JORGE LÓPEZ GÓMEZ OUSSAMA BOLBAROUD

UCLM

PROGRAMACIÓN CONCURRENTE Y TIEMPO REAL PRÁCTICA 2

**ÍNDICE**

[INTRODUCCIÓN 2](#_Toc163035135)

[MANAGER 3](#_Toc163035136)

[PISTA 4](#_Toc163035137)

[AVION 5](#_Toc163035138)

[SEMAFOROI 6](#_Toc163035139)

[MEMORIAI 7](#_Toc163035140)

[HEADERS 8](#_Toc163035141)

[MAKEFILE 8](#_Toc163035142)

[EJEMPLO DE EJECUCIÓN 9](#_Toc163035143)

INTRODUCCIÓN

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.

PISTA

El archivo “**pista.c**”

AVION

El archivo “**avion.c**”

SEMAFOROI

El archivo “**semaforoI.c**”

MEMORIAI

El archivo “**memoriaI.c**”

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.

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. Seguidamente se crea el ejecutable y se ejecuta.

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**.

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

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

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.