# PROGRAMACIÓN CONCURRENTE Y TIEMPO REAL
# PRÁCTICA 3

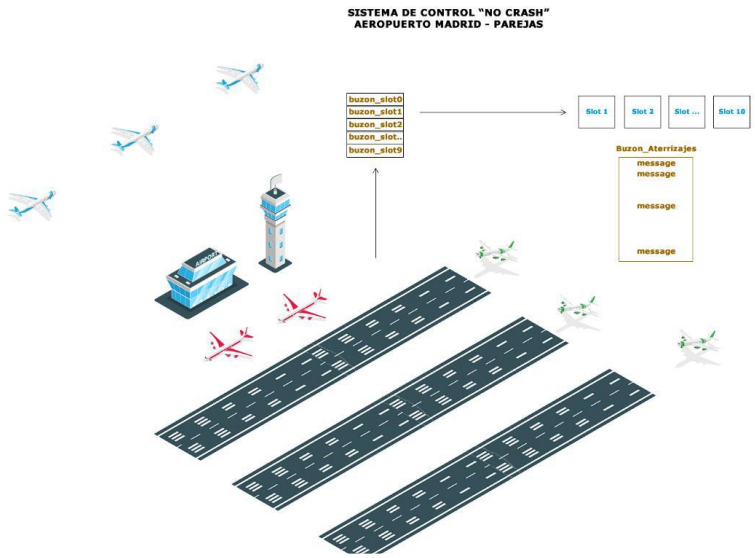
JORGE LÓPEZ GÓMEZ
OUSSAMA BOLBAROUD
UCLM

# ÍNDICE

# INTRODUCCIÓN

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

Inicialmente contamos con un número de slots y un número de pistas determinado por el usuario a través de los headers. 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 slot para aterrizar, ese slot junto con los otros n slots iniciales, han sido inicializados y están en espera a que haya alguna pista libre donde puedan aterrizar. Mientras que una pista está ocupada, el resto de slots esperan y no intentan ocupar la pista. Cuando el slot aterriza, la pista vuelve a quedar libre y podrá ser utilizada por cualquier otro slot.




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 slots 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.

# PASO DE MENSAJES

El mecanismo de paso de mensajes es una forma de sincronización entre procesos que permite la comunicación sin el uso de variables de memoria compartida. A diferencia de los semáforos, este método es ideal para entornos distribuidos, donde los sistemas tienen espacios lógicos separados. Aquí, la responsabilidad de establecer la comunicación y sincronización recae en el sistema operativo, no en el desarrollador.

El paso de mensajes se basa en dos primitivas:

- "**send**" para enviar información a un proceso.
- "**receive**" para recibir información de otro proceso.

Además, se requiere un canal de comunicación entre los procesos para garantizar la entrega y recepción de mensajes.

# 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**":

- **crear_buzones():** se encarga de crear buzones de mensajes para la comunicación entre procesos. Primero, establece las características de los buzones, especificando el número máximo de mensajes y el tamaño de los mensajes. Luego, crea un buzón llamado "**BUZON_ATERRIZAJES**". Después, en un bucle, crea "**NUMSLOTS**" buzones adicionales con nombres basados en "**BUZON_SLOTS**" y el número de iteración. Si la creación de cualquier buzón falla, imprime un mensaje de error y termina el programa.
- **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 slots que deseamos tener, el método se encarga de crear y lanzar un proceso por cada pista y un proceso por cada slot.
- **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_slot()**: es la inicialización del proceso slot, 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 es esperar a que terminen ciertos procesos. Recorre un bucle for que se ejecuta "**NUMSLOTS**" veces, y en cada iteración, llama a waitpid con el PID del proceso correspondiente, obtenido de la tabla g_process_slots_table. Esto hace que el proceso actual se pause hasta que el proceso con el PID especificado haya terminado.
- **terminar_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 slots y otra para los procesos de pistas. Finalmente, imprime un mensaje indicando que todos los slots 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 libera la memoria utilizada por las tablas de procesos g_process_pistas_table y g_process_slots_table, cierra y elimina el buzón de

aterrizajes "**BUZON_ATERRIZAJES**", y luego, en un bucle, cierra y elimina "**NUMSLOTS**" buzones de slots. Esto se hace para evitar fugas de memoria y liberar los recursos utilizados por el programa.

En la función "**main**", se va a ejecutar una secuencia lógica donde se instala el manejador de señales, se procesan las pistas y slots, se inicia la tabla de procesos, se crean los procesos y se espera a que todos terminen una vez lo solicite el usuario con el manejador de señales. 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 el funcionamiento de una pista de aterrizaje en un aeropuerto.
- Se abren colas de mensajes para aterrizajes y slots.
- En un bucle infinito, la pista espera una notificación de que está libre. Cuando recibe esta notificación, muestra un mensaje de que un avión está en aproximación desde el slot, espera durante un tiempo aleatorio entre 10 y 20 segundos, muestra un mensaje de que el avión ha aterrizado y libera el slot. Luego, envía una notificación de que la pista está libre al slot y cierra la cola de mensajes para aterrizajes.

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.

# SLOT

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

- Se definen varias variables.
- Se inicializa el generador de números aleatorios y se verifica que el programa haya recibido exactamente un argumento. Luego, se abren las colas de mensajes para aterrizajes y slots.
- En un bucle infinito, el slot espera un tiempo aleatorio, recibe un avión, envía una notificación de que el avión ha llegado, recibe una notificación de que la pista está libre y repite el proceso.

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.

# 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. El header (include) está en un archivo que se llama **definitions.h**. En el se definen las variables y constantes necesarias para el funcionamiento del programa:

1. **definitions.h**:
   - Contiene lo siguiente:
     - Constantes para clases y path: "**CLASE_PISTA**", "**RUTA_PISTA**", "**CLASE_SLOT**" y "**RUTA_SLOT**".
     - Constante para los dispositivos: "**NUMSLOTS**" y "**NUMPISTAS**".
     - Constantes para las colas de mensajes: "**BUZON_ATERRIZAJES**", "**BUZON_SLOTS**" y "**TAMANO_MENSAJES**".
     - 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.

# 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**", "**pista**" y "**slot**": 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 que hacer lo siguiente. Debemos partir 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]$ 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 -o exec/manager obj/manager.o -lpthread -lrt
gcc -Iinclude/ -c -Wall -ggdb src/pista.c -o obj/pista.o
gcc -o exec/pista obj/pista.o -lpthread -lrt
gcc -Iinclude/ -c -Wall -ggdb src/slot.c -o obj/slot.o
gcc -o exec/slot obj/slot.o -lpthread -lrt
```

A continuación, deberemos pasar por constantes definidas en la cabecera "**definitions.h**" las <pistas> <slots> necesarios para la ejecución correcta del programa.

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

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

```
[jorgelg@manjaro esqueleto]$ ./exec/manager
[MANAGER] 10 slots creados.
Slot [3013] esperando avión...
Slot [3014] esperando avión...
Slot [3015] esperando avión...
Slot [3016] esperando avión...
Slot [3017] esperando avión...
[MANAGER] 3 pistas creadas.
Slot [3018] esperando avión...
Slot [3020] esperando avión...
Slot [3019] esperando avión...
Slot [3021] esperando avión...
Slot [3022] esperando avión...
Pista [3023] en espera...
Pista [3024] en espera...
Pista [3025] en espera...
Slot [3013] recibido avión (/buzon_slot_0)...
Pista [3023] con avión en aproximación desde el Slot: /buzon_slot_0
Slot [3019] recibido avión (/buzon_slot_6)...
Pista [3024] con avión en aproximación desde el Slot: /buzon_slot_6
Slot [3017] recibido avión (/buzon_slot_4)...
Pista [3025] con avión en aproximación desde el Slot: /buzon_slot_4
Slot [3018] recibido avión (/buzon_slot_5)...
Pista [3023] ha liberado el Slot. /buzon_slot_0
Pista [3023] en espera...
Pista [3023] con avión en aproximación desde el Slot: /buzon_slot_5
Slot [3013] recibe notificación de pista libre...
Slot [3013] esperando avión...
Slot [3020] recibido avión (/buzon_slot_7)...
Pista [3024] ha liberado el Slot. /buzon_slot_6
Pista [3024] en espera...
Pista [3024] con avión en aproximación desde el Slot: /buzon_slot_7
Slot [3019] recibe notificación de pista libre...
Slot [3019] esperando avión...
Slot [3014] recibido avión (/buzon_slot_1)...
Pista [3025] ha liberado el Slot. /buzon_slot_4
Pista [3025] en espera...
Pista [3025] con avión en aproximación desde el Slot: /buzon_slot_1
Slot [3017] recibe notificación de pista libre...
Slot [3017] esperando avión...
Slot [3015] recibido avión (/buzon_slot_2)...
Slot [3021] recibido avión (/buzon_slot_8)...
Pista [3023] ha liberado el Slot. /buzon_slot_5
Pista [3023] en espera...
Pista [3023] con avión en aproximación desde el Slot: /buzon_slot_2
Slot [3018] recibe notificación de pista libre...
Slot [3018] esperando avión...
Slot [3017] recibido avión (/buzon_slot_4)...
Pista [3024] ha liberado el Slot. /buzon_slot_7
Pista [3024] en espera...
Pista [3024] con avión en aproximación desde el Slot: /buzon_slot_8
Slot [3020] recibe notificación de pista libre...
Slot [3020] esperando avión...
```

Como única alternativa hemos programado la posibilidad de terminar los procesos del programa 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.