Capítulo 1

Introducción

1.1. Los viejos (y buenos) tiempos

Los computadores siempre fueron máquinas de gran poder de cálculo y almacenamiento, y las empresas tenían un solo gran computador central de uso general que se utilizaba para todos los fines de la empresa.

Esto hacía la vida muy fácil, había que decidir qué computador comprar, y luego todas las soluciones de software y hardware provenían del mismo proveedor de los equipos. Por otro lado, los proveedores de equipos siempre fueron 2 grandes y unos 5 medianos. No había tampoco mucho donde elegir ni equivocarse (el empleado que tomaba la decisión sin arriegar el puesto, siempre elegía IBM).

1.2. Los nuevos (y confusos) tiempos

El advenimiento de los PC's, produjo cambios en esta estructura, pero no afectó fuertemente la organización computacional hasta la llegada de las redes, en que los computadores de diferentes tamaños se distribuyeron por toda la organización, pudiendo ahora interactuar entre sí. Los proveedores de PC's son cientos, las soluciones de software tienen filosofías y paradigmas diferentes y todas estas componentes se pueden interconectar en múltiples formas diferentes.

La vida ya no es simple para los gerentes de informática, el software, el hardware, los sistemas operativos y las redes forman ahora un *sistema computacional*. Y no hay uno ni dos, sino que tantas combinaciones como uno pueda imaginar, y la decisión final está en la persona a cargo, en el integrador de sistemas de diferentes marcas y proveedores.

Esta es la era de los sistemas abiertos.

1.3. Sistemas Abiertos

Hay múltiples componentes y sub-componentes en un sistema computacional. Sin embargo, para el integrador, los que importan son unos pocos:

- Software de Aplicación

  Es el sistema de software que operará en el sistema computacional: Base de Datos, aplicación orientada al objeto, etc.

- Sistema Operativo

  Es el nivel más bajo del software, que hace de interfaz entre el software de aplicación y el hardware.

- Hardware

  Es la arquitectura de la máquina en que correrá el sistema, tipo de procesador, memoria, disco, etc.

- Red

  La red es parte de casi todo lo anterior, y es lo que permite hacer aplicaciones distribuidas, que utilizan recursos de más de un computador (sistemas de archivos distribuidos, Cliente/Servidor, etc)

Un resumen de la posición de estos elementos puede verse en la Figura 1.1.

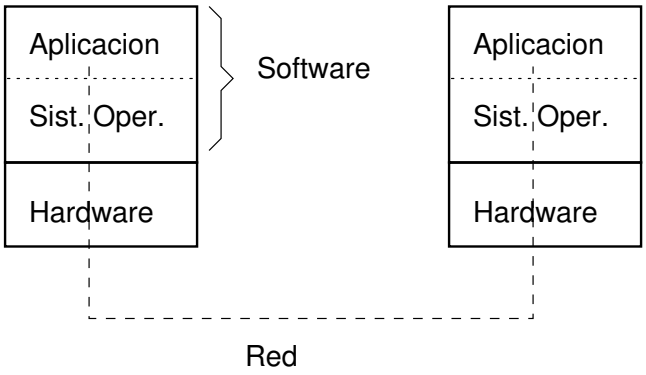

Figura 1.1: Sistema Computacional

En este curso estudiaremos las componentes del bajo nivel, la interfaz bajo nivel entre las aplicaciones y el sistema operativo, su relación con el Hardware y las redes. Este curso es el complemento (o el lado oscuro) del curso Desarrollo de Software de Aplicación, que se preocupa de las herramientas de alto nivel.

No nos preocuparemos de cómo el Sistema Operativo implementa las interfaces (eso se ve en el curso de Sistemas Operativos) sino de cómo utilizarlas eficientemente.

Capítulo 2

Arquitectura de Computadores

2.1. Introducción

El hardware de los computadores ha ido evolucionando, pero las ideas básicas son las mismas desde los primeros computadores. Básicamente podemos separar un computador en tres partes: El procesador (CPU), la memoria y los dispositivos de Entrada/Salida (E/S).

El procesador es un chip integrado que es capaz de "ejecutar"instrucciones de máquina.

La memoria es un dispositivo de alta velocidad, que permite almacenar datos, pero que es volátil: al apagar el computador esos datos se pierden.

Los dispositivos de E/S son todos los agentes que interactúan con el medio externo: terminales, impresoras, discos, cintas, etc.

Todos estos componentes se comunican entre ellos a través de buses de datos de alta velocidad.

2.2. Arquitectura

Desde un punto de vista lógico, lo que interesa es cómo un programa ejecuta en un computador. Para ello, el programa se almacena en memoria y el procesador lo va leyendo de allí. Las instrucciones que están almacenadas en la memoria son códigos que dependen del procesador.

2.2.1. Memoria

Los datos en la memoria del computador se almacenan como BITS (binary digits), o sea 0's y 1's. Para fines de traspaso entre el procesador y la memoria, estos bits se agrupan (por razones de eficiencia), definiendo lo que se conoce como el ancho de palabra del computador. Obviamente un ancho de palabra mayor permite mucha eficiencia en el manejo de datos cada vez más grandes, y define cual es el entero más grande representable en el computador por hardware.

La memoria del computador puede verse como un arreglo de “palabras”. Cada palabra tiene una posición en la memoria (su subíndice). La memoria permite acceso directo a una palabra almacenada si uno conoce su posición (se habla de dirección). El ancho de palabra también define cual es el tamaño máximo de memoria direccionable, puesto que las direcciones también se almacenan en palabras.

2.2.2. Procesador

El procesador tiene un poco de memoria interna que le permite almacenar estados y datos. Estas áreas de memoria interna se conocen como registros. Algunos registros son dedicados (como el que contiene la dirección de la memoria donde se está ejecutando) otros son de uso general, para almacenar datos.

En general, las instrucciones de máquina del procesador son una operación y sus argumentos. Los argumentos pueden ser registros, direcciones de memoria y constantes. Las instrucciones pueden ser aritméticas (sumar 2 al registro 3) o de control (saltar a la dirección de memoria contenida en el registro 2).

El procesador es capaz de pedir datos de la memoria, escribir datos en la memoria y ejecutar instrucciones. La velocidad de ejecución depende de cuánto tiempo tome cada instrucción, pero también de cuanto demore un acceso a memoria. El procesador tiene un reloj asociado, que define cuantos cambios de estado por segundo (o ciclos) puede realizar. Si todas las instrucciones se completaran en un ciclo, el procesador iría tan rápido como su reloj: 100MHz == 100 MIPS. Sin embargo las intrucciones complejas (multiplicar, dividir) pueden tomar unos cinco ciclos.

2.2.3. Representación de los datos

Todos los datos manejados por los programas deben tener una representación binaria asociada. Por ejemplo, los punteros a memoria son una palabra, los enteros también. Los flotantes pueden ser una o dos, etc.

Los enteros tienen una representación muy simple cuando son positivos: su valor en base 2. Sin embargo, hay que representar enteros negativos. Para ello, se usa el bit más significativo de la palabra. Si ese bit es 1, el número es negativo. Para que las instrucciones aritméticas funcionen bien, se usa además una representación especial del número negativo, suponiendo una palabra de 4 bits:

```
1  = 0001
-1 = not(0001) + 1 = 1110 + 1 = 1111
2  = 0010
-2 = 1101 + 1 = 1110
3  = 0011
-3 = 1100 + 1 = 1101

 1111        1101
+1110  +0011
11101  10000
```

Esto hace que en un computador de 32 bits, los enteros van desde $2^31 - 1$ hasta -2^31.

En general se habla de short integers (16 bits), long integers (32) y double integers (64). A pesar que todas las arquitecturas representan los enteros con la misma representación binaria, no todas ponen los bytes en el mismo orden. Es decir, al tener un puntero a un entero en memoria, el byte apuntado por este puntero puede ser el más significativo o el menos significativo. Esto es transparente para el programador, salvo que se cargue un entero como un arreglo de bytes, puesto que ahí el orden será diferente.

Hay dos representaciones más populares hoy en día (existieron otras versiones, pero ya no vale la pena recordarlas): el byte más significativo primero (*big endian*) y el byte menos significativo primero (*little endian*[1] Intel y Digital son *little endians* y todo el resto son `big endians`.

Los caracteres en cambio se representan típicamente en 8 bits (conocido como un byte), permitiendo desde 0 hasta 255. El estándar americano es ASCII (sólo usa 7 bits) y ha sido extendido a soporte internacional con ISO-8859 que usa 8 bits.

Tanto la notación decimal como la binaria es poco práctica para mostrar la representación interna de la máquina (que son siempre múltiplos de bytes). La notación más usada es la notación hexadecimal.

En hexadecimal (o base 16) existen 16 dígitos: 0-9, a, b, c, d, e, f. Como siempre, para calcular el resultado debemos multiplicar el dígito por la potencia de 16 correspondiente a la posición. Por ejemplo: `fe` es: $15 * 16 + 14 = 254$.

En C, para incluir constantes en hexadecimal, basta prefijarla con `0x`. Por ejemplo: `0xfe`. Esta notación es razonable, porque agrupa en un dígito 4 bits, lo que muestra un byte con dos dígitos. O sea un byte contiene un número entre `0` y `0xff`. El dígito `0xf` representa cuatro bits en 1.

2.2.4. Código de máquina

Un procesador sólo entiende dígitos binarios, y operaciones simples. Un programa de alto nivel (Pascal, Turing, C, etc) se traduce (a través de otro programa, llamado compilador) a lenguaje de máquina. Inicialmente, la tendencia en la computación fue hacia lenguajes de máquina de nivel cada vez más alto (CISC), para ayudar a los humanos que programaban en él (todos los sistemas que requerían alta eficiencia se escribían en lenguaje de máquina, particularmente los Sistemas Operativos).

A medida que la tecnología de compilación progresaba y que los procesadores no podían andar más rápido por problemas de complejidad en sus circuitos, se diseñó una nueva estrategia: hacer lenguajes de máquina simples (RISC), difíciles de programar para un humano, pero muy simples de generar para un programa. Con esto, un procesador se reducía en complejidad, pudiendo ejecutar mucho más rápido (aunque ahora son instrucciones más simples).

Como ejemplo estudiaremos un lenguaje de máquina RISC, que es el Assembler SPARC de Sun (ver Manual de Sparc, de Chris Perlberg).

[1] Los términos vienen de una historia de los Viajes de Gulliver, donde habían dos bandos irreconciliables con esos nombres. Los *little endians* decían que el huevo debía comerse por el lado más pequeño y los *big endians* decían lo contrario.

2.3. Programación Eficiente en C

C es un lenguaje de alto nivel, que no pierde de vista la arquitectura típica de un computador, de modo de ser más eficiente en ejecución.

2.3.1. Globales, locales, dinámicas

Las áreas de datos en C están divididas en tres áreas principales, que cumplen con requisitos de eficiencia en ejecución y en uso memoria diferentes:

Área de Datos Estática

Al declarar una variable global o una variable local con prefijo `static`, el compilador genera una instrucción `.data` en assembler, para dejar un espacio suficiente asignado para esa variable. Si es un arreglo de tamaño máximo N, se reserva un espacio de tamaño N, independientemente de si el programa al final lo usa o no.

Este espacio está en una dirección de memoria (calculada al hacer el `ld` del ejecutable) fija.

El Stack

Al declarar una variable local, y al pasar parámetros a las subrutinas, se usa espacio del stack de activación del proceso. Esto es más caro en ejecución, porque hay que hacer crecer el stack dinámicamente, pero muy poco más caro (puesto que normalmente es un incremento y una comparación para ver si aún queda espacio). Sin embargo, también se corre el riesgo que en ejecución nos quedemos sin memoria para estos datos. La gran ventaja del espacio en el stack es que sólo se paga el costo en memoria si los datos son necesarios (si la rutina es invocada alguna vez) y se liberan automáticamente en cuanto ya no son requeridos (al retornar).

Memoria Dinámica

Al usar `malloc` y `free` obtenemos la memoria justa que se requiere en el programa, lo que hace que es muy eficiente en memoria, pero más ineficiente en ejecución (buscar bloques de memoria disponibles de tamaño variable es caro).

Por otro lado, la liberación de los espacios es compleja, puesto que siempre podemos tener un puntero a ellos en otras partes del programa.

2.3.2. Arreglos y Matrices

La representación de los datos es un problema complejo en eficiencia. Por ejemplo, un arreglo de estructuras, en una arquitectura en que los punteros requieren ser direcciones de bytes, requiere la siguiente operación:

```
j = a[i] es:

j <- *((char *)a+(i*sizeof a[i]))
```

Por cada subindicación, requiero multiplicar el índice por cuatro, sumárselo a `a` y hacer un load de esa dirección. En C, esto se genera automáticamente, y por eso se considera equivalente el decir:

```
a[i] == *(a+i)
```

Por ejemplo el siguiente programa imprime correctamente 3 en todas sus líneas:

```
int a[]={1,2,3,4};

main()
{
    int *b;

    b = a;
    printf("%d\n", b[2]);
    printf("%d\n", *(b+2));
    printf("%d\n", a[2]);
    printf("%d\n", *(a+2));
    printf("%d\n", *((char *)a+(2*sizeof a[2]))
}
```

En assembler, la idea es que hay que multiplicar el subíndice por el número de bytes que ocupa la estructura (eso retorna `sizeof`). Por ejemplo `sizeof(int)` no es igual en todas las arquitecturas.

2.3.3. Manejo de bits

En C, se pueden manipular directamente los bits en memoria. Para esto existen las operaciones usuales sobre bits (and, or, xor, not) y también shift. El shift es con o sin signo según si el argumento es un entero con o sin signo.

Esto es muy útil para manejar variables de un bit (flags), para manejar marcas en objetos, etc.

Por ejemplo, este código cuenta el número de bits en uno del parámetro:

```
bits1(unsigned int i)
{
    int cnt = 0;

    while(i != 0)
      {
/* veo si el último bit es 1 */
        if( i & 1 ) cnt++;

/* i es unsigned para que el shift
 * no me propague el bit de signo
 */
```

```
        i >>= 1;
      }
}
```

Encender y apagar bits es posible usando máscaras, que son secuencias de bits, y usando and (para apagar) y or (para encender).

Por ejemplo, quiero poner en uno el tercer bit y el primer bit, sin modificar el resto:

```
flags = flags | 4 | 1;
```

Ahora quiero ponerlos en 0:

```
flags = flags & ~(4 | 1);
```

Muchas veces, por razones de eficiencia, prefiero usar shift que multiplicar o dividir por potencias de dos.

Mezclando manejo de bits y casts de punteros, puedo hacer programas no portables, o que se comportan distinto según la arquitectura. Por ejemplo:

```
print_bytes(int i)
{
    char *c;

    c = (char *)&i;
    printf("%x\n", *c++);
    printf("%x\n", *c++);
    printf("%x\n", *c++);
    printf("%x\n", *c++);
}
```

Esto dará resultados diferentes si la máquina es *big endian* o *little endian* y dará resultados falsos si no es de 32 bits. Por ejemplo un Intel bajo DOS opera como un procesador de 16 bits, pero bajo Linux se dispone de los 32 bits reales.

2.3.4. Macros

En C existe un preprocesador de macros, que procesa las líneas que comienzan con `#`. Es simplemente un expansor de macros, de directivas `include` y también soporta compilación condicional (`ifdef`).

Las macros son interesantes para definir constantes, pero también permiten macros con parámetros, con son expandidas textualmente en el código. Un uso típico es para la manipulación de flags de bits sin que se noten demasiado:

```
#define ECHO_FLAG 4

#define set_echo(flags) ((flags) |= ECHO_FLAG)
#define unset_echo(flags) ((flags) &= ~ECHO_FLAG)
#define is_echo(flags) ((flags) & ECHO_FLAG)
```

Los paréntesis redundantes en el parámetro de las macros son importantes para evitar conflictos cuando el parámetro es una expresión. Estas declaraciones típicamente se ponen en un archivo `.h` que será incluido por la aplicación.

Obviamente la macro podría ser reemplazada por una función, pero la función no podría modificar directamente el parámetro, salvo que se le pasara un puntero a él. Esto es complejo, porque el programador debe estar advertido que estas funciones son macros.

La compilación condicional permite generar código diferente según definiciones de la arquitectura y/o opciones de compilación. Por ejemplo, es usual tener una opción de compilación con código para debugging o sin debugging.

```
#ifdef DEBUG
#define DEBUG_F(str)  {if(debug_flag) fputs(str, stderr);}
#else  DEBUG
#define DEBUG_F(str)
#endif
```

Luego, en el código, puedo escribir:

```
    if( option == "-v" )
      {
        verbose = 1;
        DEBUG_F("ejecuto con opcion -v\n");
      }
    else if( option == "-d" )
      {
#ifdef DEBUG
        debug_flag = 1;
#else  DEBUG
        fprintf(stderr, "programa fue compilado sin soporte para debug\n")
        exit(1);
#endif DEBUG
      }
```

Al compilar, puedo elegir tener o no tener soporte para debugging, si DEBUG no está definido en ninguno de los archivos:

```
% cc -DDEBUG p.c
\end{end{verbatim}

Existe una versión más flexible del comando {\tt #ifdef} que se
llama {\tt #if}. Ésta soporta expresiones constantes, pero se usa poco.

\subsection{Tablas de Saltos}

Al tener una secuencia larga de selecciones de código para ser ejecutado,
es muy ineficiente hacer todas las preguntas. Por ejemplo, un editor
```

```
acaba de detectar que una tecla fue oprimida, y quiere saber qué comando es:
\begin{verbatim}
if( c == 'q' )
    quit();
else if( c == 'a' )
    insert();
else if( c == 'w' )
    write();
else if( c == 'x' )
  {
    write();
    quit();
  }
else if ...
...
else
    fprintf(stderr, "Unknown Command\n");
```

Esto involucra una comparación y un salto por cada alternativa. Una mejor solución es usar un `switch`:

```
switch(c)
  {
case 'q':
        quit();
        break;
case 'a':
        insert();
        break;
case 'w':
        write();
        break;
case 'x':
        write();
        quit();
        break;
...
default:
        fprintf(stderr, "Unknown Command\n");
        break;
  }
```

Al usar `switch` hay que fijarse en siempre poner las instrucciones `break` antes del `case` siguiente, sino sigue ejecutando las instrucciones de más abajo.

Un `switch` es mucho más eficiente, porque no genera una comparación por cada alternativa. El compilador genera una tabla de saltos. La idea es sub-indicar directamente con el parámetro del `switch` en una tabla, y en ese lugar debe estar el puntero al código que debemos ejecutar (ver Figura 2.1).

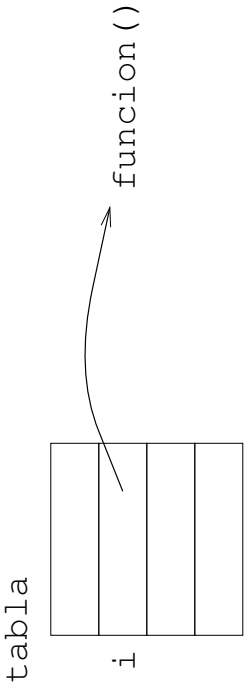

Figura 2.1: Tabla de Saltos

El compilador arma una tabla que va desde el mínimo hasta el máximo de los valores constantes presentes en instrucciones `case`. Antes de subindicar, debe validar que el parámetro esté en ese rango (dos comparaciones) y luego usa la tabla (una indirección y un salto incondicional).

En C, una posibilidad es implementar este concepto directamente, usando un arreglo de funciones. Puedo tener una función asociada a cada tecla posible, e invocarla directamente:

```
extern quit(), write(), insert(), error(), write_quit();
for( c = 0; c < 256; c++ )
    tabla[c] = error;

tabla['q'] = quit;
tabla['a'] = insert;
tabla['w'] = write;
tabla['x'] = write_quit;
...

(*tabla[c])();
```

En C, esto se declara en forma compleja, pero el concepto es que se pueden manipular punteros a funciones. La tabla del ejemplo se declara como:

```
void (*tabla[256])();
```

Que es un arreglo de punteros a funciones que no retornan valores.

2.3.5. Saltos no locales

Una función que pertenece a la biblioteca C, pero que es parte de las cosas importantes del lenguaje, es `setjmp`. Esto permite hacer `goto` no locales, implicando también deshacer el stack de activación.

El clásico ejemplo de uso es cuando estamos en una función cualquiera del programa y detectamos un error, que debería hacernos volver al ciclo principal, que está en otra parte:

```
/* Main loop */
while( (cmd = read_cmd()) != QUIT )
  {
     execute(cmd);
  }

...
error(char *str)
{
    fprintf(stderr, "%s\n", str);
    --> Volver al while!
}
```

Para eso debemos insertar una marca en la ejecución del programa con la función `setjmp` que recibe una variable de tipo `jmp_buf`. Luego podemos saltar a esa marca usando `longjmp`. El ejemplo anterior sería:

```
#include <setjmp.h>

jmp_buf label;

...
/* Main loop */
setjmp(label);
while( (cmd = read_cmd()) != QUIT )
  {
     execute(cmd);
  }

...
error(char *str)
{
    fprintf(stderr, "%s\n", str);
    longjmp(label, 1);
}
```

El segundo argumento de `longjmp` es un entero que se usa como valor de retorno de `setjmp`. La primera vez que `setjmp` es invocado retorna 0, permitiendo diferenciar en un `if` la primera invocacio'n de las veces que se retorna desde un `longjmp`.

Capítulo 3

Unix

3.1. Introducción

Este capítulo busca estudiar en detalle las funcionalidades del núcleo del S.O. Unix.

3.2. Entrada/Salida

3.2.1. Sistema de Archivos

Dispositivos, archivos, directorios, links simbolicos

3.2.2. Open/Read/Write

3.2.3. Directorios

stat, lstat, readdir, links hard

3.3. Señales

3.4. Procesos