

[MANAGER] Terminacion del programa (todos los procesos terminados).
```

MANAGER

El archivo "**manager.c"** es el núcleo fundamental de un programa que simula un sistema de procesos concurrentes. Dentro de este archivo reside la esencia del proceso "**Manager**", encargado de coordinar y gestionar el comportamiento de los demás procesos en el sistema.

A continuación, vamos a hablar sobre algunas de las funciones más importantes del archivo "**manager.c**":

- **procesar_argumentos()**: Esta función es responsable de analizar los argumentos que se pasan al programa. Dichos argumentos incluyen el nombre del archivo, el patrón a buscar y el número de líneas en el archivo. Dentro de este método se realizan procesos fundamentales para la supervivencia del programa, por ejemplo: es fundamental que el número de argumentos que se pasan en ejecución sean exactamente 3. Además, hay que comprobar si el archivo existe y se puede abrir, para utilizaremos la función fopen y le pasaremos como parámetros el nombre del archivo y que tenga los permisos en modo lectura "r" para poder trabajar con el programa. Intentar acceder a un archivo que no existe o que no podemos abrir es considerado un error catastrófico que finaliza los procesos. Una vez terminada la comprobación de que todos los argumentos son correctos y utilizables, se procede a la lectura línea a línea del archivo de texto.
- **instalar_manejador_senhal()**: En esta función se instala un manejador de señal. Este manejador se encarga de gestionar las señales enviadas al proceso, como la señal de terminación. En caso de que exista un error en el proceso, se informa al usuario.
- **manejador_senhal()**: Es una función que sirve para controlar la terminación de los procesos mediante una señal Ctrl + C y liberar el espacio ocupado para que pueda ser utilizado en futuros programas.
- **procesar_patrones()**: Su tarea consiste en examinar los patrones contenidos en el archivo. Lee cada patrón del archivo correspondiente y, por cada uno encontrado, crea un nodo en una lista para su posterior uso. Inicialmente comprueba que se puede acceder al archivo de patrones ya que sino el proceso tendría que terminar. Después, se lee el archivo línea a línea y se pasa cada una de esas líneas a una nueva función que se encarga de procesarlas. Finalmente, cierra el archivo para liberar recursos.
- **procesar_linea()**: Es una función que sirve para ir obteniendo cada una de las palabras que contiene la línea utilizando como separador el espacio en blanco y después lo va insertando al final de la lista de caracteres.
- **iniciar_tabla_procesos()**: Aquí se establece el inicio de la tabla de procesos. Esta estructura de datos mantiene un registro de todos los procesos creados por el manager durante la ejecución del programa, es decir, una tabla que permite tener un resumen de todos los procesos que se encuentran en ejecución identificados por su PID.
- **crear_procesos()**: Esta función desempeña el papel crucial de generar los procesos necesarios. Por cada línea en el archivo de texto, crea un proceso contador. Asimismo, por cada nodo en la lista de patrones, genera un proceso procesador. Todo esto nos permitirá más adelante llevar la cuenta de cuantas coincidencias hay y en que línea.
- **lanzar_proceso_contador()**: es la inicialización del proceso contador, a partir del PID del proceso y tras comprobar que no es un PID erróneo, asigna el PID del proceso a la tabla de procesos y asigna a la clase a la tabla de procesos.
- **lanzar_proceso_procesador()**: hace lo mismo que la función lanzar procesos contador, pero a diferencia de este, se encarga de hacerlos con los procesos en lugar de con los contadores.
- **esperar_procesos()**: Su función principal es aguardar a que todos los procesos hayan finalizado su ejecución. Cuando todos los procesos completen sus tareas, el manager puede liberar los recursos utilizados.
- **liberar_recursos()**: Esta función se encarga de liberar los recursos que han sido reservados por el manager durante la ejecución del programa. Esto incluye la liberación de memoria asignada para la lista de patrones y la tabla de procesos.
- **terminación_procesos()**: Esta función sirve para recorrer la tabla de procesos y siempre y cuando el proceso siga vivo, lo para y lo finaliza liberando asi los recursos que estaba utilizando y volviendo todo a como estaba antes de la ejecución del programa.

En la función "**main**", se va a ejecutar una secuencia lógica donde se procesan los argumentos, se instala el manejador de señales, se procesan los patrones del archivo, se inicia la tabla de procesos, se crean los procesos y se espera a que todos terminen. Por último, se procede a la liberación de los recursos.

Finalmente, podemos llegar a la conclusión que el archivo "**manager.c**" es de suma importancia para el funcionamiento integral del programa, ya que centraliza la creación, gestión y finalización de los procesos. Además, cumple un papel vital en el manejo de las señales y la liberación adecuada de recursos una vez que todos los procesos han completado sus tareas.

PROCESADOR

El archivo "**procesador.c**" contiene la implementación del proceso "**Procesador**" dentro del sistema de simulación de procesamiento de texto, siendo gestionado por el proceso "**Manager**". Este proceso es responsable de buscar patrones específicos dentro del texto contenido en un archivo dado, como parte de la funcionalidad más amplia del sistema.

En el contexto del proceso "**Manager**", el archivo "**procesador.c**" se ejecuta como un proceso hijo, creado dinámicamente según la necesidad de buscar patrones en el texto. El proceso "**Manager**" coordina la creación, ejecución y finalización de múltiples instancias del proceso "**Procesador**", cada una encargada de buscar un patrón particular en el texto, es decir, de crear un proceso por patrón contenido en el archivo de patrones.

La función "**main**" en "**procesador.c**" maneja la lógica específica del proceso "**Procesador**". A continuación, se describen las acciones principales llevadas a cabo por este proceso:

1. **Procesamiento de argumentos**: El proceso "**Procesador**" recibe como argumentos el nombre del archivo de texto y un patrón específico a buscar dentro de dicho texto. Estos argumentos se pasan al proceso mediante la línea de comandos al crearlo. Tendrá que comprobar que el número de argumentos es el correcto, si no fuera así, se informa al usuario y se para la ejecución.
2. **Apertura del archivo de texto**: El proceso "**Procesador**" abre el archivo de texto proporcionado por el "**Manager**" para su lectura, lo que le permite acceder al contenido del texto y buscar el patrón especificado. Este proceso debe comprobar que se puede abrir el archivo en modo lectura, si no, informar al usuario y parar la ejecución del programa.
3. **Búsqueda de patrones**: Utilizando un algoritmo de búsqueda, el proceso "**Procesador**" examina el contenido del texto línea por línea en busca del patrón específico proporcionado. Cuando se encuentra una coincidencia, se imprime un mensaje que indica la línea en la que se encontró el patrón.
4. **Finalización del proceso**: Una vez que se ha examinado todo el texto en busca de patrones, el proceso "**Procesador**" finaliza su ejecución y retorna el control al proceso "**Manager**", que puede continuar con otras tareas o finalizar la simulación en su conjunto, según la lógica del sistema. Para ello, cierra los archivos, liberando memoria y devuelve un éxito para la salida.

En resumen, el archivo "**procesador.c**" define el comportamiento del proceso "**Procesador**" dentro del sistema de simulación de procesamiento de texto, trabajando en conjunto con el proceso "**Manager**" para buscar patrones específicos en el texto proporcionado.

CONTADOR

El archivo "**contador.c**" es un programa que despliega una funcionalidad esencial: contar el número de palabras en una línea de texto dada. Algunas de las partes más importantes de este programa son las siguientes:

1. **Declaración de la función "contar()"**: En esta parte, se declara la función "**contar()**", que se encargará de realizar la tarea principal del programa. Esta función, aunque se define más adelante en el código, tiene la importante responsabilidad de contar el número de palabras en una línea de texto específica.
2. **Función "main()"**: Esta función es la puerta de entrada del programa. Comienza por verificar si se han proporcionado los argumentos necesarios para su ejecución. Si falta alguno, el programa muestra un mensaje de error y se detiene. De lo contrario, procesa los argumentos y llama a la función "**contar**".
3. **Función "contar()"**: A diferencia de la declaración de la función "**contar()**". Esta función es la más importante del archivo "**contador.c**". Su tarea principal es recorrer cada carácter de la línea de texto y determinar cuántas palabras contiene. Lo hace identificando tanto los espacios, como las tabulaciones, como saltos de línea o como retornos de carro, y contando el número de transiciones entre estos caracteres para calcular el número total de palabras en la línea. Al final, imprime el número de línea y la cantidad de palabras encontradas en ella.

Finalmente, el programa "**contador.c**" cumple una función aparentemente simple pero de gran importancia: contar palabras. Aunque su tarea puede parecer básica, es esencial en numerosos contextos donde se requiere procesar texto de manera eficiente y precisa.

LISTA

El archivo "**lista.c**" contiene una serie de funciones que operan sobre una estructura de datos de lista enlazada. Estas funciones proporcionan operaciones comunes para manipular y administrar listas enlazadas, como la creación, inserción, eliminación, obtención de elementos y la impresión de la lista. Algunas de las funciones son:

- **void crear(TLista *pLista, char *valor)**: Esta función crea una lista enlazada con un nodo inicial. Recibe un puntero a la estructura de lista TLista y el valor que se asignará al nodo. Reserva memoria para el primer nodo, copia el valor pasado como argumento al nodo y establece el puntero al siguiente nodo como NULL.
- **void destruir(TLista *pLista)**: Elimina todos los nodos de la lista, liberando la memoria ocupada por cada nodo y la propia estructura de lista. Utiliza un bucle para recorrer la lista y liberar cada nodo individualmente.
- **void insertar(TLista *pLista, char *valor)**: Inserta un nuevo nodo al principio de la lista. Reserva memoria para el nuevo nodo, copia el valor pasado como argumento y establece el puntero al siguiente nodo como el primer nodo actual.
- **void insertarFinal(TLista *pLista, char *valor)**: Inserta un nuevo nodo al final de la lista. Si la lista está vacía, llama a la función crear para crear un nodo. Luego, recorre la lista hasta encontrar el último nodo y agrega el nuevo nodo después de él.
- **void insertarN(TLista *pLista, int index, char *valor)**: Inserta un nuevo nodo en la posición index de la lista. Si index es 0, llama a insertar para insertar al principio. Si index es mayor que la longitud de la lista, imprime un mensaje de error y termina la ejecución. En caso contrario, recorre la lista hasta el nodo en la posición index - 1 e inserta el nuevo nodo después de él.

- **void eliminar(TLista *pLista)**: Elimina el primer nodo de la lista. Libera la memoria del nodo eliminado y actualiza el puntero al primer nodo.
- **void eliminarN(TLista *pLista, int index)**: Elimina el nodo en la posición index de la lista. Si index es 0, elimina el primer nodo. Si index es mayor que la longitud de la lista, imprime un mensaje de error y termina la ejecución. En caso contrario, recorre la lista hasta el nodo en la posición index - 1 y elimina el nodo siguiente.
- **char *getElementoN(TLista *pLista, int index)**: Obtiene el valor del nodo en la posición index de la lista. Si index es negativo o mayor que la longitud de la lista, imprime un mensaje de error y devuelve NULL. En caso contrario, recorre la lista hasta el nodo en la posición index y devuelve su valor.
- **void imprimir(TLista *pLista)**: Imprime los valores de todos los nodos de la lista. Utiliza un bucle para recorrer la lista y mostrar los valores de los nodos uno por uno.
- **int longitud(TLista *pLista)**: Devuelve la longitud de la lista. Utiliza un bucle para contar el número de nodos en la lista y asi, obtienes la longitud de la lista.

HEADERS

Los archivos de cabecera en C, se utilizan para definir funciones, variables y tipos de datos que se pueden utilizar en diferentes archivos de código fuente.

Por ejemplo el archivo de cabecera "**lista.h**" define una estructura de lista enlazada y las operaciones que se pueden realizar en ella permitiendo que otros archivos de código fuente incluyan "**lista.h**" y utilicen estas estructuras y operaciones.

Este tipo de archivos es de gran utilidad ya que permiten:

- **Reutilización de código**: Se puede definir funciones y tipos de datos en un archivo de cabecera y luego reutilizarlos en varios archivos de código fuente.
- **Organización del código**: Los archivos de cabecera ayudan a organizar el código al separar las definiciones, que irán en archivos de cabecera y las implementaciones, que irán en archivos de código fuente.
- **Ocultación de detalles de implementación**: Al incluir solo la definición de una función en un archivo de cabecera, puedes ocultar los detalles de su implementación.

El archivo "**definitions.h**" es un archivo de cabecera en C que define constantes y una estructura que se pueden utilizar en otros archivos de código fuente.

#define CLASE_PROCESADOR "PROCESADOR" y **#define CLASE_CONTADOR "CONTADOR"**: Estas son directivas de preprocesador que definen constantes para las clases de procesos.

#define RUTA_PROCESADOR "./exec/procesador" y **#define RUTA_CONTADOR "./exec/contador"**: Estas son directivas de preprocesador que definen las rutas a los ejecutables de los procesos.

struct TProcess_t: Esta es una definición de estructura que representa un proceso. Cada proceso tiene un PID (**pid_t pid**) y una clase (**char *clase**). El tipo **pid_t** es un tipo de dato entero que se utiliza para representar PID. La clase probablemente se refiere a si el proceso es un "**PROCESADOR**" o un "**CONTADOR**".

MAKEFILE

El archivo **Makefile** es un guion o script que establece las reglas y comandos necesarios para compilar y construir un proyecto. Escrito en un lenguaje especializado de **Make**, este archivo es interpretado por el programa **make**, que ejecuta las instrucciones especificadas para llevar a cabo la compilación y construcción del proyecto que en este caso estamos haciendo en C.

Ahora, vamos a desarrollar lo que contiene el **Makefile**:

1. **Definición de directorios**: Se establecen variables que representan los directorios utilizados en el proyecto, como el directorio de objetos ("**DIROBJ**"), el de ejecutables ("**DIREXE**"), el de encabezados ("**DIRHEA**") y el de fuentes ("**DIRSRC**").
2. **Definición de variables de compilación**: Se definen las variables "**CFLAGS**", "**LDLIBS**" y "**CC**". "**CFLAGS**" contiene las banderas de compilación, como las opciones de inclusión de directorios y las advertencias activadas. "**LDLIBS**" especifica las bibliotecas que se enlazarán durante la fase de enlace. "**CC**" determina el compilador que se utilizará, en este caso, **gcc**.
3. **Objetivo de** "**all**": Es el objetivo predeterminado cuando se llama al comando "**make**" sin argumentos. Dependiendo de las reglas de los objetivos, "**make**" ejecutará las reglas de construcción para los objetivos especificados, que en este caso son "**manager**", "**procesador**", "**contador**" y "**test**".
4. **Objetivo de** "**dirs**": Crea los directorios necesarios ("**DIROBJ**" y "**DIREXE**") utilizando el comando "**mkdir -p**".
5. **Objetivos de** "**manager**", "**procesador**" y "**contador**": Construyen los ejecutables correspondientes a cada parte del proyecto, utilizando los archivos objeto correspondientes. Estos objetivos dependen de sus archivos objeto respectivos y se especifica cómo construirlos.
6. **Regla de construcción de archivos objeto**: Se utiliza una regla de patrón para generar automáticamente los archivos objeto a partir de los archivos fuente. Esto simplifica el proceso de compilación y asegura que los archivos objeto se generen correctamente.
7. **Objetivo de** "**test**": Ejecuta el programa "**manager**" con los argumentos "**data/solution.txt**" y "**data/patrones.txt**", lo que permite realizar pruebas rápidas del sistema.
8. **Objetivo de** "**clean**": Elimina todos los archivos generados durante el proceso de compilación y construcción, incluidos los archivos objeto, los ejecutables y cualquier archivo temporal. Esto ayuda a mantener el directorio del proyecto limpio y organizado.

Como conclusión, el **Makefile** proporciona un conjunto de reglas y comandos que automatizan el proceso de compilación, construcción y ejecución del proyecto, facilitando así su desarrollo y mantenimiento.

EJEMPLO DE EJECUCIÓN

Para ejecutar el programa realizado, tenemos dos opciones igualmente válidas. La primera partiría de la utilización de la sentencia obligatoria por la terminal desde el directorio **/esqueleto**.

Consola: $ **make clean && make**. Como se puede observar en la captura tras la ejecución del comando se crea la estructura de carpetas y gcc necesarios para el correcto funcionamiento del programa. Seguidamente se crea el ejecutable y se ejecuta.

```
[jorgelg@manjaro esqueleto]$ make clean && make
rm -rf *~ core obj/ exec/ include/*~ src/*~
mkdir -p obj/ exec/
gcc -Iinclude/ -c -Wall -ggdb src/manager.c -o obj/manager.o
gcc -Iinclude/ -c -Wall -ggdb src/lista.c -o obj/lista.o
gcc -o exec/manager obj/manager.o obj/lista.o -lpthread -lrt
gcc -Iinclude/ -c -Wall -ggdb src/procesador.c -o obj/procesador.o
gcc -o exec/procesador obj/procesador.o -lpthread -lrt
gcc -Iinclude/ -c -Wall -ggdb src/contador.c -o obj/contador.o
gcc -o exec/contador obj/contador.o -lpthread -lrt
./exec/manager data/solution.txt data/patrones.txt
[MANAGER] 6 procesos creados.
[CONTADOR 3213] La linea '0' tiene 10 palabras
[CONTADOR 3215] La linea '2' tiene 4 palabras
[CONTADOR 3214] La linea '1' tiene 10 palabras
[PROCESADOR 3217] Patron 'clavito' encontrado en la linea 1
[PROCESADOR 3217] Patron 'clavito' encontrado en la linea 2
[PROCESADOR 3217] Patron 'clavito' encontrado en la linea 3
[PROCESADOR 3216] Patron 'Pablito' encontrado en la linea 1
[PROCESADOR 3216] Patron 'Pablito' encontrado en la linea 2
[PROCESADOR 3216] Patron 'Pablito' encontrado en la linea 3
[PROCESADOR 3218] Patron 'calvito' encontrado en la linea 1
[PROCESADOR 3218] Patron 'calvito' encontrado en la linea 2
[MANAGER] Proceso CONTADOR terminado [3213]...
[MANAGER] Proceso CONTADOR terminado [3214]...
[MANAGER] Proceso CONTADOR terminado [3215]...
[MANAGER] Proceso PROCESADOR terminado [3216]...
[MANAGER] Proceso PROCESADOR terminado [3217]...
[MANAGER] Proceso PROCESADOR terminado [3218]...

[MANAGER] Terminacion del programa (todos los procesos terminados).
```

La segunda opción consistiría en saltarnos la creación de carpetas y directorios e ir directamente a la ejecución, pasando por parámetros los archivos necesarios para la búsqueda de patrones. Esta alternativa tiene que hacer obligatoriamente uso de una estructura de ejecutables que haya sido creada antes con un **make clean && make**.

Consola: $ **./exec/manager data/solution.txt data/patrones.txt**

```
[jorgelg@manjaro esqueleto]$ ./exec/manager data/solution.txt data/patrones.txt
[MANAGER] 6 procesos creados.
[CONTADOR 3390] La linea '0' tiene 10 palabras
[CONTADOR 3392] La linea '2' tiene 4 palabras
[CONTADOR 3391] La linea '1' tiene 10 palabras
[PROCESADOR 3395] Patron 'calvito' encontrado en la linea 1
[PROCESADOR 3393] Patron 'Pablito' encontrado en la linea 1
[PROCESADOR 3395] Patron 'calvito' encontrado en la linea 2
[PROCESADOR 3393] Patron 'Pablito' encontrado en la linea 2
[PROCESADOR 3393] Patron 'Pablito' encontrado en la linea 3
[PROCESADOR 3394] Patron 'clavito' encontrado en la linea 1
[PROCESADOR 3394] Patron 'clavito' encontrado en la linea 2
[PROCESADOR 3394] Patron 'clavito' encontrado en la linea 3
[MANAGER] Proceso CONTADOR terminado [3390]...
[MANAGER] Proceso CONTADOR terminado [3391]...
[MANAGER] Proceso CONTADOR terminado [3392]...
[MANAGER] Proceso PROCESADOR terminado [3393]...
[MANAGER] Proceso PROCESADOR terminado [3394]...
[MANAGER] Proceso PROCESADOR terminado [3395]...

[MANAGER] Terminacion del programa (todos los procesos terminados).
```

También, se ha programado como alternativa la posibilidad de terminar los procesos del programa de manera anticipada a su terminación natural mediante la combinación de teclas: **Ctrl + C**. Como puede verse en la captura, esta combinación termina con todos los procesos y cierra el programa.

```
[jorgelg@manjaro esqueleto]$ ./exec/manager data/solution.txt data/patrones.txt
[MANAGER] 6 procesos creados.
[CONTADOR 3453] La linea '0' tiene 10 palabras
[CONTADOR 3454] La linea '1' tiene 10 palabras
[PROCESADOR 3456] Patron 'Pablito' encontrado en la linea 1
[CONTADOR 3455] La linea '2' tiene 4 palabras
[PROCESADOR 3456] Patron 'Pablito' encontrado en la linea 2
[PROCESADOR 3456] Patron 'Pablito' encontrado en la linea 3
[PROCESADOR 3457] Patron 'clavito' encontrado en la linea 1
[PROCESADOR 3457] Patron 'clavito' encontrado en la linea 2
[PROCESADOR 3457] Patron 'clavito' encontrado en la linea 3
[PROCESADOR 3458] Patron 'calvito' encontrado en la linea 1
[PROCESADOR 3458] Patron 'calvito' encontrado en la linea 2
^C
[MANAGER] Terminacion del programa (Ctrl + C).

----- [MANAGER] Terminar con los procesos hijos ejecutándose -----
[MANAGER] Terminando proceso CONTADOR [3453]...
[MANAGER] Terminando proceso CONTADOR [3454]...
[MANAGER] Terminando proceso CONTADOR [3455]...
[MANAGER] Terminando proceso PROCESADOR [3456]...
[MANAGER] Terminando proceso PROCESADOR [3457]...
[MANAGER] Terminando proceso PROCESADOR [3458]...
```