1. INTRODUCCIÓN

1.1 ESTRUCTURA DEL SISTEMA

Unix/Linux es un sistema operativo de tiempo compartido. El núcleo del sistema (kernel) es un programa que siempre está residente en memoria y, entre otros, brinda los siguientes servicios:

- Controla los recursos del hardware.
- Controla los dispositivos periféricos (discos, terminales, impresoras, etc.).
- Permite a distintos usuarios compartir recursos y ejecutar sus programas.
- Proporciona un sistema de archivos que administra el almacenamiento de información (programas, datos, documentos, etc.).

En un esquema más amplio, Unix/Linux abarca también un conjunto de programas estándar, como pueden ser:

- Compiladores de lenguajes.
- Editores de texto.
- Intérpretes de órdenes.
- Programas de gestión de archivos y directorios.

Unix/Linux puede ser dividido en tres capas:

El nivel más interno no pertenece realmente al sistema operativo, si no que es el hardware, la máquina sobre la que está implementado el sistema y cuyos recursos queremos gestionar.

Directamente en contacto con el hardware se encuentra el kernel, el cual está escrito en el lenguaje C en su mayor parte, aunque coexistiendo con código ensamblador. Las primeras implementaciones de Unix se hicieron en lenguaje ensamblador, pero en 1973 Ritchie reescribió el sistema en lenguaje C el cual se había desarrollado para recodificar Unix de una forma independiente de la máquina.

En el tercer nivel de nuestra estructura se encuentran programas estándar de cualquier sistema Unix/Linux (vi, grep, sh, who, etc.) y programas generados por el usuario. Hay que hacer notar que estos programas del tercer nivel nunca van a actuar sobre el hardware de forma directa. Para esto existe un mecanismo que nos permite indicarle al kernel que necesitamos operar sobre determinados recursos de hardware. Este mecanismo es lo que se conoce como "llamadas al sistema" (system calls) y es el objeto principal de este curso. Así pues, cualquier programa que se esté ejecutando bajo el control de Unix/Linux, cuando necesite hacer uso de alguno de los recursos que le brinda el sistema, deberá efectuar una llamada a alguna de las "system calls".

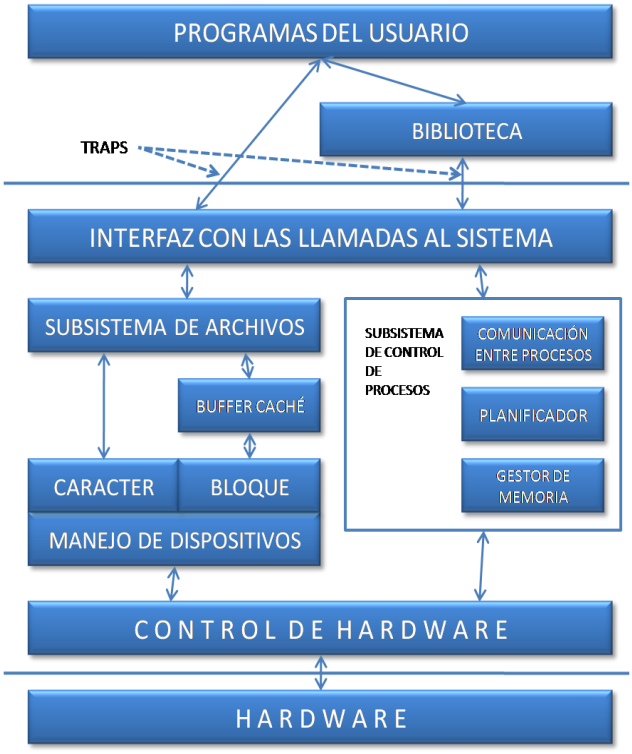
La jerarquía de programas no tiene porqué verse limitada a tres niveles. El usuario puede crear tantos niveles como necesite. Puede haber también programas que se apoyen en diferentes niveles y que se comuniquen con el kernel por un lado, y con otros programas existentes por otro.

1.2 ARQUITECTURA DEL SISTEMA OPERATIVO UNIX/LINUX

A continuación esbozaremos, desde un punto vista muy general, los bloques funcionales básicos de que consta el núcleo de Unix/Linux.

Los dos conceptos centrales sobre los que se basa la arquitectura de Unix/Linux son los archivos y los procesos. El kernel está pensando para facilitarnos servicios relacionados con el sistema de archivos y con el control de procesos.

La siguiente figura muestra los tres niveles que vamos a estudiar en la arquitectura del sistema: hardware, kernel y usuario.



Las llamadas al sistema y su biblioteca asociada representan la frontera entre los programas del usuario y el kernel. La biblioteca asociada a las llamadas es el mecanismo mediante el cual podemos invocar una llamada desde un programa C/C++. Esta biblioteca se lanza por defecto al compilar cualquier programa C/C++ y se encuentra en el archivo `/usr/lib/libc.a`. Los programas escritos en lenguaje ensamblador pueden invocar directamente a las llamadas al sistema sin necesidad de ninguna biblioteca intermedia.

Las llamadas al sistema se ejecutan en modo kernel (o modo supervisor) y para entrar en este modo hay que ejecutar una sentencia en código máquina como trap (o interrupción de software). Es por esto que las llamadas al sistema pueden ser invocadas directamente desde ensamblador y no desde C/C++.

En la figura anterior podemos apreciar que el núcleo está dividido en dos subsistemas principales: subsistema de archivos y subsistema de control de procesos.

El subsistema de archivos controla los recursos del sistema de archivos y tiene funciones como reservar el espacio para los archivos, administrar el espacio libre, controlar el acceso a los archivos, permitir el intercambio de datos entre los archivos y el usuario, etc. Los procesos interactúan con el subsistema de archivos a través de unas llamadas específicas.

El subsistema de archivos se comunica con los dispositivos de almacenamiento secundario (discos duros, unidades de cinta, etc.) a través de los manejadores de dispositivo (device drivers). Los manejadores de dispositivo se encargan de proporcionar el protocolo de comunicación (handshake) entre el núcleo y los periféricos. Se consideran dos tipos de dispositivos según la forma de acceso: dispositivos modo bloque (block devices) y dispositivos modo carácter (row devices). El acceso a los dispositivos en modo bloque se lleva a cabo con la intervención de buffers que mejoran enormemente la velocidad de transferencia. El acceso a dispositivos en modo carácter se lleva a cabo de forma directa, sin la intervención de buffers. Un mismo dispositivo físico puede ser manejado tanto en modo bloque como en modo carácter, dependiendo de qué manejador usemos para acceder a él.

El subsistema de control de procesos es el responsable de la planificación de los procesos (scheduler), su sincronización, comunicación entre los mismos (IPC – Inter Process Communication) y del control de la memoria principal.

El módulo de gestión de memoria se encarga de controlar qué procesos están cargados en la memoria principal en todo momento. Si en un momento determinado no hay suficiente memoria principal para todos los procesos que lo solicitan, el gestor de memoria debe recurrir a mecanismos de intercambio (swapping) para que todos los procesos tengan derecho a un tiempo mínimo de ocupación de la memoria y se puedan ejecutar.

El intercambio consiste en llevar los procesos cuyo tiempo de ocupación de la memoria expira a una memoria secundaria (que generalmente es el área que se dedica a intercambio en el disco – área de swap- y que se monta como un sistema de archivos aparte) y traer de esa memoria secundaria los procesos a los que se les asigna tiempo de ocupación de la memoria principal. Al módulo gestor de memoria se le conoce también como intercambiador (swapper).

El planificador o scheduler se encarga de gestionar el tiempo de CPU que tiene asignado cada proceso. El scheduler entra en ejecución cada cierto tiempo y decide si el proceso actual tiene derecho a seguir ejecutándose (esto depende de su prioridad y de sus privilegios) o ha de cambiarse de contexto (asignarle el CPU a otro proceso).

La comunicación entre procesos puede realizarse de forma asíncrona (señales) o síncrona (colas de mensaje, semáforos).

Por último, el control del hardware es la parte del núcleo encargada del manejo de las interrupciones y de la comunicación con la máquina. Los dispositivos pueden interrumpir al CPU mientras se está ejecutando un proceso. Si esto ocurre, el núcleo debe reanudar la ejecución del proceso después de atender a la

interrupción. Las interrupciones no son atendidas por procesos, sino por funciones especiales, codificadas en el núcleo, que son invocadas durante la ejecución de cualquier proceso.

2. ARQUITECTURA DEL SISTEMA DE ARCHIVOS

2.1 CARACTERÍSTICAS DEL SISTEMA DE ARCHIVOS

El sistema de archivos de Unix/Linux se caracteriza por:

- Poseer una estructura jerárquica.
- Realizar un tratamiento consistente de los datos de los archivos.
- Poder crear y borrar archivos.
- Permitir un crecimiento dinámico de los archivos.
- Tratar los dispositivos y periféricos (terminales, unidades de disco, etc.) como si fueran archivos.

El sistema de archivos está organizado, a nivel lógico, en forma de árbol, con un nodo principal conocido como raíz ("/"). Cada nodo dentro del árbol es un directorio que puede contener a su vez otros nodos (subdirectorios), archivos normales o archivos de dispositivos.

Los programas que se ejecutan en Unix/Linux no conocen el formato interno con el que el kernel almacena los datos. Cuando accedemos al contenido de un archivo mediante una llamada al sistema (read), el sistema nos lo va a presentar cómo una secuencia de bytes sin formato. Nuestro programa será el encargado de interpretar la secuencia de bytes y darle un significado según sus necesidades. Por lo tanto, la sintaxis (forma) del acceso a los datos de un archivo viene impuesta por el sistema y es la misma para todos los programas, y la semántica (significado) de los datos es responsabilidad del programa que trabaja con el archivo.

2.2 ESTRUCTURA DEL SISTEMA DE ARCHIVOS

Los sistemas de archivos suelen estar situados en dispositivos de almacenamiento modo bloque, tales como cintas o discos.

Las cintas tienen un tiempo de acceso mucho más alto que los discos, por ello, es poco práctico instalar un sistema de archivos sobre ellas. Sin embargo, son muy útiles para realizar copias de seguridad (backup) de un sistema ya instalado.

Lo normal es que un sistema Unix/Linux se arranque (boot) desde cinta cuando, tras producirse una falla (crash) del sistema, queremos restaurarlo a su situación anterior (proceso conocido como recovery system).

Un sistema Unix/Linux puede manejar uno o varios discos físicos, cada uno de los cuales puede contener uno o varios sistemas de archivos. Los sistemas de archivos son particiones lógicas del disco.

Hacer que un disco físico contenga varios sistemas de archivos permite una administración más segura, ya que si uno de los sistemas de archivos se daña, perdiéndose la información que hay en él, este accidente no se habrá propagado al resto de los sistemas de archivos que hay en el disco y podremos seguir trabajando con ellos para intentar una restauración o una reinstalación.

El kernel trabaja con el sistema de archivos a un nivel lógico y no trata directamente con los discos a nivel físico. Cada disco es considerado como un dispositivo lógico que tiene asociados unos números de dispositivos. Estos números se utilizan para indexar dentro de una tabla de funciones, la cual tenemos que emplear para acceder al manejador del disco. El manejador del disco se va a encargar de transformar las direcciones lógicas (kernel) de nuestro sistema de archivos a direcciones físicas del disco.

Un sistema de archivos se compone de una secuencia de bloques lógicos, cada uno de los cuales tiene un tamaño fijo. El tamaño de cada bloque es el mismo para todo el sistema de archivos y suele ser múltiplo de 512 bytes. Existen diferentes tipos de sistemas de archivos, cada uno de ellos ideado para una tarea específica y con un diferente tamaño de bloque.

El tamaño elegido para el bloque va a influir en las prestaciones globales del sistema. Por un lado nos interesa que los bloques sean grandes para que la velocidad de transferencia entre el disco y la memoria sea grande. Sin embargo, si los bloques lógicos son demasiados grandes, la capacidad de almacenamiento del disco se puede ver desaprovechada cuando abundan archivos pequeños que no llegan a ocupar un bloque completo. Valores típicos, en bytes, para el tamaño de un bloque son: 512, 1024, 2048.

La estructura de un sistema de archivos en Unix/Linux es:

En la figura podemos ver cuatro partes:

- **El sector de arranque** (MBR) ocupa la parte del principio del sistema de archivos, típicamente el primer sector, y puede contener el código de arranque. Este código es un pequeño programa (GRUB, NTLOADER, LILO) que se encarga de buscar el sistema operativo y cargarlo en memoria para inicializarlo.

- **El superbloque** describe el estado de un sistema de archivos. Contiene información acerca de su tamaño, total de archivos que puede contener, qué espacio queda libre, etc.

- **La lista de nodos índice (nodos-i)**. Se encuentra a continuación del superbloque. Esta lista tiene una entrada por cada archivo, donde se guarda una descripción del mismo: situación del archivo en el disco, propietario, permisos de acceso, fecha de actualización, etc. En algunas versiones de Unix/Linux es posible determinar el tamaño de la lista de nodos-i al momento de configurar el sistema.

- **Los bloques de datos** empiezan a continuación de la lista de nodos-i y ocupan el resto del sistema de archivos. En esta zona es donde se encuentra situado el contenido de los archivos a los que hace

referencia la lista de nodos-i. Cada uno de los bloques destinados a datos sólo puede ser asignado a un archivo, tanto si lo ocupa completamente o no.

2.3 SUPERBLOQUE

Como antes se mencionó, el superbloque contiene la descripción del estado del sistema de archivo. En el archivo cabecera `<sys/filsys.h>` está declarada una estructura en C que describe el significado del contenido del superbloque.

El superbloque contiene, entre otros elementos, la siguiente información:

- Tamaño del sistema de archivos.
- Lista de bloques libres disponibles.
- Índice del siguiente bloque libre en la lista de bloques libres.
- Tamaño de la lista de nodos-i.
- Total de nodos-i libres.
- Lista de nodos-i libres.
- Índice del siguiente nodo-i libre en la lista de nodos-i libres.
- Campos de bloqueo de elementos de la lista de bloques y de nodos-i libres. Estos campos se emplean cuando se realiza una petición de bloque o de nodo-i libre.
- Indicador que informa si el superbloque ha sido modificado o no.

Cada vez que, desde un proceso, se accede a un archivo, es necesario consultar el superbloque y la lista de nodos-i. Como el acceso a disco suele degradar bastante el tiempo de ejecución de un programa, lo normal es que el kernel realice la E/S con el disco a través de un buffer cache y que el sistema tenga en memoria una copia del superbloque y de las listas de nodos.-i. Esto plantea un problema de consistencia de los datos, ya que una actualización en memoria del superbloque y de la tabla de nodos-i no implica una actualización inmediata en disco. La solución a este problema consiste en realizar periódicamente una actualización en disco de los datos de administración que mantenemos en memoria. De esta tarea se encarga un “daemon” que se arranca al inicializar el sistema. Naturalmente, antes de apagar el sistema hay que actualizar al superbloque y las tablas de nodos-i del disco. El programa `shutdown` se encarga de esta tarea y es fundamental tener presente que no se debe de realizar una salida del sistema sin haber invocado previamente a este programa, porque de lo contrario el sistema de archivos podría quedar seriamente dañado.

2.4 NODOS-I (INODES)

Cada archivo en un sistema Unix/Linux tiene asociado un nodo-i. El nodo-i contiene la información necesaria para que un proceso pueda acceder al archivo. Esta información incluye: propietario, derechos de acceso, tamaño, localización en el sistema de archivos, etc.

La lista de nodos-i se encuentra situada en los bloques que hay a continuación del superbloque. Durante el proceso de arranque del sistema, el núcleo lee la lista de nodos-i del disco y carga una copia en memoria, conocido como tabla de nodos-i. Las manipulaciones que haga el subsistema de archivos (parte del código del kernel) sobre los archivos van a involucrar a la tabla de nodos-i pero no la lista de nodos-i. Mediante este mecanismo se consigue una mayor velocidad de acceso a los archivos, ya que la tabla de nodos-i está cargada siempre en memoria. En el punto anterior vimos que existe un proceso del sistema (`syncer`) que se encarga de actualizar periódicamente el contenido de la lista de nodos-i con la tabla de nodos-i.

En el archivo cabecera `<sys/ino.h>` está declarada una estructura en C que describe la información de un nodo-i. Los campos que la integran son:

- Identificador del propietario del archivo. La posesión se divide entre un propietario individual y un grupo de propietarios y define el conjunto de usuarios que tiene derecho de acceso al archivo. El superusuario tiene derecho de acceso a todos los archivos del sistema.

- Tipo de archivo. Los archivos pueden ser de datos, directorios, especiales de dispositivos (en modo carácter o en modo bloque) y tuberías (FIFO).

- Tipo de acceso al archivo. El sistema protege los archivos estableciendo tres niveles de permisos: permisos del propietario, del grupo al que pertenece el propietario y del resto de los usuarios (conocidos también como el mundo). Cada clase de usuarios puede tener habilitados o deshabilitados los derechos de lectura, escritura y ejecución. Para los directorios el derecho de ejecución significa poder acceder o no a los archivos que contiene.

- Registro de acceso al archivo. Dan información sobre la fecha de la última modificación del archivo, la última vez que se accedió a él y la última de vez que se modificaron los datos de su nodo-i.

- Número de enlaces del archivo. Representa el total de los nombres que el archivo tiene en la jerarquía de directorios. Como veremos más adelante, un archivo puede tener asociados diferentes nombres que correspondan a diferentes rutas, pero a través de los cuales accedamos a un mismo nodo-i y, por consiguiente, a los mismos bloques de datos.

- Entradas para los bloques de dirección de los datos de un archivo. Si bien los usuarios tratan los datos de un archivo como si fuesen una secuencia de byte contiguos, el kernel puede almacenarlos en bloques que no tiene por qué ser contiguos. En los bloques de dirección es donde se especifican los bloques de disco que contienen los datos del archivo.

- Tamaño del archivo. Los bytes de un archivo se pueden direccionar indicando un desplazamiento a partir de la dirección de inicio del archivo (desplazamiento 0). El tamaño del archivo es igual al desplazamiento del byte más alto incrementado en uno. Por ejemplo, si un usuario crea un archivo y escribe en él sólo un byte en la posición 2000, el tamaño del archivo es 2001 bytes.

Hay que hacer notar que el nombre del archivo no queda especificado en el nodo-i. Es en los archivos de tipo directorio donde a cada nombre del archivo se le asocia su nodo-i correspondiente.

También hay que señalar que hay una diferencia entre escribir el contenido de un nodo-i en disco y escribir el contenido de un archivo. El contenido del archivo (sus datos) cambia sólo cuando se escribe en él. El contenido de un nodo-i cambia cuando se modifican los datos del archivo o la situación administrativa del mismo (propietarios, permisos, enlaces, etc.).

La tabla de nodos-i contiene la misma información que la lista de nodos-i, además de lo siguiente:

- El estado del nodo-i, que indica:
    - Si el nodo-i está bloqueado,
    - Si hay algún proceso esperando a que el nodo-i quede desbloqueado,
    - Si la copia del nodo-i que hay en memoria difiere de la que hay en el disco,
    - Si la copia de los datos del archivo que hay en memoria difiere de los datos que hay en el disco (caso de la escritura en el archivo a través del buffer caché).
- El número de dispositivo lógico del sistema de archivos en donde se almacena.
- El número de nodo-i. Como los nodos-i se almacenan en el disco en un arreglo lineal, al cargarlo en memoria, el kernel le asigna un número en función de su posición en el arreglo. El nodo-i del disco no necesita esta información.
- Punteros a otros nodos-i cargados en memoria. El kernel enlaza los nodos-i sobre una cola de dispersión (cola hash) y sobre una lista libre. Las claves de acceso a la cola de dispersión nos las dan el número de dispositivo lógico del nodo-i y el número de nodo-i.
- Un contador que indica el número de copias del nodo-i que están activas (por ejemplo, porque el archivo está abierto por varios procesos).

2.5 BLOQUES DE DATOS

Los bloques de datos están situados a partir de la lista de nodos-i. Como se mencionó antes, cada nodo-i tiene unas entradas (bloques de direcciones) para localizar dónde están los datos de un archivo en el disco. Como cada bloque del disco tiene asociada una dirección, las entradas de direcciones consisten en un conjunto de direcciones de bloques del disco.

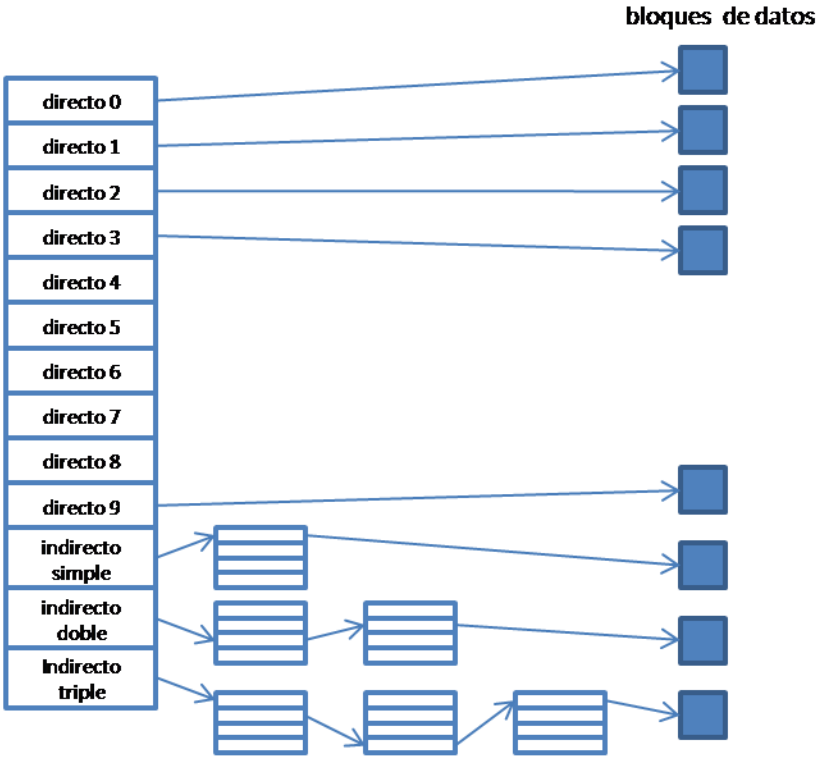
Si los datos de un archivo se almacenasen en bloques consecutivos, para poder acceder a ellos nos bastaría con conocer la dirección del bloque inicial y el tamaño del archivo. Sin embargo, esta estrategia provocaría un desaprovechamiento del disco, ya que tenderían a proliferar áreas libres demasiado pequeñas para ser usadas. Por ejemplo, supongamos que un usuario crea tres archivos, A, B y C, cada uno de los cuales ocupa

10 bloques en el disco. Supongamos que el sistema sitúa estos tres archivos en bloques contiguos. Si el usuario necesita añadir 5 bloques al archivo B, el sistema va a tener que buscar 15 bloques contiguos que se encuentren libres para copiar el archivo B en esa zona. Esto presenta dos inconvenientes, el primero es el tiempo que se pierde al copiar los 10 bloques de B que no cambiarían en contenido, el segundo es que los bloques que quedan libres sólo pueden contener archivos con un tamaño inferior o igual a 10 bloques. Esto va a provocar, a la larga, una microfragmentación del disco que lo va a dejar inservible.

El kernel puede minimizar la fragmentación del disco ejecutando periódicamente procesos para compactarlos, pero esto produce una degradación de las prestaciones del sistema en cuanto a velocidad. Por esto y para una mayor velocidad, el kernel reserva los bloques para un archivo de uno en uno, como se vayan requiriendo, y permite que los datos de un archivo estén esparcidos por todo el sistema de archivos.

Las entradas de direcciones de un nodo-i van a consistir en una lista de direcciones de los bloques que tienen los datos del archivo. Un cálculo simple nos muestra que almacenar la lista de bloques del archivo en un nodo-i es algo difícil de implementar. Por ejemplo, si los bloques son de 1 Kbyte y el archivo ocupa 10 Kbytes (10 bloques), vamos a necesitar almacenar 10 direcciones en el nodo-i. Pero si el archivo es de 100 Kbytes (100 bloques), necesitaremos 100 direcciones para acceder a todos los datos. Así pues, esta estrategia nos impone que el tamaño del nodo-i debe ser variable, ya que si lo fijamos a un valor concreto, estamos fijando el tamaño máximo de los archivos que podemos manejar. Debido a lo poco manejable que resulta, la idea de nodos-i de tamaño variable es algo que no se implementa en ningún sistema.

Para conseguir que el tamaño de un nodo-i sea pequeño y a la vez podamos manejar archivos grandes, las entradas de direcciones de un nodo-i se ajustan al siguiente esquema:



Los nodos-i tienen 13 entradas. Las entradas marcadas como directas, en la figura, contienen direcciones de bloques en los que hay datos del archivo. La entrada marcada como indirecta simple direcciona un bloque de datos que contiene una tabla de direcciones de bloques de datos. Para acceder a los datos a través de una entrada indirecta, el kernel debe leer el bloque cuya dirección nos indica la entrada indirecta y buscar en él la dirección del bloque donde realmente está el dato, para a continuación leer ese bloque y acceder al dato. La entrada marcada como indirecta doble contiene la dirección de un bloque cuyas entradas actúan como entradas indirectas simples y la entrada indirecta triple direcciona un bloque cuyas entradas son indirectas dobles.

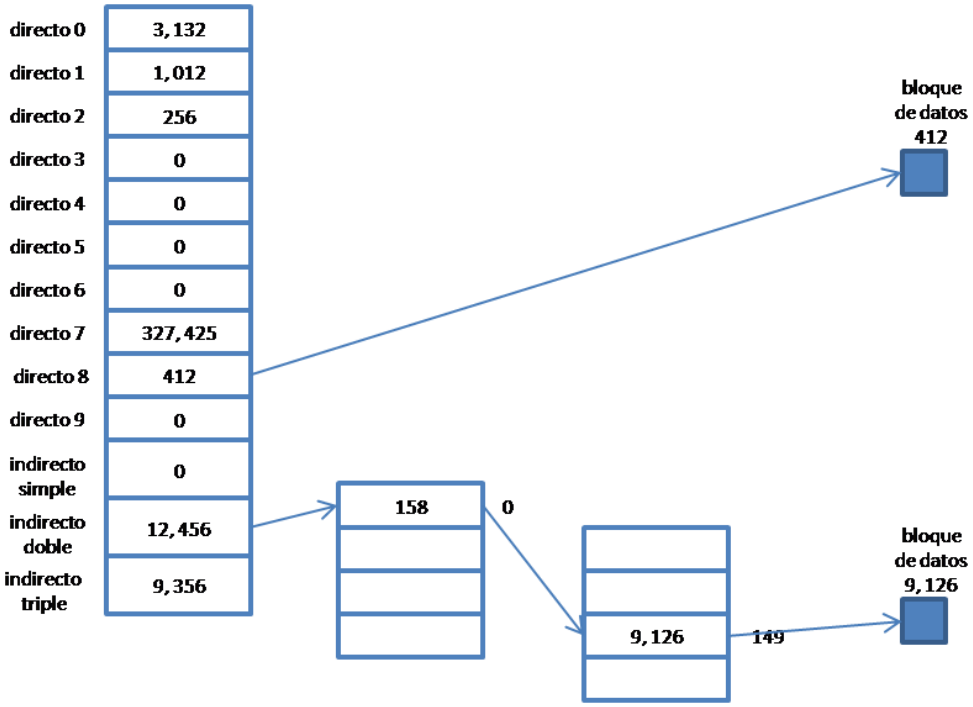
Este método puede extenderse para soportar entradas indirectas cuádruples, quíntuples, etc., pero en práctica tenemos más que suficiente con una indirecta triple.

Vamos a ver el caso práctico en el que los bloques son de 1 Kbyte y el bloque se direcciona con 32 bits (2^{32} = 4 Gdirecciones posibles). En esta situación, un bloque de datos puede almacenar 256 direcciones de bloques y un archivo podría llegar a tener un tamaño de 16 Gbytes.

Tipo de Entrada	Total de bloques accesibles	Total de bytes accesibles
10 entradas directas	10 bloques de datos	10 Kbytes
1 entrada indirecta simple	1 bloque indirecto simple -> 256 bloques directos	256 Kbytes
1 entrada indirecta doble	1 entrada indirecta doble -> 256 bloques indirectos simples -> 65, 536 bloques directos	64 Mbytes
1 entrada indirecta triple	1 entrada indirecta triple -> 256 bloques indirectos dobles -> 65, 536 bloques indirectos simples -> 16, 777, 216 bloques directos	16 Gbytes

Los procesos acceden a los datos de un archivo indicando la posición, con respecto al inicio del archivo, del byte que queremos leer o escribir. El archivo es visto como una secuencia de bytes que empieza en el byte número 0 y llega hasta el byte cuya posición, con respecto al inicial, coincide con el tamaño del archivo menos uno. El kernel se encarga de transformar las posiciones de los bytes, tal y como las ve el usuario, a direcciones de los bloques de disco.

Veamos un ejemplo. Supongamos un archivo cuyos datos están en los bloques que nos indican las entradas de direcciones del nodo-i descrito en la siguiente figura:



Vamos a seguir suponiendo que el bloque tiene un tamaño de 1024 bytes. Si un proceso quiere acceder a un byte que se encuentra en la posición 9, 125 del archivo, el kernel calcula que ese byte está en el número 8 del archivo (empezando a numerar los bloques lógicos desde 0).

En efecto:

$$bloque_{archivo} = \frac{desplazamiento_{archivo}}{tamaño_{bloque}} = \frac{9{,}125\,bytes}{1{,}024\,bytesxbloque} = 8$$

Para ver a qué bloque del disco corresponde el bloque lógico número 8 del archivo, hay que consultar el número de bloque almacenado en la entrada número 8 (entrada de dirección directa) de la tabla de direcciones del nodo-i. En el caso de la figura, el bloque de disco buscado es el 412. Dentro de este bloque, el byte 9,125 del archivo corresponde con el byte 933 con respecto al inicio del bloque (los bytes del bloque se numeran de 0 a 1023).

En efecto:

$$desplazamiento_{bloque} = desplazamiento_{archivo} \, mod \, tamaño_{bloque}$$
$$= 9{,}125 \, mod \, 1{,}024\,bytesxbloque = 933$$

En este ejemplo el cálculo ha sido sencillo porque el byte buscado era accesible desde una entrada directa nodo-i.

Vamos a ver qué ocurre si queremos localizar en el disco el byte que se encuentra en la posición 425,000 del archivo. Si calculamos su bloque lógico de archivo, veremos que se encuentra en el bloque 415.

$$bloque_{archivo} = \frac{desplazamiento_{archivo}}{tamaño_{bloque}} = \frac{425{,}000\,bytes}{1{,}024\,bytesxbloque} = 415$$

El total de bloques que podemos acceder con las entradas directas es de 10. Con la entrada indirecta simple podemos acceder a 256 direcciones, y con la indirecta doble, a 65,536. Por lo que el byte buscado estará en una entrada indirecta doble.

A esta entrada pertenecen los bloques comprendidos entre los números lógicos de archivo 266 y 65,581 (incluyéndolos) y el bloque buscado es el 415. Si nos fijamos en la figura, el número de bloque que contiene la entrada indirecta doble es el 12,456, que es el bloque de disco donde están las direcciones de los bloques con entradas indirectas simples.

La entrada número 0 del bloque indirecto doble nos da acceso a los bloques de archivo comprendidos entre el 266 y el 521, ya que cada entrada actúa como indirecto simple y da acceso a 256 bloques de datos. En la entrada 0 vemos que el número de bloque de disco del bloque indirecto simple que buscamos es 158. Dentro del bloque indirecto simple, la entrada que nos interesa es la diferencia entre 415 (bloque lógico del archivo) y 266 (bloque inicial al que da acceso el indirecto doble) que es 149. Según la figura, la entrada 149 del bloque de disco 158 contiene el número 9,126. Es en el bloque de disco 9,126 donde se encuentra el dato que buscamos, y en el byte 40 de este bloque está el byte 425,00 de nuestro archivo.

$$desplazamiento_{bloque} = desplazamiento_{archivo} mod tamaño_{bloque}$$
$$= 425{,}000\,mod\,1{,}024\,bytesxbloque = 40$$

Si observamos la figura anterior con más detenimiento, vemos que hay algunas entradas del nodo-i que están en 0. Esto significa que no referencian a ningún bloque del disco y que los bloques lógicos correspondientes del archivo no tienen datos. Esta situación se da cuando se crea un archivo y nadie escribe en los bytes correspondientes a estos bloques, por lo que permanecen en su valor inicial 0. Al no reservar el sistema bloques de discos para estos bloques lógicos, se consigue ahorro de los recursos del disco. Imaginemos que creamos un archivo y sólo escribimos un byte en la posición 1,048,276, esto significa que el archivo tiene un tamaño de 1 Mbyte. Si el sistema reservase bloques de discos para este archivo en función de su tamaño y no en función de los bloques lógicos que realmente tiene ocupados, nuestro archivo ocuparía 1024 bloques de discos en lugar de 1, como en realidad ocupa.

3. USANDO EL GNU CC

3.1 CARACTERÍSTICAS DEL GNU CC

gcc es uno de los compiladores mas usados en los ambientes de programa Unix/Linux. Es un compilador muy versátil y flexible. El compilador gcc nos permite:

- Detener el proceso de compilación en cualquier etapa y examinar la salida que nos da el compilador en cada una de ellas.

- Es capaz de manejar varios dialectos de C, como ANSI C. Así como compilar programas en C++ y Objective C.
- Se puede controlar la cantidad y tipo de información de "debugging" que estará embebida en el archivo binario resultante.
- Realizar optimización del código generado.

3.2 UN BREVE TUTORIAL

Antes de ver a profundidad al gcc, veremos un pequeño ejemplo que nos ayudará para empezar a usarlo. Hagamos un "hola mundo".

Archivo: programa1.c

```
#include <stdio.h>


int main() {


      fprintf(stdout, "Hola Mundo!!!\n");


}
```

Para compilar y correr este programa, debemos ejecutar las siguientes instrucciones:

```
$ gcc programa1.c –o programa

$ ./programa1

$Hola Mundo!!!

$
```

El primer comando le indica a gcc que compile y ligue el archivo fuente programa1.c, creando un archivo ejecutable, especificado con el argumento -o, llamada programa1. El segundo comando ejecuta el programa, resultando en la salida que nos muestra la tercera línea.

Muchas cosas suceden debajo de estas instrucciones. Primero, gcc pasa el archivo programa1.c a través de un preprocesador, cpp, que expande cualquier macro e inserta los contenidos de los archivos cabecera. El siguiente paso es compilar el código resultante del preproceso y así generar un código objeto. Y, finalmente, el enlazador (ligador, linker), ld, se encargará de crear el archivo binario, programa1.

Podemos recrear cada paso manualmente, pasando a través de todo el proceso de compilación. Para indicarle a gcc que se detenga después del preproceso, usaremos la opción -E.

```
$ gcc -E programa1.c -o programa1.cpp
```

Si examinamos el archivo programa1.cpp, podremos ver el contenido que se ha agregado a causa del archivo stdio.h, junto con otros elementos del preproceso. El siguiente paso es convertir el archivo programa1.cpp en código objeto (proceso de compilación). Usamos gcc con la opción de -c para lograr esto:

```
$ gcc -x cpp-output -c programa1.cpp -o programa1.o
```

En esta sentencia se usa la opción -x para indicarle a gcc que empiece la compilación en el paso indicado, en este caso, con el código generado en el preproceso.

Por último el proceso de enlace (linking) con el fin de crear el archivo ejecutable (binario):

```
$ gcc programa1.o -o programa1
```

La mayoría de los programas en C/C++ consisten de múltiples archivos fuentes. Cuando sucede esto, es necesario que cada archivo sea convertido a archivo objeto antes del proceso final de enlace. Este requerimiento es fácil de cumplir. Supongamos, por ejemplo, que estamos trabajando en el programa killerapp.c, que usa el código del archivo helper.c. Para compilar esta aplicación, deberíamos usar el siguiente comando:

```
$ gcc killerapp.c helper.c -o killerapp
```

De esta forma gcc recorre el mismo proceso de generación (preproceso-compilación-enlace), creando antes los archivos objetos de cada archivo fuente.

3.2.1 ARCHIVO CABECERA Y LIBRERÍAS

Si en nuestro proyecto usamos librerías o archivos cabecera que no están en localizaciones estándar, las opciones -L{DIRNAME} y -I{DIRNAME) nos permiten especificar esas localidades y asegurar que se busque ahí antes de hacerlo en los directorios estándar. Por ejemplo, si almacenamos los archivos cabecera en la dirección `/usr/local/include/killerpapp`, entonces la instrucción que debemos utilizar sería:

```
$ gcc someapp.c -I/usr/local/include/ -o killerapp
```

De manera similar, supongamos que estamos probando una nueva librería de programa, `libnew.so` (.so es la extensión por convención que tienen todas las librerías compartidas), que temporalmente se encuentra guardada en el directorio /home/foo/lib. Supongan, además, que los archivos cabecera se encuentran almacenados en `/home/foo/include`, entonces la instrucción para gcc sería:

```
$ gcc myapp.c -L/home/foo(lib -I/home/foo/include -lnew
```

La opción `-l` le indica al enlazador (linker) que tome el código objeto de la librería especificada. En este ejemplo, queremos que el enlace se haga con respecto a `libnew.so`. Una convención largamente aceptada en Unix es que las librerías son llamadas `lib{algo}.so` por lo que gcc, como la mayoría de los compiladores, confía en esta convención. Si nos equivocamos (u omitimos) el uso de la opción `-l` cuando estemos enlazando librerías, el paso de enlace presentará errores de referencias no definidas.

Por omisión gcc utiliza librerías compartidas, pero si queremos usar librerías estáticas, debemos usar la opción `-static`.

```
$ gcc curseapp.c -lncurses -static
```

El uso de librerías estáticas nos da como resultado archivos binarios más grandes que cuando usamos librerías compartidas. Entonces, ¿cuándo es conveniente usar librerías estáticas? Si usamos librerías compartidas y necesitamos ejecutar el código del programa, es hasta ese momento que las librerías son enlazadas. Eso no sucede con las librerías estáticas, puesto que el enlace se realizó en el momento de compilación. Por eso cuando queremos garantizar que los usuarios puedan ejecutar un programa siempre, no importa la instalación que tenga el sistema, lo recomendable es compilar usando librerías estándar.

3.2.2 OPCIONES DE OPTIMIZACIÓN

La optimización de código es una forma de mejorar el rendimiento de un programa. Su principal desventaja es que el proceso de compilación toma más tiempo e incrementa la memoria usada.

La opción `-O` indica a gcc que debe reducir tanto el código como el tiempo de ejecución. Equivale a usar el parámetro `-O1`. El tipo de optimización realizada, en este nivel, depende del procesador destino, pero siempre incluye un adecuado uso de saltos y uso de stack.

El nivel de optimización `-O2` incluye el primer nivel de optimización agregando trucos que modifican la calendarización de instrucciones en el procesador. En este nivel, el compilador se asegura de que el procesador tenga instrucciones que ejecutar mientras está esperando el resultado de otras o por la latencia de la memoria principal o el cache. Esta opción es todavía más dependiente del procesador que la anterior. El nivel `-O3` incluye lo anterior más otras características específicas del procesador.

Por lo general, el nivel `-O2` es suficiente.

3.2.3 OPCIONES DE "DEBUGGING"

La opción `-g` tiene tres niveles (1,2 y 3) que nos permiten especificar la cantidad de información que se incluirá en el archivo binario. El nivel 2, que es el de omisión (`-g2`), incluye una extensa tabla de símbolos, números de líneas e información acerca de las variables locales y globales. El nivel 3 incluye, además, las definiciones de todas las macros existentes. El nivel 1 genera solo la información relacionada con el manejo de stack, no incluye ninguna información relacionada con número de línea o variables.

4. USANDO EL GNU MAKE

4.1 DISTRIBUCIÓN DE ARCHIVOS EN UN PROYECTO

La gestión inadecuada de un proyecto de programa grande puede obstaculizar su adecuado desarrollo. Aunque no existen unos criterios formalizados que permitan unificar la forma de enfocar y desarrollar una aplicación, sí podemos plantear una serie de consejos, fruto de experiencias, que nos pueden ayudar y facilitar la codificación de un programa de envergadura.

Los puntos que se enumeran a continuación van a ser relevantes cuando estemos desarrollando una aplicación en lenguaje C/C++ y en un sistema operativo Unix/Linux:

1. Dedicar un directorio o subdirectorio para almacenar los archivos que componen la aplicación.
2. A la hora de nombrar los distintos archivos y directorios conviene utilizar nombres significativos. Aunque no hay una norma al respecto algunos de los nombres más usados son los siguientes:
    - src: Directorio donde residen los archivos en código fuente (.c, .cc, etc.).
    - include: Directorio donde residen los archivos cabecera (.h).
    - lib: Directorio donde residen las bibliotecas (.a).
    - bin: Directorio donde residen los programas ejecutables que se han obtenido como resultado de la compilación y enlace.
    - doc: Directorio donde residen la documentación externa que acompaña a la aplicación. Por ejemplo: manual de usuario, manual de administrador, etc.
3. Dividir el programa fuente en módulos que agrupen funciones y datos que tengan alguna relación semántica. Cuanto más minucioso sea el análisis previo, mayor el nivel de detalle que se puede alcanzar en esta división modular.

Con respecto a este punto conviene tener en cuenta los siguientes apartados:

- Las partes del programa que no sean transportables de un ordenador a otro deben estar perfectamente localizadas y aisladas del resto del programa.

  Las incompatibilidades pueden deberse a diferentes causas:

  - El programa accede a recursos de hardware mediante procedimientos no estándar. Por ejemplo, acceso directo a la tarjeta de vídeo, reprogramación directa de dispositivos de entrada/salida, etc.
  - El programa accede a recursos de software mediante procedimientos no estándar. Por ejemplo, acceso a estructuras de control del sistema operativo, uso de llamadas al sistema que no son compatibles, etc.

Hay que decir que utilizando la compilación condicional se pueden subsanar, de una forma elegante, gran parte de los problemas que impiden la transportabilidad del código fuente. Para ello hay que conocer cuáles son los sistemas donde se va a compilar la aplicación y utilizar directrices del preprocesador para incluir unas secciones del código u otras.

- Las partes del programa destinadas a la definición de los datos deben formar parte de un archivo de cabecera que se incluirá en los módulos que necesiten conocer esas estructuras de datos.

Este planteamiento evitará la existencia de fuentes de errores tales como:

- Definiciones redundantes de una misma estructura de datos.
- Inconsistencias producidas al modificar uno de los archivos donde aparece la definición de los datos sin haber modificado los restantes.

Los elementos que deben aparecer en un archivo de cabecera son:

- Definiciones de aquellos tipos de datos que son compartidos por dos o más módulos.
- Definiciones de los prototipos de funciones que son compartidas por dos o más módulos.

Bajo ningún concepto debe figurar en los archivos cabecera la definición de los datos, es decir, la reserva de memoria para variables de un tipo determinado.

4.2 AUTOMATIZACIÓN DE TRABAJOS

A medida que estructuramos los módulos de una aplicación y el número de éstos se incrementa, se va introduciendo complejidad con respecto a las órdenes que se deben emitir para generar el programa ejecutable. Para que estuviéramos consiguiendo un efecto contrario al perseguido. Es decir, en lugar de incrementar la elegancia y sencillez de la gestión, parece que estemos introduciendo sofisticaciones que no aportan ventaja alguna.

El objetivo de esta sección es mostrar cómo podemos automatizar la parte relacionada con la gestión de órdenes para compilar las aplicaciones de una forma más cómoda y segura.

La primera solución que podemos aportar consiste en escribir un archivo de procesamiento por lotes en el que se encuentren todas las órdenes necesarias para compilar una aplicación.

Aunque esta solución es válida y cómoda, sin embargo, presenta algunos inconvenientes:

1. Cada vez que se ejecuta el archivo se recompilan todos los módulos fuentes.

Supongamos que la aplicación se compone de 50 módulos y que por necesidades de depuración se han modificado 5 de ellos. Al ejecutar la orden de compilar, se van a recompilar los 50 módulos cuando sólo es

necesario recompilar los 5 que han sido modificados. Esta forma de trabajo acarrea pérdidas de tiempo innecesarias.

2. En el archivo de procesamiento por lotes no quedan reflejadas todas las relaciones que existen entre los distintos archivos de la aplicación; en concreto, no están reflejadas las relaciones entre archivos de cabera y fuente.

El programa `make` fue diseñado para solucionar estas y otras deficiencias y poder automatizar las compilación de aplicaciones. Aunque aquí vamos a ver la forma de recompilar aplicaciones escritas en C/C++, el programa `make` se puede utilizar para gestionar proyectos escritos en otros lenguajes y para gestionar trabajos que no estén relacionados con el desarrollo de programas.

La sintaxis de esta orden es:

```
make [-f makefile] [opciones] [objetivos]
```

El programa `make` lee del archivo `makefile` la especificación de cómo los componentes de una aplicación están relacionados entre sí y cómo procesarlos para crear una versión actualizada del programa.

El archivo `makefile` se especifica con la opción `-f makefile`. Si esta opción no está presente, los archivos que se van a considerar por defecto son: `makefile` y `Makefile`. El nombre `Makefile` es el utilizado con más frecuencia por los programadores.

Objetivos suele ser el nombre del programa o programas que se deben generar como resultado de la actualización realizada por `make`. Conocida la relación entre módulos, `make` revisa la fecha de la última modificación de los archivos y determina la cantidad mínima de recompilación necesaria para generar una nueva versión del programa.

Una vez determinadas las órdenes que se deben ejecutar para regenerar la aplicación, el programa `make` se encarga de invocarlas.

4.3 SINTAXIS DE UN ARCHIVO MAKEFILE

Un archivo `Makefile` consiste de un conjunto de dependencias y reglas. Una dependencia tiene un objetivo (un archivo a crear) y un conjunto de archivos de los cuales depende. Las reglas describen cómo crear el archivo objetivo con base en los archivos dependencia.

Ahora veremos un ejemplo de un archivo `Makefile`:

```
1 editor : editor.o screen.o keyboard.o

2     gcc -o editor editor.o screen.o keyboard.o

3 editor.o : editor.c editor.h keyboard.h screen.h

4     gcc -c editor.c

5 screen.o : screen.c screen.h
```

```
6     gcc -c screen.c
7 keyboard.o : keyboard.c keyboard.h
8     gcc -c kerboard.c
9 clean :
10    -rm editor *.c
```

Para compilar editor, simplemente se ejecuta la siguiente instrucción:

```
$ make -f Makefile
```

Este archivo `Makefile` tiene 5 reglas. La primera, `editor`, es llamada objetivo por defecto (el archivo que `make` tratará de crear), tiene tres dependencias, `editor.o`, `screen.o` y `keyboard.o`; estos tres archivos deben de existir para que `editor` pueda ser creador. La línea 2 es el comando que `make` debe ejecutar. Las siguientes líneas contienen las reglas para crear cada archivo individual.

4.3.1 OBJETIVOS FALSOS (PHONY)

Además de los archivos objetivos normales, `make` nos permite especificar objetivos falsos (Phony Targets). Los objetivos falsos son nombrados así porque no corresponden a un archivo real. En el ejemplo anterior, `clean` es un objetivo falso. Estos objetivos existen para especificar comandos que `make` debe ejecutar. Sin embargo, dado que `clean` no tiene dependencias, sus comandos no son ejecutados automáticamente. Esto debido a lo siguiente: al momento de encontrar el objetivo `clean`, `make` ve si las dependencias existen y, como `clean` no tiene dependencias, `make` asume que el objetivo está actualizado. Si queremos usar este objetivo, el comando debería ser:

```
$ make -f makefile clean
```

De esta forma, `clean` remueve los archivos generados previamente (ejecutable y objeto). Si estamos pensando en distribuir nuestro programa, siempre es recomendable poner un objetivo `clean` que nos permita limpiar los directorios de trabajo.

4.3.2 VARIABLES

Para simplificar la edición y mantenimiento de los archivos Makefile, make nos permite crear y usar variables. Una variable es, simplemente, un nombre definido en el archivo que representa un texto; este texto es llamado valor de la variable. Una variable se define, de forma general, así:

VARNAME = texto

Para obtener el valor de esa variable, se debe encerrar el nombre entre paréntesis y precedido del símbolo $:

${VARNAME}

Las variables están, usualmente, definidas en la parte superior de un archivo Makefile. Por convención, los nombres de las variables se definen en mayúsculas, aunque no es requerido.

Si cambia algún valor sólo es necesario hacer un cambio en vez de muchos, simplificando el mantenimiento.

Otro ejemplo de un archivo Makefile. Esta vez conteniendo el manejo de variables y objetivos falsos.

```
1 all: myapp
2 # compilador
3 CC = gcc
4
5 #donde instalar la aplicación
6 INSTDIR = /usr/local/bin
7
8 #donde estan los archivos cabera
9 INCLUDE = ./include/
10
11 #opciones de compilación
12 CFLAGS = -g -Wall -ansi
13
14 #opciones de distribución
15 # CFLAGS = -O2 -Wall -ansi
16
17 myapp: main.o 2.o 3.o
18    ${CC} -o myapp main.o 2.o 3.o
19
20 main.o: main.c a.h
21    ${CC} -I${INCLUDE} ${CFLAGS} -c main.c
22
23 2.o: 2.c a.h b.h
```

```
24    ${CC} -I${INCLUDE} ${CFLAGS} -c 2.c
25
26 3.o: 3.c b.h c.h
27    ${CC} -I${INCLUDE} ${CFLAGS} -c 3.c
28
29 clean:
30    -rm myapp main.o 2.o 3.o
31
32 install: myapp
33    @if [ -d ${INSTDIR} ]; \
34                then \
35                cp myapp ${INSTDIR}; \
36                chmod a+x ${INSTDIR} /myapp; \
37                chmod pg-w ${INSTDIR} /myapp; \
38                echo "Instalado en ${INSTDIR}"; \
39    else \
40                echo "Error, ${INSTDIR} no existe"; \
41    fi
```

El objetivo `install` es dependiente de `myapp`, de esta forma `make` sabe que primero es necesario crear la aplicación y después ejecutar la regla. En este caso la regla, en realidad, un script de shell. Debido a que `make` invoca un shell para ejecutar las reglas y usa un shell para cada regla, debemos agregar diagonales invertidas (`\`) a cada instrucción para que forme una sola línea lógica y sean pasadas como una sola instrucción al shell.

El objetivo `install` ejecuta varias instrucciones, una después de otra, para instalar la aplicación en un directorio final. Pero no verifica si la instrucción se ejecutó correctamente, antes de pasar a la siguiente. Si queremos que exista esa verificación es necesario agregar `&&` entre los comandos, de esta forma:

```
33    @if [ -d ${INSTDIR} ]; \
34                then \
```

```
35              cp myapp ${INSTDIR}; &&\
36              chmod a+x ${INSTDIR} /myapp; &&\
37              chmod pg-w ${INSTDIR} /myapp; &&\
38              echo "Instalado en ${INSTDIR}"; \
39    else \
40              echo "Error, ${INSTDIR} no existe"; false; \
41    fi
```

APÉNDICE: LA ESTRUCTURA DE UN COMPILADOR

Por lo regular, consideramos al compilador como una caja simple que mapea un programa fuente a un programa destino con equivalencia semántica. Si abrimos esta caja un poco, podremos ver que hay dos procesos en esta asignación: análisis y síntesis.

La parte de análisis divide el programa fuente en componentes e impone una estructura gramatical sobre ellas. Después utiliza esta estructura para crear una representación intermedia del programa fuente. Si la parte del análisis detecta que el programa fuente está mal formado en cuanto a la sintaxis, o que no tiene una semántica consistente, entonces debe proporcionar mensajes informativos para que el usuario pueda corregirlo. La parte del análisis también recolecta información sobre el programa fuente y la almacena en una estructura de datos llamada tabla de símbolos, la cual se papa junto con la representación intermedia a la parte de la síntesis.

La parte de síntesis construye el programa destino deseado a partir de la representación intermedia y de la información en la tabla de símbolos. A la parte del análisis se le llama comúnmente el front-end del compilador; la parte de la síntesis (propiamente la traducción) es el back-end.

Si examinamos el proceso de compilación con más detalle, podremos ver que opera como una secuencia de fases, cada una de las cuales transforma una presentación del programa fuente en otro. En la práctica varias fases pueden agruparse, y las representaciones intermedias entre las fases agrupadas no necesitan construirse de manera explícita. La tabla de símbolos, que almacena información sobre todo el programa fuente, se utiliza en todas las fases del compilador.

Algunos compiladores tienen una fase de optimización de código independiente de la máquina, entre el front-end y el back-end. El propósito de esta optimización es realizar transformaciones sobre la representación intermedia, para que el back-end pueda producir un mejor programa destino de lo que hubiera producido con una representación intermedia sin optimizar.

ANÁLISIS LÉXICO

A la primera fase de un compilador se le llama análisis léxico o escaneo. El analizador léxico lee el flujo de los caracteres que componen el programa fuente y los agrupa en secuencias significativas, conocidas como lexemas. Para cada lexema, el analizador léxico produce como salida un token de la forma:

```
{ nombre_token, valor_atributo }
```

que pasa a la fase siguiente, el análisis de la sintaxis. En el token, el primer componente nombre-token es un símbolo abstracto que se utiliza durante el análisis sintáctico, y el segundo componente valor-atributo apunta a una entrada en la tabla de símbolos para este token. La información de la entrada en la tabla de símbolos se necesita para el análisis semántico y la generación de código.

Por ejemplo, suponga que un programa fuente contiene la instrucción de asignación:

```
posicion = inicial + velocidad * 60
```

Los caracteres en esta asignación podrían agruparse en los siguientes lexemas y mapearse a los siguientes tokens que se pasan al analizador sintáctico:

1. `posicion` es un lexema que se asigna a un token `{id, 1}`, en donde id es un símbolo abstracto que representa la palabra identificador y 1 apunta a la entrada en la tabla de símbolos para posicion. La entrada en la tabla de símbolos para un identificador contiene información acerca de éste, como su nombre y tipo.
2. El símbolo de asignación = es un lexema que se asigna al token `{=}`. Como este token no necesita un atributo-valor, hemos omitido el segundo componente. Podríamos hacer utilizado cualquier símbolo abstracto como asignar para el nombre-token, pero por conveniencia de notación hemos optado por usar el mismo lexema como el nombre para el símbolo abstracto.
3. `inicial` es un lexema que se asigna al token `{id, 2}`, en donde 2 apunta a la entrada en la tabla de símbolos para inicial.
4. + es un lexema que se asigna al token `{+}`.
5. `velocidad` es un lexema que se asigna al token `{id, 3}`, en donde 3 apunta a la entrada en la tabla de símbolos para `velocidad`.
6. * es un lexema que se asigna al token `{*}`.
7. `60` es un lexema que se asigna al token `{60}`.

El analizador léxico ignora los espacios en blanco que separan a los lexemas.

La representación de la instrucción de asignación después del análisis léxico sería:

```
{id, 1} {=} {id,2} {+} {id,3} {*} {60}
```

En esta representación, los nombres de los tokens =, +, y * son símbolos abstractos para los operadores de asignación, suma y multiplicación, respectivamente.

ANALIZADOR SINTÁCTICO

La segunda fase del compilador es el análisis sintáctico o parsing. El parser (analizador sintáctico) utiliza los primeros componentes de los tokens producidos por el analizador léxico para crear una representación intermedia en forma de árbol que describa la estructura gramatical del flujo de tokens. Una representación típica es el árbol sintáctico, en el cual cada nodo interior representa una operación y los hijos del nodo representan los argumentos de la operación.

Las fases siguientes del compilador utilizan la estructura gramatical para ayudar a analizar el programa fuente y generar el programa destino.

ANALIZADOR SEMÁNTICO

El analizador semántico utiliza el árbol sintáctico y la información en la tabla de símbolos para comprobar la consistencia semántica del programa fuente con la definición del lenguaje. También recopila información sobre el tipo y la guarda, ya sea en el árbol sintáctico o en la tabla de símbolos, para usarla más tarde durante la generación de código intermedio.

Una parte importante del análisis semántico es la comprobación (verificación) de tipos, en donde el compilador verifica que cada operador tenga operandos que coincidan. Por ejemplo, muchas definiciones de lenguajes de programación requieren que el índice de un arreglo sea entero; el compilador debe reportar un error si se utiliza un número de punto flotante para indexar el arreglo.

Las especificaciones del lenguaje pueden permitir ciertas conversiones de tipo conocidas como coerciones. Por ejemplo, puede aplicarse un operador binario aritmético a un par de enteros o a un par de números de punto flotante. Si el operador se aplica a un número de punto flotante y a un entero, el compilador puede convertir u obligar a que se convierta en un número de punto flotante.

GENERACIÓN DE CÓDIGO INTERMEDIO

En el proceso de traducir un programa fuente a código destino, un compilador puede construir una o más representaciones intermedias, las cuales pueden tener una variedad de formas. Los árboles sintácticos son una forma de representación intermedia; por lo general, se utilizan durante el análisis sintáctico y semántico.

Después del análisis sintáctico y semántico del programa fuente, muchos compiladores genera un nivel bajo explícito, o una representación intermedia similar al código máquina, que podemos considerar como un programa para una máquina abstracta. Esta representación intermedia debe tener dos propiedades importantes: debe ser fácil de producir y fácil de traducir en la máquina destino.

Consideremos una forma intermedia llamada código de tres direcciones, que consiste en una secuencia de instrucciones similares a ensamblador, con tres operandos por instrucción. Cada operando puede actuar como registro. La salida del generador de código intermedio consiste en la secuencia de código de tres direcciones, por ejemplo:

```
t1 = inttofloat(6)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Hay varios puntos que vale la pena mencionar sobre las instrucciones de tres direcciones. En primer lugar, cada instrucción de tres direcciones tiene, por lo menos, un operador del lado derecho. Por ende, estas instrucciones corrigen el orden en el que se van a realizar las operaciones; la multiplicación va antes que la suma en el programa fuente. En segundo lugar, el compilador debe generar un nombre temporal para guardar el valor calculado por una instrucción de tres direcciones. En tercer lugar, algunas "instrucciones de tres direcciones" como la primera y la última de la secuencia anterior, tienen menos de tres operandos.

OPTIMIZACIÓN DE CÓDIGO

La fase de optimización de código independiente de la máquina trata de mejorar el código intermedio, de manera que se produzca un mejor código destino. Por lo general, mejor significa más rápido, pero pueden lograrse otros objetivos, como un código más corto, o un código destino que consuma menor poder. Por ejemplo, un algoritmo directo genera el código intermedio, usando una instrucción para cada operador en la representación tipo árbol que produce el analizador sintáctico.

Un algoritmo simple de generación de código intermedio, seguido de la optimización de código, es una manera razonable de obtener un buen código de destino. El optimizador puede deducir que la conversión del 60, de un entero a punto flotante, puede realizarse de una vez por todas en un tiempo compilación, por lo que se puede eliminar la operación inttofloat sustituyendo el entero 60 por el número de punto flotante 60.0. Lo que es más, t3 se utiliza sólo una vez para transmitir su valor a id1, para que el optimizador pueda transformar en la siguiente secuencia más corta:

```
t1 = id3 * 60.0
id1 = id2 * t1;
```

Hay una gran variación en la cantidad de optimización de código que realizan los distintos compiladores. En aquellos que realizan la mayor optimización, a los que se les denomina como "compiladores optimizadores", se invierte mucho tiempo en esta fase. Hay optimizaciones simples que mejoran en forma considerable el tiempo de ejecución del programa destino, sin reducir demasiado la velocidad de la compilación.

GENERACIÓN DE CÓDIGO

El generador de código recibe como entrada una representación intermedia del programa fuente y la asigna al lenguaje destino. Si el lenguaje destino es código máquina, se seleccionan registro o ubicaciones (localidades) de memoria para cada una de las variables que utiliza el programa. Después, las instrucciones intermedias se traducen en secuencias de instrucciones de máquina que realizan la misma tarea. Un aspecto crucial de la generación de código es la asignación juiciosa de los registros para guardar las variables.

Por ejemplo, usando los registros R1 y R2, el código intermedio

```
LDF R2, id3
```

```
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
```

El primer operando de cada instrucción especifica un destino. La F de cada instrucción nos indica que trata con número de punto flotante. El código anterior carga el contenido de la dirección id3 en el registro R2, y después lo multiplica con la constante de punto flotante 60.0. El # indica que el número 60.0 se va a tratar como una constante inmediata. La tercer instrucción mueve id2 al registro R1 y la cuarta lo suma al valor que se había calculado antes en el registro R2. Por último, el valor en el registro R1 se almacena en la dirección de id1, por lo que el código implementa en forma correcta la instrucción de asignación.