
Resumen COMPUTADORS

Paralelismo y Sistemas Distribuidos
Grado en ciencia e ingeniería de datos
Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)

Licencia



**Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)**

<https://creativecommons.org/licenses/by-nc-sa/4.0/deed.es>

SOFTWARE

Generación de ejecutables

- Nuestros ejecutables están generalmente compuestos por N ficheros que generamos nosotros y N ficheros que ya existen (instalados en el sistema) y que utilizamos
 - Los ficheros que utilizamos son los “**ficheros fuente**”
 - ▶ Ficheros con definición de variables y código de funciones (extensión .c)
 - ▶ Ficheros “cabeceras o *includes*”: con definiciones de tipos de datos, constantes y cabeceras de funciones (extensión .h)
 - ▶ En el caso de programitas puede ser solo 1 fichero .c
 - Los ficheros ya instalados es lo que se conoce como **librerías**
 - ▶ Una librería incluye un fichero que ya ha sido procesado por el compilador (extensión .a o .so) y un/varios ficheros cabecera

Generación de ejecutables

■ Herramientas

● Paso 1: **Compilador** (gcc)

- ▶ Procesa 1 fichero fuente, comprueba la sintaxis y los tipos de datos
- ▶ Genera un fichero nuevo con un formato "similar" al código ejecutable. Ficheros "objeto"
- ▶ Algunas funciones y definición de variables pueden no estar incluidos en el fichero. En este caso el compilador lo marca como "undefined"

● Paso 2: **Linkador** / Enlazador (gcc)

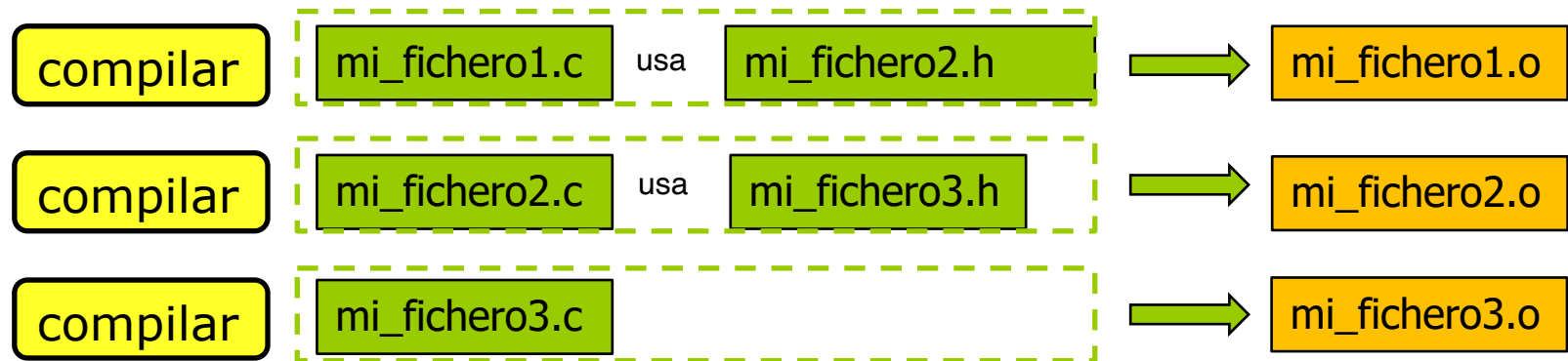
- ▶ Procesa N ficheros "objeto" y genera un único ejecutable donde todas las funciones y variables están definidas

■ En programas pequeños estos pasos se hacen a la vez pero en software grandes se separan en 2 para ahorrar tiempo

- Opción 1, todo junto: se compilan todos los ficheros siempre
- Opción 2, separado: se compilan solo los ficheros modificados

Generación de ejecutables

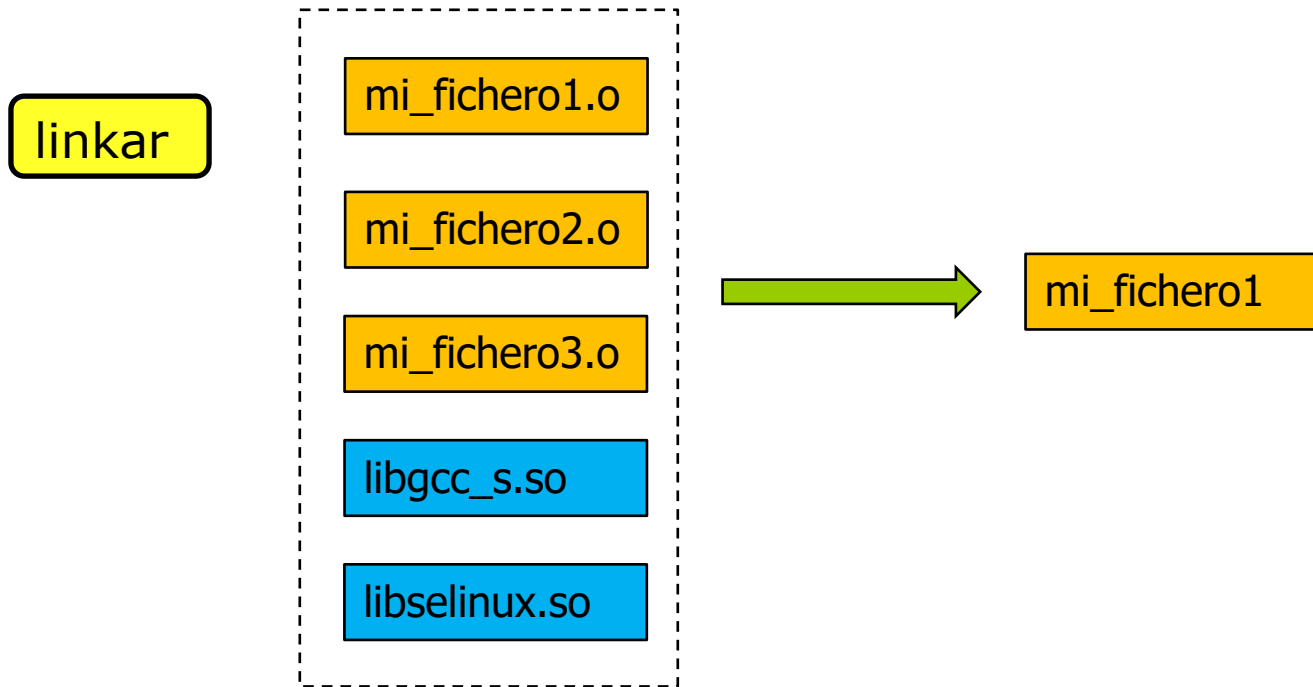
Paso 1: compilar, cada fichero fuente independientemente. Cada fichero genera 1 fichero "objeto". El fichero objeto es casi código ejecutable



```
$ gcc -c mi_fichero1.c  
$ gcc -c mi_fichero2.c  
$ gcc -c mi_fichero3.c
```

Generación de ejecutables

- Una vez tenemos nuestros ficheros "compilados", generamos nuestro ejecutable
- Las librerías del sistema no hay que especificarlas
- Los fichero cabecera tampoco ya que están incluidas en el código



```
$ gcc -o mi_fichero1 mi_fichero1.o mi_fichero2.o mi_fichero3.o
```

Ejemplo

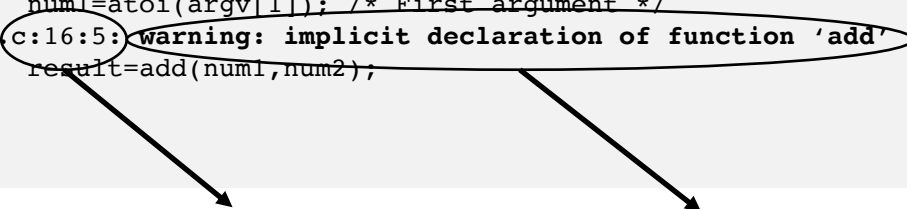
```
/* add.c */
void usage()
{
    printf("Usage: add num1 num2\n");
    exit(0);
}
void main(int argc, char *argv[])
{
    int num1, num2, result;
    printf("hello world! \n");
    printf("This program adds two numbers\n");
    if (argc != 3) usage();
    printf("I'm %s and my arguments are %s and %s\n", argv[0], argv[1], argv[2]);
    num1 = atoi(argv[1]); /* First argument */
    num2 = atoi(argv[2]); /* Second argument */
    result = add(num1, num2);
    printf("The result is %d\n", result);
}
```

```
/* add_function.c */
int add(int num1, int num2)
{
    return num1 + num2;
}
```


Ejemplo

- Compilar add.c devuelve algunos warnings (`gcc -c -Wimplicit-function-declaration add.c`)

```
add.c:4:5: warning: incompatible implicit declaration of built-in function 'printf' [enabled by default]
    printf("Usage: add num1 num2\n");
add.c:5:5: warning: incompatible implicit declaration of built-in function 'exit' [enabled by default]
    exit(0);
add.c: In function 'main':
add.c:10:5: warning: incompatible implicit declaration of built-in function 'printf' [enabled by default]
    printf("hello world! \n");
add.c:14:5: warning: implicit declaration of function 'atoi' [-Wimplicit-function-declaration]
    num1=atoi(argv[1]); /* First argument */
add.c:16:5: warning: implicit declaration of function 'add' [-Wimplicit-function-declaration]
    result=add(num1,num2);
```



Linea del fichero

Tipo de error/warning

- El compilador no “conoce” estas funciones, falta el prototipo/cabecera de la función

Ejemplo

```
/* add_function.h */
#ifndef ADD_FUNCTION_H
#define ADD_FUNCTION_H
int add(int num1,int num2);
#endif
```

```
/* add.c */
#include <stdio.h>
#include <stdlib.h>
#include <add_function.h>
void usage()
{
...
}
void main(int argc,char *argv[])
{
...
}
```

```
$gcc -c -Wimplicit-function-declaration -I add.c
$gcc -c add_function.c
$gcc -o add add.o add_function.o
$./add 1 2
hello world!
This program adds two numbers
I'm ./add and my arguments are 1 and 2
The result is 3
```

Ejemplo

- Típicamente usamos la herramienta **Makefile** para automatizar la generación de ejecutables y minimizar errores

```
$cat Makefile
all:add
add:add.o add_function.o
      gcc -o add add.o add_function.o
add.o:add.c
      gcc -Wimplicit-function-declaration -I. -c add.c
add_function.o:add_function.c
      gcc -Wimplicit-function-declaration -c add_function.c
clean:
      rm add *.o
$make
```

Tipos de datos

- Tipos de datos habituales en nuestros programas
- Depende del lenguaje de programación
- El compilador se encarga de reservar espacio en memoria para la variable y comprobar compatibilidad
 - Simples (no están todos)
 - ▶ char, int, float, double
 - ▶ Se puede definir un "puntero a " (int *, char *)
 - Complejos:
 - ▶ Vectores o matrices
 - int v[100];
 - int m[100][100]
 - ▶ Struct:

```
struct mis_datos{  
    int v[100];  
    int m[100][100];  
    char op;  
}
```
 - ▶ Unions
 - Es un conjunto de N posibles tipos: la variable usa el máximo de cada uno de los posibles tipos individualmente
 - Ej:

```
union int_o_char{  
    int x;  
    char c;  
}
```

Funciones

- Una función agrupa una serie de operaciones. Normalmente por claridad del código y reutilización.
- Partes
 - Valor de retorno, puede ser void.
 - Nombre
 - Argumentos, puede ser un número indefinido

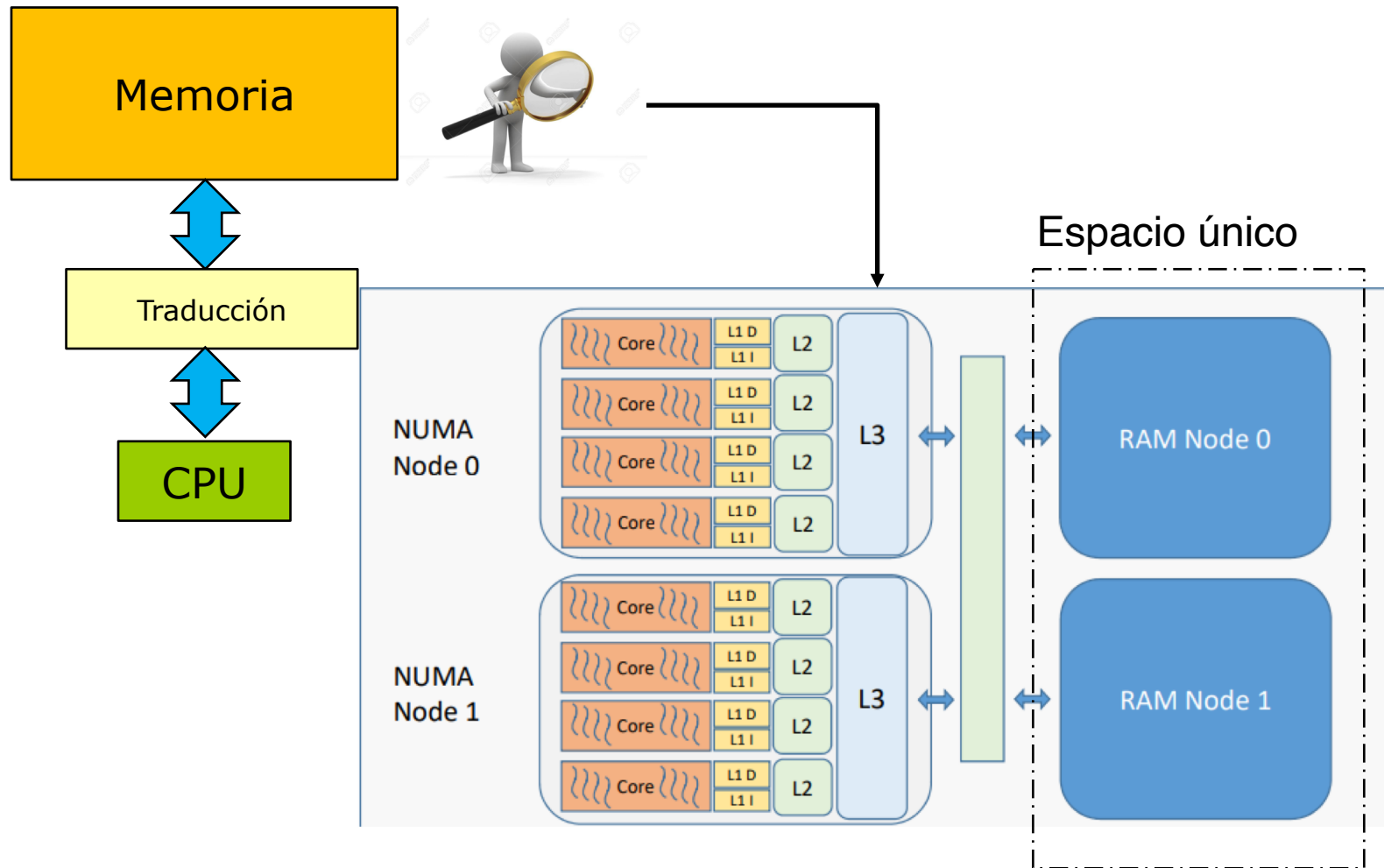
```
/* add_function.c */  
int add(int num1,int num2)  
{  
    return num1+num2;  
}
```

HARDWARE

Arquitectura

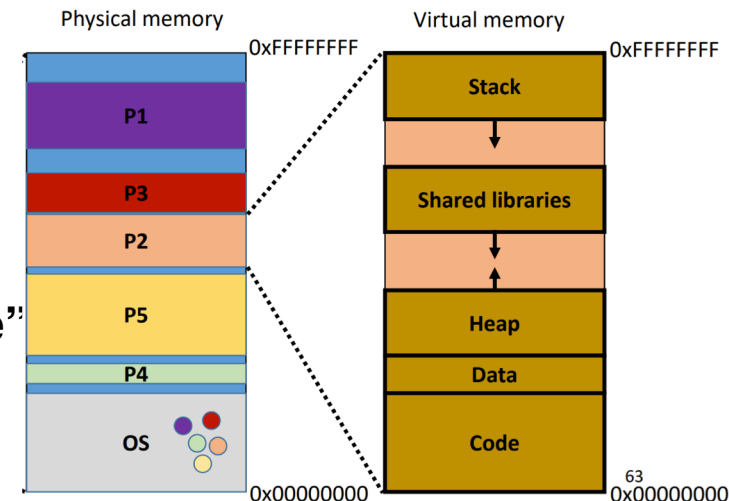
- Modelo de arquitectura
- **Tenemos el programa (codigo + datos) en memoria**
- Etapas
 - La CPU lee una instrucción (código) de memoria
 - Se decodifica (qué hace esta instrucción?)
 - Se accede a los datos si los necesita
 - Se realiza la operación
 - Se almacena el resultado
 - ▶ Si hay resultado
 - ▶ Si no hay ningún fallo

Arquitectura



Arquitectura

- Los accesos a memoria son lentos, por eso tenemos los diferentes niveles de memoria cache
 - El acceso es nivel de línea de cache (N bytes, típicamente 64)
 - Cada nivel es mayor que el anterior y más lento ($L3 > L2 > L1$)
 - Cada nivel es inclusivo, L3 incluye todas las líneas de L2. L2 incluye todas las líneas de L1.
 - En la mayoría de arquitecturas L1 está separado en código y datos para aprovechar al máximo la localidad espacial y temporal de cada caso.
 - Gracias al sistema de coherencia de memoria, la memoria principal se ve como un espacio consecutivo
 - **A recordar**
 - ▶ Dos datos diferentes pueden caer en la misma “línea de cache”

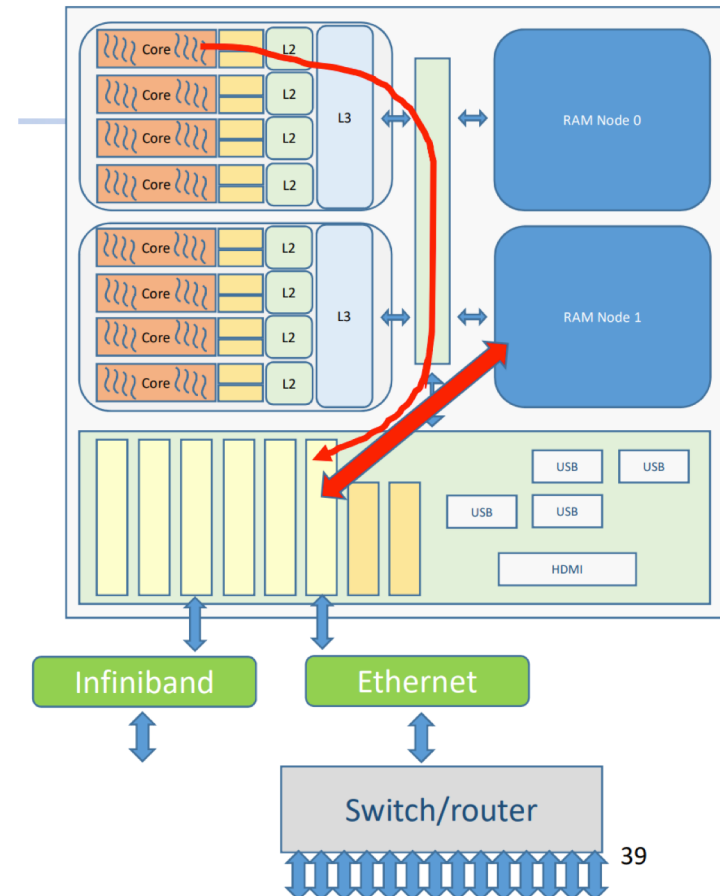
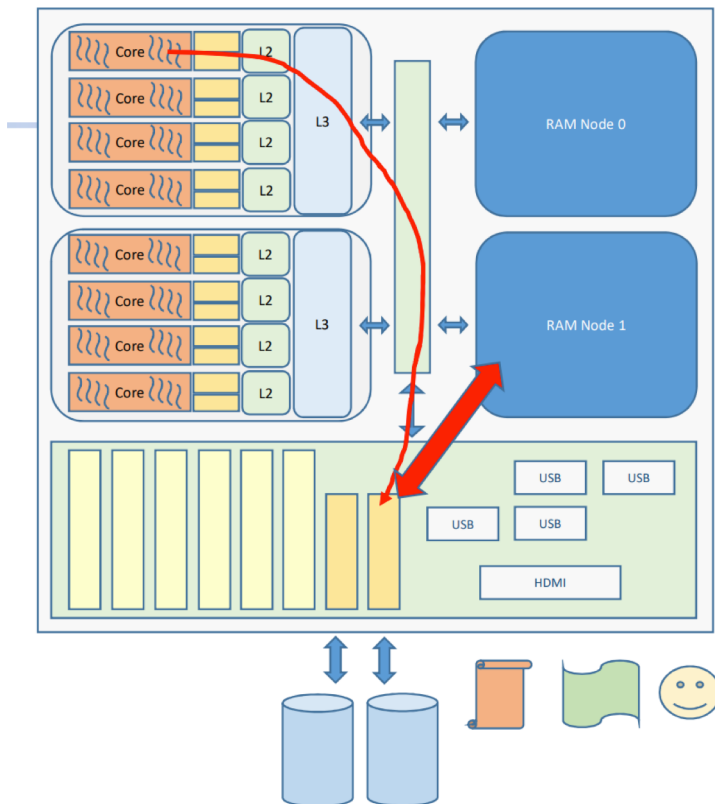


Arquitectura

- Dentro de cada CPU hay N cores
 - Cada Core tiene (típicamente)
 - ▶ N unidades funcionales de enteros/vectoriales
 - ▶ Bancos de registros de diferentes tamaños
 - ▶ Cache L1/L2
- Ejemplo: Intel® Xeon® Platinum 8160 Processor (MN4)
 - 33M Cache, 2.10 GHz (maxima 3.70 GHz)
 - **24 cores**
 - 48 threads (Intel® HT Technology)
 - 2 AVX-512 FMA Units
 - Intel® SSE4.2, Intel® AVX, Intel® AVX2, Intel® AVX-512
- <https://ark.intel.com/products/120501/Intel-Xeon-Platinum-8160-Processor-33M-Cache-2-10-GHz->

Arquitectura

- Existen otros componentes en un nodo además de la CPU y la memoria
 - ▶ GPUS, disco, network,...
 - ▶ Se acceden a través de operaciones de lenguaje máquina diferentes



Ejecutables y arquitectura

- Para poder ejecutar un programa, es necesario copiarlo (cargarlo) desde disco a memoria, inicializar las estructuras hardware de gestión de la memoria, de los dispositivos, de las interrupciones, inicializar los registros de la CPU, etc.
- Todos (algunos) estos pasos hay que hacerlos cada vez que queremos ejecutar un programa nuevo o queremos modificar alguna característica del programa que se está ejecutando.
- Estos pasos son muy complejos
- Hay una posibilidad real de afectar a la máquina si se comete un error
- Dependiendo de la máquina, algunos pasos son automáticos (por hardware) o no
- Estas y otras razones (gestión privilegios, ejecución multi-programada, etc) justifican la existencia del Sistema Operativo como software privilegiado que controla el sistema.

EL SISTEMA OPERATIVO

El sistema operativo

- Es un software que se ejecuta con privilegios (privilegios a nivel hardware) que nos ofrece:
 - **Protección** al ejecutar nuestros programas frente a interferencias no deseadas de otros programas también en ejecución
 - ▶ Especialmente si son de otros usuarios
 - **Robustez** al ejecutar nuestros programas evitando que podamos dañar al hardware, ya que el acceso a instrucciones privilegiadas se hace siempre a través del S.O.
 - **Usabilidad** ya que minimiza la complejidad de acceder a la máquina. Aunque el uso directo del S.O. no es sencillo, es infinitamente más sencillo que acceder a la máquina directamente.

El sistema operativo

- Es un software que se ejecuta con privilegios (privilegios a nivel hardware) que nos ofrece:
 - **Portabilidad** ya que permite que un mismo programa se ejecute en diferentes arquitecturas siempre simplemente recompilándolo, incluso, si es el mismo S.O. y mismo lenguaje máquina, sin recompilar.
 - **Eficiencia** ya que es un software sofisticado que hace un uso eficiente de la máquina, especialmente en casos de **multi-programación**.
 - **Configurabilidad** de los recursos, permitiendo la asignación de más o menos porcentaje de la CPU a usuarios, grupos, etc. Porcentajes de disco, etc.

El Sistema operativo

- El S.O. ofrece servicios muy potentes pero muy básicos (poco amigables para un usuario normal) que permiten construir, utilizándolos, otros programas de sistema muy potentes.
- Servicios del S.O:
 - **Gestión de “programas” (procesos):** Creación, inicialización, asignación de recursos (cpu, memoria, acceso a dispositivos), gestión del uso de la CPU (políticas de planificación), etc.
 - **Gestión de los dispositivos:** Red, Disco, “ficheros”, “consolas”, etc
 - ▶ Algunos son físicos, otros son “virtuales”. Por ejemplo: “fichero”
 - ▶ Inicialización, gestión de transferencias, asignación a programas, gestión de permisos, gestión de “espacios” en caso que aplique, compartición, gestión de accesos (políticas de acceso)
 - **Gestión de la memoria:** Carga de los programas en memoria, asignación de espacios, gestión de la memoria reservada durante la ejecución.

El Sistema operativo

- El S.O. controla el hardware controlando las interrupciones que se generan
 - Periódicamente por el reloj del Sistema
 - Al apretar una tecla
 - Al recibir datos por la red
 - Al finalizar una lectura o escritura de datos en disco
 - Etc
- **El S.O. ofrece su funcionalidad mediante "llamadas a sistema"**
 - Desde el punto de vista del programador son funciones normales
 - `fork()`, `execvp(..)`, `exit(...)`, `open(..)`, `read(...)`, `write(...)`, `close()`, `malloc(...)`
- Como programadores, habitualmente utilizaremos librerías que usan esta funcionalidad básica y que nos simplifican aún más la vida

El Sistema operativo

```
/* hello.c */
#include <stdio.h> /* printf */
#include <stdlib.h> /* exit */
#include <string.h> /* strlen */
void main(int argc,char *argv[])
{
    if (argc!=2) exit(1);
    /* Using libc printf */
    printf("hello world! , I'm %s and my argument is %s \n",argv[0],argv[1]);

    /* Using write system call */
    char my_text[256];
    sprintf(my_text,"hello world! , I'm %s and my argument is %s
\n",argv[0],argv[1]);
    write(1,my_text,strlen(my_text));
}
```

```
$gcc -o hello hello.c
$./hello 1
hello world! , I'm ./hello and my argument is 1
hello world! , I'm ./hello and my argument is 1
```

Sistemas operativos

- Algunos conceptos a recordar: Proceso
 - Definición clásica (Linux): Programa que está en ejecución
 - Es una definición antigua, en programas paralelos, 1 programa se corresponde típicamente con N procesos
 - Cada proceso tiene su espacio de direcciones en memoria asignado
 - ▶ Puede haber memoria compartida entre procesos pero no por defecto
 - Internamente se representa con una estructura de datos llamada genéricamente PCB (Process Control Block)
 - Cada proceso tiene sus estructuras de datos (locales) para acceder a dispositivos: *file descriptor table* o tabla de canales

Sistemas operativos

■ Thread

- Un thread es una unidad de ejecución vinculada a un proceso
- Un proceso tiene entre 1 y N threads
- Todos los threads de un proceso comparten sus recursos: memoria y acceso a dispositivos
- Cuando el S.O. Asigna un core, lo hace a un thread

ANÁLISIS DE RENDIMIENTO

Análisis de rendimiento

- El objetivo de nuestros programas es que sean correctos y eficientes
- **Correctos** → necesitaremos algún método de validar los resultados
 - Ejecución secuencial (1 proceso-1 thread): normalmente habrá algún valor de referencia para comprobar el resultado
 - Ejecución paralela: normalmente se comprueba con el resultado (previamente validado) en secuencial.
 - ▶ Los problemas se magnifican al aumentar el nivel de paralelismo, así que no es suficiente con realizar 1 prueba con pocos threads.

Análisis de rendimiento

■ Eficientes

- Minimizar el tiempo de ejecución → obtener resultados más rápidos
- Minimizar el consumo de memoria → los nodos tienen una memoria limitada, si un thread consume mucha memoria, quizás no podremos aprovechar todos los cores del nodo
- Hacer un uso eficiente de las cpus → obtener una aceleración, idealmente, proporcional al número de cores que usamos
- Si nuestro programa genera datos, intentar desacoplar el acceso a datos del acceso a cpu para evitar esperas ineficientes y tener la cpu “parada”

Análisis de rendimiento

- ¿Cómo podemos saber si el tiempo de ejecución es el más óptimo?
 - ¿Cómo podemos saber cuanta memoria estamos usando?
 - ¿Cómo podemos medir el tiempo de ejecución?
 - ¿Cómo podemos saber cuantos accesos para leer/escribir hacemos?
-
- ¿Porqué es importante?
 - Hay que tener en cuenta, que normalmente tendremos acceso a máquinas potentes por un tiempo limitado, cuanto más eficiente sean nuestros programas, más ejecuciones útiles podremos hacer en ese tiempo asignado
 - Y porque los recursos son caros!

Análisis de rendimiento

■ Funciones para medir detalles

- Tiempo de ejecución de trozos de código
- Espacio de memoria ocupado
- Información detallada sobre el hardware como accesos a memoria realizados en un intervalo
- Podemos encontrar desde super eficientes leyendo registros a más pesadas como funciones de librería que usan dispositivos especiales (por ejemplo para medir energía)

■ Ejemplo:

- `gettimeofday`
- `time`
- `getrusage`
- Librerías (por ejemplo PAPI) que nos permiten obtener información de detalle sobre instrucciones de LM ejecutadas

Análisis de rendimiento

- Comandos que normalmente aplican a la ejecución complete de un programa o realizan un muestreo y devuelven información periódica
- Pueden ofrecer una visión global del nodo, incluyendo todos los procesos, o de un único proceso
 - Global: ps, top, free, vmstat, iostat, strace, perf
 - Un único proceso: ps (opción -p pid_number), time, strace, perf
- Aplicada al hardware
 - cpupower

Análisis de rendimiento

- ¿Que tipo de métricas nos ofrecen?
 - **Tiempo de ejecución** (tiempo final- tiempo inicial) (segundos)
 - **Tiempo de cpu**: incluye el tiempo de todos los cores utilizados
 - **Tiempo de usuario/sistema**: tiempo que pasamos ejecutando código del programa o del sistema operativo
 - ▶ Idealmente el tiempo de sistema debería ser mínimo
 - **Latencia**: tiempo que tardan en comenzar las operaciones de transferencia de datos
 - **Energia (Joules), Power (Watts)**
- ¿Qué tipo de métricas derivadas son interesantes?
 - **Speedup**: es la aceleración al ejecutar con N cores
 - **Eficiencia**: $\text{Speedup}(p)/p$, nos da idea de cómo aprovecha nuestro programa los cores adicionales
 - **Bandwidth** (bytes/s)

Análisis de rendimiento

- Estadísticas
 - Average
 - Std deviation
 - Mean
 - etc

PARALELISMO...