PROGRAMACIÓN CONCURRENTE Y TIEMPO REAL
PRÁCTICA 2

JORGE LÓPEZ GÓMEZ
OUSSAMA BOLBAROUD
UCLM

ÍNDICE

INTRODUCCIÓN

En esta segunda práctica nos piden que implementemos un programa basado en la utilización de semáforos como recurso para sincronizar la utilización y aterrizaje de una serie de aviones en las pistas de un aeropuerto. Se exigen diferentes pautas de como debe funcionar el sistema de procesos concurrentes.

Inicialmente contamos con un número de aviones y un número de pistas determinado por el usuario. El funcionamiento es el siguiente: cada pista tiene que entrar en funcionamiento antes de poder usarse, una vez inicializada la pista, será usada por un avión para aterrizar, ese avión junto con los otros n aviones iniciales, han sido inicializados y están en espera a que haya alguna pista libre donde puedan aterrizar. Mientras que una pista esta ocupada, el resto de aviones esperan y no intentan ocupar la pista. Cuando el avión aterriza, la pista vuelve a quedar libre y podrá ser utilizada por cualquier otro avión.

Todo proceso debe ser impreso por pantalla para que pueda seguirse de manera clara, utilizándose para ello el "**PID**" o también llamado identificador del proceso, consiguiéndose así saber que pista está ocupada en cada momento, que aviones están aterrizando y cuales están en espera. Del mismo modo también se debe imprimir la terminación de los procesos para asegurarnos de que toda la memoria compartida utilizada queda libre para futuros usos.

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. Se va a encargar de verificar y procesar los argumentos de la línea de comandos proporcionados al programa. Espera exactamente 3 argumentos (incluyendo el nombre del programa). Los dos argumentos adicionales deben ser enteros positivos que representan el número de pistas (numPistas) y el número de aviones (numAviones). Si los argumentos no cumplen con estos criterios, la función imprime un mensaje de error y termina el programa. Si los argumentos son válidos, los convierte a enteros y los almacena en las variables apuntadas por numPistas y numAviones.

- **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.
- **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. A recibir por parámetros el número de pistas y el número de aviones que deseamos tener, el método se encarga de crear y lanzar un proceso por cada pista y un proceso por cada avión.
- **lanzar_proceso_pista()**: es la inicialización del proceso pista, 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 la tabla de procesos.
- **lanzar_proceso_avion()**: es la inicialización del proceso avión, 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 la tabla de procesos.
- **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. La función espera a que todos los procesos de aviones terminen. Inicialmente, establece el número de procesos a esperar. Luego, entra en un bucle, esperando a que cada proceso termine y disminuyendo el contador de procesos. Cuando un proceso termina, busca su PID en la tabla de procesos de aviones, imprime un mensaje de terminación, reinicia el PID en la tabla y sale del bucle interno. Este proceso se repite hasta que todos los procesos de aviones hayan terminado.
- **terminación_procesos()**: esta función se encarga de terminar todos los procesos pendientes. Primero imprime un mensaje indicando que va a terminar cualquier proceso pendiente. Luego, llama a la función "**terminar_procesos_especificos**" dos veces, una para los procesos de aviones y otra para los procesos de pistas. Finalmente, imprime un mensaje indicando que todos los aviones están en tierra y todas las pistas están cerradas.
- **terminar_procesos_especificos()**: la función principal es recorrer una tabla de procesos y terminar cada proceso cuyo PID no es 0. Imprime un mensaje indicando que está terminando el proceso y luego intenta enviar la señal SIGINT al proceso. Si la llamada a kill falla, imprime un mensaje de error.
- **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 destrucción de los semáforos y la destrucción de la memoria compartida. Por último, se liberará todos los procesos presentes en la tabla de pistas y aviones.

En la función "**main**", se va a ejecutar una secuencia lógica donde se procesan los argumentos, se instala el manejador de señales, se procesan las pistas y aviones, 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.

PISTA

El archivo "**pista.c**" tiene un "**main**" que va a ser el punto de entrada del programa. Esta función realiza las siguientes operaciones:

- Define varias variables locales, incluyendo el identificador de proceso ("**pid**"), un generador de números aleatorios inicializado con el "**pid**" y una variable "**valorEspera**" que se inicializa a 0.
- Obtiene los semáforos y la memoria compartida utilizando las funciones "**get_sem**" y "**obtener_var**". Los semáforos se utilizan para sincronizar los procesos y la memoria compartida se utiliza para almacenar el número de aviones en espera.
- Entra en un bucle infinito donde realiza las siguientes operaciones:
    - Imprime un mensaje indicando que la pista está en servicio y luego señaliza el semáforo "**sem_pistas**" y espera en el semáforo "**sem_aviones**".
    - Espera en el semáforo "**sem_mutex**", consulta el valor de la variable de memoria compartida, decrementa el valor de "**valorEspera**", modifica el valor de la variable de memoria compartida y luego señaliza el semáforo "**sem_mutex**".
    - Imprime un mensaje indicando que la pista está ocupada y el número de aviones en espera.
    - Duerme durante un tiempo aleatorio entre 30 y 60 segundos.

Finalmente, devuelve "**EXIT_SUCCESS**" para indicar que el programa ha terminado con éxito. Sin embargo, dado que la función está en un bucle infinito, esta línea de código nunca se alcanzará a menos que el programa se interrumpa mediante una señal.

AVION

El archivo "**avion.c**" tiene un "**main**" que va a ser el punto de entrada del programa. Esta función realiza las siguientes operaciones:

- Define varias variables locales, incluyendo el identificador de proceso ("**pid**"), un generador de números aleatorios inicializado con el "**pid**" y una variable "**valorEspera**" que se inicializa a 0.
- Obtiene los semáforos y la memoria compartida utilizando las funciones "**get_sem**" y "**obtener_var**". Los semáforos se utilizan para sincronizar los procesos y la memoria compartida se utiliza para almacenar el número de aviones en espera.
- Imprime un mensaje indicando que el avión está en camino al aeropuerto y luego duerme durante un tiempo aleatorio entre 30 y 60 segundos.
- Espera en el semáforo "**sem_mutex**", consulta el valor de la variable de memoria compartida, incrementa el valor de "**valorEspera**", modifica el valor de la variable de memoria compartida y luego señaliza el semáforo "**sem_mutex**". Luego imprime el número de aviones en espera.
- Imprime un mensaje indicando que el avión está esperando una pista libre, luego espera en el semáforo "**sem_pistas**" y señaliza el semáforo "**sem_aviones**".
- Imprime un mensaje indicando que el avión está comenzando a aterrizar y luego duerme durante 60 segundos.
- Imprime un mensaje indicando que el avión ha aparcado.

Finalmente, devuelve "**EXIT_SUCCESS**" para indicar que el programa ha terminado con éxito. Sin embargo, dado que la función está en un bucle infinito, esta línea de código nunca se alcanzará a menos que el programa se interrumpa mediante una señal.

SEMAFOROI

En el archivo "**semaforoI.c**" se maneja un proceso de semáforos que son una herramienta esencial para la sincronización entre procesos, permitiendo controlar el acceso concurrente a recursos compartidos y evitar condiciones de carrera. Se compone de las siguientes funciones:

- **crear_sem():** se encarga de crear o abrir un semáforo con un nombre y valor inicial dados. Si la creación del semáforo falla, imprime un mensaje de error y termina el programa. En cambio, si tiene éxito, devuelve un puntero al semáforo.
- **get_sem():** va a proporcionar un semáforo existente con un nombre dado. Si la apertura del semáforo falla, imprime un mensaje de error y termina el programa. En cambio, si tiene éxito, devuelve un puntero al semáforo.
- **destruir_sem():** cierra y elimina un semáforo existente con un nombre dado. Si el cierre o la eliminación del semáforo fallan, imprime un mensaje de error y termina el programa.
- **signal_sem():** es una función que incrementa (señaliza) un semáforo dado. Si la operación falla, imprime un mensaje de error y termina el programa.
- **wait_sem():** es una función que decrementa (espera) un semáforo. Si la operación falla, imprime un mensaje de error y termina el programa.

MEMORIAI

El archivo "**memoriaI.c**" contiene funciones para trabajar con memoria compartida. Las funciones que lo van a componer serán las siguientes:

- **crear_var()**: Esta función crea un nuevo objeto de memoria compartida con un nombre y valor dados. Si no puede abrir, establecer el tamaño o mapear el objeto de memoria compartida, imprime un mensaje de error y termina el programa. Finalmente, establece el valor del objeto de memoria compartida al valor dado, desmapea el objeto de memoria compartida, y devuelve el descriptor de archivo del objeto de memoria compartida.
- **obtener_var()**: Esta función abre un objeto de memoria compartida existente con un nombre dado y devuelve su descriptor de archivo. Si no puede abrir el objeto de memoria compartida, imprime un mensaje de error y termina el programa.
- **destruir_var()**: Esta función destruye un objeto de memoria compartida existente con un nombre dado. Primero, obtiene el descriptor de archivo del objeto de memoria compartida. Luego, cierra el descriptor de archivo y desvincula el nombre del objeto de memoria compartida. Si no puede cerrar el descriptor de archivo o desvincular el nombre, imprime un mensaje de error y termina el programa.
- **modificar_var()**: Esta función modifica el valor de un objeto de memoria compartida existente con un descriptor de archivo y valor dados. Primero, mapea el objeto de memoria compartida en el espacio de direcciones del proceso. Si no puede mapear el objeto, imprime un mensaje de error y termina el programa. Luego, establece el valor del objeto de memoria compartida al valor dado y desmapea el objeto de memoria compartida.
- **consultar_var()**: Esta función consulta el valor de un objeto de memoria compartida existente con un descriptor de archivo dado y guarda el valor en un puntero dado. Primero, mapea el objeto de memoria compartida en el espacio de direcciones del proceso. Si no puede mapear el objeto, imprime un mensaje de error y termina el programa. Luego, obtiene el valor del objeto de memoria compartida y lo guarda en el puntero dado, y desmapea el objeto de memoria compartida.

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. Los headers (include) están divididos en 3 archivos que son **definitions.h**, **memoriaI.h** y **semaforoI.h**. En cada uno de ellos se definen las variables y constantes necesarios para el funcionamiento del programa:

1. **definitions.h**:
   - Contiene lo siguiente:
     - Constantes para los nombres de los semáforos: "**MUTEXESPERA**", "**PISTAS**", "**AVIONES**".
     - Constante para el nombre de la memoria compartida: "**AVIONESESPERA**".
     - Constantes para las clases y rutas de los aviones y pistas: "**CLASE_PISTA**", "**RUTA_PISTA**", "**CLASE_AVION**", "**RUTA_AVION**".
     - Estructura "**TProcess_t**": Esta estructura tiene dos campos, "**pid**" que es de tipo "**pid_t**" y se utiliza para almacenar el identificador del proceso, y "**clase**" que es un puntero a char y se utiliza para almacenar la clase del proceso.
2. **memoriaI.h**:
   - Contiene lo siguiente:
     - **crear_var()**: Esta función toma un nombre y un valor como argumentos, y crea un nuevo objeto de memoria compartida con ese nombre y valor. Devuelve el descriptor de archivo del objeto de memoria compartida.
     - **obtener_var()**: Esta función toma un nombre como argumento, y obtiene el descriptor de archivo asociado al objeto de memoria compartida con ese nombre.
     - **destruir_var()**: Esta función toma un nombre como argumento, y destruye el objeto de memoria compartida con ese nombre.
     - **modificar_var**(): Esta función toma un descriptor de archivo y un valor como argumentos, y modifica el valor del objeto de memoria compartida asociado a ese descriptor de archivo.
     - **consultar_var()**: Esta función toma un descriptor de archivo y un puntero a un valor como argumentos, y consulta el valor del objeto de memoria compartida asociado a ese descriptor de archivo. El valor consultado se guarda en el lugar apuntado por el puntero.
3. **semaforoI.h**:
   - Contiene lo siguiente:
     - **crear_sem()**: Esta función toma un nombre y un valor como argumentos, y crea un nuevo semáforo POSIX con ese nombre y valor. Devuelve un puntero al semáforo.
     - **get_sem()**: Esta función toma un nombre como argumento, y obtiene un semáforo POSIX ya existente con ese nombre. Devuelve un puntero al semáforo.
     - **destruir_sem()**: Esta función toma un nombre como argumento, y cierra el semáforo POSIX con ese nombre.
     - **signal_sem**(): Esta función toma un puntero a un semáforo como argumento, e incrementa el semáforo. Esto se utiliza para señalar que un recurso está disponible.
     - **wait_sem()**: Esta función toma un puntero a un semáforo como argumento, y decrementa el semáforo. Esto se utiliza para esperar hasta que un recurso esté disponible.

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** "**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.

```
[jorgelg@manjaro esqueleto_P2]$ 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/semaforoI.c -o obj/semaforoI.o
gcc -Iinclude/ -c -Wall -ggdb src/memoriaI.c -o obj/memoriaI.o
gcc -o exec/manager obj/manager.o obj/semaforoI.o obj/memoriaI.o -lpthread -lrt
gcc -Iinclude/ -c -Wall -ggdb src/pista.c -o obj/pista.o
gcc -o exec/pista obj/pista.o obj/semaforoI.o obj/memoriaI.o -lpthread -lrt
gcc -Iinclude/ -c -Wall -ggdb src/avion.c -o obj/avion.o
gcc -o exec/avion obj/avion.o obj/semaforoI.o obj/memoriaI.o -lpthread -lrt
```

A continuación, deberemos pasar por parámetros las <pistas> <aviones> necesarios para la ejecución correcta del programa.

Le vamos a pasar por ejemplo: 3 (pistas) y 10 (aviones).

Consola: $ **./exec/manager 3 10**

```
[jorgelg@manjaro esqueleto_P2]$ ./exec/manager 3 10
[MANAGER] 3 pistas creadas.
Pista [3658] en servicio...
Pista [3659] en servicio...
Pista [3660] en servicio...
Avion[3661] en camino al aeropuerto
Avion[3662] en camino al aeropuerto
Avion[3663] en camino al aeropuerto
[MANAGER] 10 aviones creados.
Avion[3664] en camino al aeropuerto
Avion[3665] en camino al aeropuerto
Avion[3666] en camino al aeropuerto
Avion[3667] en camino al aeropuerto
Avion[3668] en camino al aeropuerto
Avion[3669] en camino al aeropuerto
Avion[3670] en camino al aeropuerto
Nº de Aviones en espera: 1
Avion [3665] esperando pista libre...
Avion [3665] comenzando aterrizaje...
Pista [3658] ocupada...
Nº de Aviones en espera: 0
Nº de Aviones en espera: 1
Avion [3670] esperando pista libre...
Avion [3670] comenzando aterrizaje...
Pista [3659] ocupada...
Nº de Aviones en espera: 0
Nº de Aviones en espera: 1
Avion [3669] esperando pista libre...
Avion [3669] comenzando aterrizaje...
Pista [3660] ocupada...
Nº de Aviones en espera: 0
Nº de Aviones en espera: 1
Avion [3668] esperando pista libre...
Nº de Aviones en espera: 2
Avion [3662] esperando pista libre...
Nº de Aviones en espera: 3
Avion [3664] esperando pista libre...
Nº de Aviones en espera: 4
Avion [3663] esperando pista libre...
Nº de Aviones en espera: 5
Avion [3661] esperando pista libre...
Nº de Aviones en espera: 6
Avion [3666] esperando pista libre...
Nº de Aviones en espera: 7
Avion [3667] esperando pista libre...
Pista [3660] en servicio...
Avion [3668] comenzando aterrizaje...
Pista [3660] ocupada...
Nº de Aviones en espera: 6
Pista [3658] en servicio...
Avion [3662] comenzando aterrizaje...
Pista [3658] ocupada...
Nº de Aviones en espera: 5
Pista [3659] en servicio...
Avion [3664] comenzando aterrizaje...
Pista [3659] ocupada...
```

```
Nº de Aviones en espera: 4
Avion [3665] aparcado...
[MANAGER] Proceso AVION terminado [3665]...
Avion [3670] aparcado...
[MANAGER] Proceso AVION terminado [3670]...
Avion [3669] aparcado...
[MANAGER] Proceso AVION terminado [3669]...
Pista [3658] en servicio...
Avion [3663] comenzando aterrizaje...
Pista [3658] ocupada...
Nº de Aviones en espera: 3
Pista [3660] en servicio...
Avion [3661] comenzando aterrizaje...
Pista [3660] ocupada...
Nº de Aviones en espera: 2
Pista [3659] en servicio...
Avion [3666] comenzando aterrizaje...
Pista [3659] ocupada...
Nº de Aviones en espera: 1
Avion [3668] aparcado...
[MANAGER] Proceso AVION terminado [3668]...
Avion [3662] aparcado...
[MANAGER] Proceso AVION terminado [3662]...
Pista [3660] en servicio...
Avion [3667] comenzando aterrizaje...
Pista [3660] ocupada...
Nº de Aviones en espera: 0
Avion [3664] aparcado...
[MANAGER] Proceso AVION terminado [3664]...
Pista [3658] en servicio...
Avion [3663] aparcado...
[MANAGER] Proceso AVION terminado [3663]...
Pista [3659] en servicio...
Avion [3661] aparcado...
[MANAGER] Proceso AVION terminado [3661]...
Avion [3666] aparcado...
[MANAGER] Proceso AVION terminado [3666]...
Pista [3660] en servicio...
Avion [3667] aparcado...
[MANAGER] Proceso AVION terminado [3667]...

----- [MANAGER] Terminar con cualquier proceso pendiente ejecutándose -----
[MANAGER] Terminando proceso PISTA [3658]...
[MANAGER] Terminando proceso PISTA [3659]...
[MANAGER] Terminando proceso PISTA [3660]...

Todos los aviones en tierra y pistas cerradas

[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_P2]$ ./exec/manager 3 10
[MANAGER] 3 pistas creadas.
Pista [5145] en servicio...
Pista [5146] en servicio...
Pista [5147] en servicio...
Avion[5148] en camino al aeropuerto
Avion[5149] en camino al aeropuerto
Avion[5150] en camino al aeropuerto
[MANAGER] 10 aviones creados.
Avion[5151] en camino al aeropuerto
Avion[5152] en camino al aeropuerto
Avion[5153] en camino al aeropuerto
Avion[5154] en camino al aeropuerto
Avion[5155] en camino al aeropuerto
Avion[5156] en camino al aeropuerto
Avion[5157] en camino al aeropuerto
^C
[MANAGER] Terminacion del programa (Ctrl + C).

----- [MANAGER] Terminar con cualquier proceso pendiente ejecutándose -----
[MANAGER] Terminando proceso AVION [5148]...
[MANAGER] Terminando proceso AVION [5149]...
[MANAGER] Terminando proceso AVION [5150]...
[MANAGER] Terminando proceso AVION [5151]...
[MANAGER] Terminando proceso AVION [5152]...
[MANAGER] Terminando proceso AVION [5153]...
[MANAGER] Terminando proceso AVION [5154]...
[MANAGER] Terminando proceso AVION [5155]...
[MANAGER] Terminando proceso AVION [5156]...
[MANAGER] Terminando proceso AVION [5157]...
[MANAGER] Terminando proceso PISTA [5145]...
[MANAGER] Terminando proceso PISTA [5146]...
[MANAGER] Terminando proceso PISTA [5147]...

Todos los aviones en tierra y pistas cerradas

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