Entorno de Desarrollo

Ricardo Sanz

UPM-ASLab

Curso 2005-2006

Entorno de desarrollo

- El entorno de desarrollo trata de:
  - Mejorar el desarrollo de las aplicaciones generales, mediante facilidades que permitan una mayor comodidad y productividad en la programación.
  - Aumentar la seguridad y eficiencia del sistema, mediante facilidades que controlen el acceso a los recursos del sistema, mejoren y faciliten al usuario su uso.
  - Aumentar la capacidad del sistema, mediante la integración de nuevos recursos en el sistema.

Entorno de desarrollo

- El entorno de desarrollo depende de dos elementos:
  - El ***host***: i.e. el computador sobre el que trabajamos
  - El ***target***: I.e. el computador para el que programamos
- Por ejemplo, podemos desarrollar en un compuador con Windows XP una aplicación para VxWorks

Que hay en un entorno

- Recursos fundamentales
- Herramientas
  - Compiladores
  - Ensambladores
  - Enlazadores
  - Depuradores
  - IDEs
  - etc.

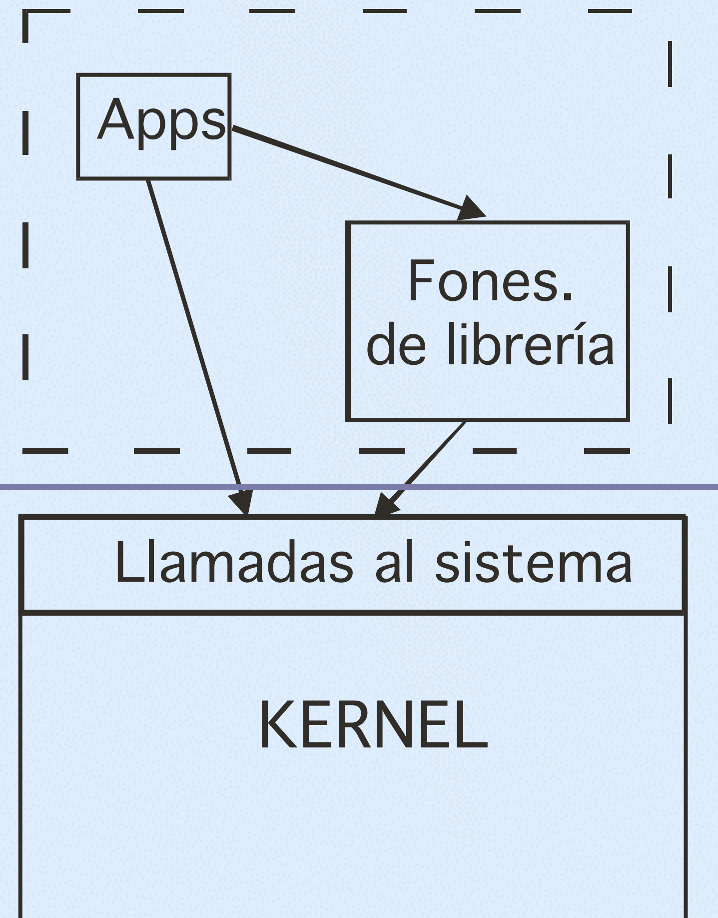
Recursos fundamentales

- Llamadas al sistema
- Funciones de librería
- Se distinguen por:
  - Acceso al kernel o ejecución en el espacio de usuario
  - Funcionalidad
  - “permanencia”o sustituibilidad

Llamadas al sistema/Funciones de librería

Modo Usuario

Modo Kernel



Llamadas al sistema/Funciones de librería

- Ejemplos:
  - sbrk/malloc
  - printf/write
  - Funciones de fecha y hora
  - fork, exec, wait/system

Llamadas al sistema

- Ejemplo de llamada:

count = ***read****(file, buffer, nbyte);*

read → nombre de la llamada al sistema
file → fichero de donde leer
buffer → buffer de datos
nbytes → nº de bytes a leer
count → nº de bytes leídos,

Llamadas al sistema

Ejemplo de uso del open para abrir el fichero especial “/dev/tty” de la consola asociada al proceso

```
# include  <fcntl.h>

int main (int argc, char **argv)  {
/* abrimos la consola */
     int fd = open(“/dev/tty”, O_RDWR);
/* el descriptor de fichero “fd” devuelto por open es
     utilizado por write para escribir */
     write(fd, “hola joven!\n”, 12);
/* cerramos el fichero */
     exit(0);
}
```

Páginas del manual (man)

- Secciones en Linux:
  1 = comandos de usuario
  2 = llamadas al sistema, definiciones en C
  3 = funciones de librería
  4 = ficheros especiales
  5 = ficheros de configuración
  6 = juegos
  8 = comandos administrativos y de mantenimiento

Sintaxis: *man seccion funcion*

Ficheros cabecera

- Puede recurrirse a ellos para buscar funciones y/o ver la sintaxis exacta de las mismas
- Ubicados en */usr/include* en maquinas UNIX*
- Por ejemplo */usr/include/unistd*

Manejo de errores

- En caso de error: Las llamadas al sistema suelen devolver -1 ó NULL (si la función devuelve tipo puntero)
- Se deben siempre comprobar después de una llamada si todo es correcto
- Para ello Linux proporciona una variable llamada *errno* y una función llamada *perror*():

Manejo de errores

Ejemplo de de uso de la variable global **errno** y del procedimiento **perror()**

```
/*  Lista los 53 primeros errores de llamadas al sistema  */
# include <stdio.h>
main(argc,argv)
int argc;
char *argv[];
{
int i;
extern int errno;
  for (i=0;i<=53;i++){
     fprintf(stderr,"%3d",i);
     errno=i;
     perror(" ");
  }
  exit(0);
}
```

Normas y estándares

- ANSI C: Define sintaxis, semántica y librería estándar
- POSIX - Portable Operating System Interface. Define una coleecion de llamadas a sistema y funciones de librería
- Los ficheros de cabecera dan información de grupos de funciones

LINUX tiene alrededor de 200 llamadas al sistema (como UNIX y POSIX)

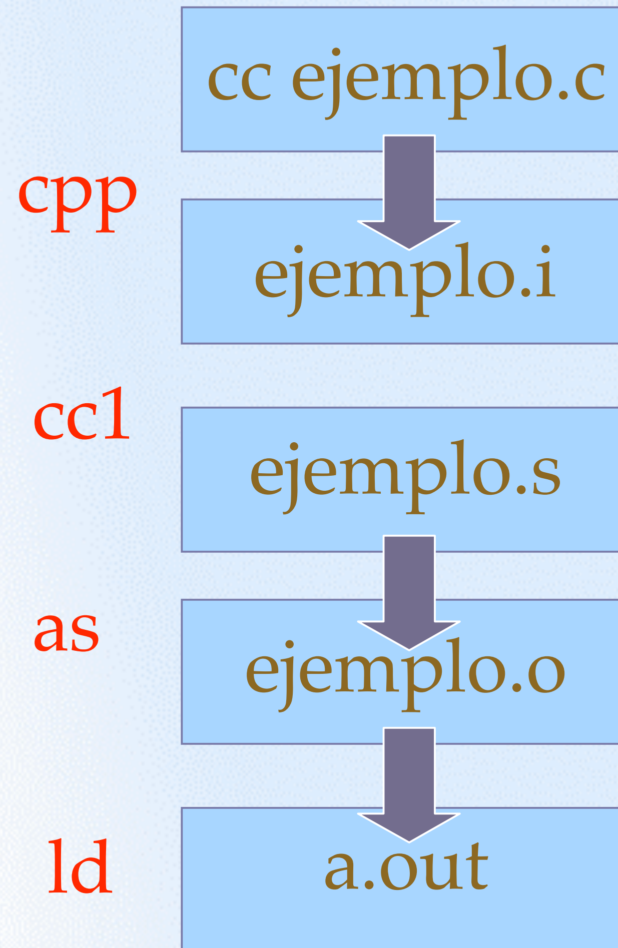
Herramienta Primaria

- Compilador
- *cc* = C Compiler
- *gcc* = GNU C Compiler

- Traduce lenguaje de alto nivel a lenguaje de máquina

Fases de la compilación

gcc



gcc

- Opciones más importantes:
  - -c = sólo compilación
  - -o = nombre de fichero ejecutable
  - -l = librería. Útiles: m, curses
  - -I = directorio de include
  - -L = directorio de librerías
  - -g = incorpora información para depuración

Depurador

- Sintaxis: *gdb ejecutable*
- Opciones más importantes:
  - run, help, quit
  - set args <argumentos>
  - break <nombre_de_funcion>
  - list
  - step, next (no entra en las funciones), continue
  - print <variable>
  - where
- *ddd*: version con ventanas

strace

- Para ver las llamadas al sistema que hace un programa
- Sintaxis: *strace a.out*

make

- Imaginemos un programa complejo compuesto por un conjunto de módulos (cada uno de ellos con sus respectivos ficheros cabecera)
  - pantalla.c ficheros.c bdatos.c principal.c
  - pantalla.h ficheros.h bdatos.h

```
$cc -c pantalla.c
...
$cc -o programa pantalla.o ficheros.o bdatos.o
principal.o

// en un solo paso
$cc -o programa pantalla.c ficheros.c bdatos.c
principal.c
```

make

- Solución:
  - Para no tener que teclear tanto cada vez que se necesita recompilar se puede escribir un *script* del *shell*
- Problema
  - Supongamos que en el ejemplo anterior realizamos una pequeña modificación en el programa principal (principal.c)
  - Cuando lanzamos el *script* se compilaría todo nuevamente

make

- Herramienta que teniendo en cuenta las dependencias entre los módulos sólo compile aquellos que hayan sido modificados
- Programa *make* (o *gmake*)
  - La información de dependencias se escribe en un fichero *makefile*
  - Al ejecutar *make,* se lee dicho fichero y se genera un fichero ejecutable compilando sólo aquellos módulos que hayan sufrido modificaciones

make

- Dependencias de un fichero
  - Conjunto de ficheros que participan directa o indirectamente en su elaboración
- Reglas explícitas
  - El programa *make* lee del fichero *makefile* un conjunto de reglas de construcción de ficheros

```
target: fich1 fich2 .....
        orden1
        orden2
```

make

- Para construir un *target* se escribe:
  - *make target*
- Funcionamiento
  - Se lee el fichero *makefile* del directorio de trabajo y se busca la regla para construir *target*
  - Se aplican las reglas de dependencia, si existen
  - Se ejecutan las órdenes de construcción en caso de que:
    - No existe el objetivo
    - La fecha de ultima modificación de alguna de las dependencias es posterior a la del objetivo

make

- Las macros se utilizan en *make* para abreviar textos que se utilizan en varias partes del fichero *makefile*
- Se declaran en cualquier parte del fichero *makefile*
  - nombre = texto
  - Uso: $(nombre)
  - Se distinguen mayúsculas de minúsculas

make

- Si una regla no tiene dependencias se aplica siempre
- Se pueden escribir reglas pertenecientes a varios proyectos de software en el mismo fichero **makefile**

touch

- Cambia la fecha de última modificación del fichero
- Sintaxis: *touch fichero*
- Propósito: el fichero será recompilado por *make*

ar

- Gestión de bibliotecas: crear, modificar y extraer miembros
- Ejemplos de bibliotecas: *libc.a, libm.a*
- Extensión: *.a*
- Ejemplo:

cc –c complejo.c
ar libcomplejo.a complejo.o
...
cc principal.c libcomplejo.a

Tipos de arcivo

- .c codigo fuente C
- .cpp codigo fuente C++
- .o fichero objeto
- .i preprocesado
- .s ensamblador
- .a librería (biblioteca)
- .so objeto compartido
- .sa librería compartida (.dll)

nm

- Sirve para mostrar el contenido de una biblioteca o módulo objeto. Las funciones que exporta
- Sintaxis: *nm ejecutable*

ldd

- Muestra las bibliotecas compartidas que un ejecutable necesita para funcionar:

```
$ ldd /usr/bin/mutt
libnsl.so.1 => /lib/libnsl.so.1 (0x40019000)
libslang.so.1 => /usr/lib/libslang.so.1
    (0x4002e000)
libm.so.6 => /lib/libm.so.6 (0x40072000)
libc.so.6 => /lib/libc.so.6 (0x4008f000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2
    (0x40000000)
```

Editores

- vi
- emacs
- gedit
- xemacs
- kate
- ...

Entornos Integrados

- Kdevelop
- CodeWarrior
- Visual .NET
- Eclipse
- NetBeans
- XCode
- …

SudokuEvaluator

SudokuEvaluator | Active Target | Action | Build | Build and Go | Tasks | Info | Editor | String Matching | Search

Groups & Files

- SudokuEvaluator
  - Source
  - Documentation
  - Products
- Targets
- Executables
- Errors and Warnings
- Find Results
- Bookmarks
- SCM
- Project Symbols
- Implementation Files
- NIB Files

File Name	Code
main.cpp	
SudokuEvaluator	
SudokuEvaluator.1	

main.cpp

Build | Build and Go | Tasks | Fix | Breakpoints | Project | Ungrouped

main.cpp:1 <No selected symbol>

```
#include <iostream>

int main (int argc, char * const argv[]) {
    // insert code here...
    std::cout << "Hello, World!\n";
    return 0;
}
```

- X
- ..