

Introducción a UNIX

LABORATORIO DE SISTEMAS DE OPERACIÓN I
(ci-3825)

Prof. Yudith Cardinale

Introducción a UNIX

Características generales:

- Multiusuario, multitasking (Tiempo Compartido) y multiconexiones
- Permite la creación, modificación y destrucción de procesos, programas y archivos.
- Provee una jerarquía de directorios y accesos a través de caminos absolutos y caminos relativos.
- Los dispositivos son tratados como archivos, esto crea un estándar que hace extensible a Unix.
- Esquema consistente de protección: dueño - grupo - todos. Es decir, todos los archivos son protegidos con el mismo esquema.
- Provee utilidades estándares y rutinas de librerías
- Provee llamadas al sistema como apoyo al programador que puede usar los servicios del sistema de operación.
- El Shell es programable. Los shell scripts son programas escritos en lenguaje especial de Unix que permiten sintetizar operaciones adaptadas a los requerimientos específicos del usuario.
- Es portable

Introducción a UNIX

Existen dos maneras de interactuar con el SO:

- A través de llamadas al sistema. Las llamadas al sistema son servicios al programador que provee el sistema de operación. Ejemplo de algunas llamadas al sistema:
 - Para Manejo de Procesos: `fork()`, `wait()`, `exit()`, `exec()`, `getpid()`.
 - Para Manejo de Archivos: `open()`, `close()`, `read()`, `write()`, `lseek()`.
- Por medio de órdenes en el Shell . El Shell está por encima del kernel y es un proceso más a nivel de usuario. Es un interpretador de órdenes. Actúa como mediador entre usuario y kernel.
Provee facilidades como: multitasking, pipes, metacaracteres y redireccionamiento.

Qué hace el Shell :

Realiza inicializaciones

Muestra un prompt y espera por una orden de usuario.

Cuando se introduce una orden, la ejecuta y vuelve al paso anterior.

Introducción a UNIX

Funciones del Shell:

- Órdenes built-in: rutinas del shell que son residentes (cd, echo). Existen otras órdenes que son archivos ejecutables almacenados en la jerarquía de directorios con permiso de ejecución (ls,cat,cp)
- Ejecución de Scripts
- Mantener variables locales y de ambiente. (\$HOME,\$SHELL,\$PRINTER, \$USER, \$MAIL, \$PATH) : con la orden **env** se pueden ver las variables definidas
- Redireccionamiento:
command > file , command < file, command >> file
- Metacaracteres: &,#, \$, ?, *, <, >, |.

- Sustitución de órdenes: 'date'
- Secuencias no condicionales: ls;date;
- Secuencias condicionales:
cmd1 && cmd2, cmd1 || cmd2
- Subshells
- Procesamiento en ``background" (& al final de la orden)

Introducción a UNIX

Algunas órdenes de UNIX:

- Primer contacto: login, password y home.
- passwd: cambia tu password
- Sintaxis de comandos: Comando [opciones [parámetros]] [parámetros]
- man: manual en línea. man [command]. man [chaper]
man 2 chmod
- man -k keyword: muestra todas las entradas del manual que contienen keyword.
Ejemplo man -k mode tex2html_wrap_inline129 chmod (1V), chmod (2V)
- date: muestra y cambia la fecha. Sólo el superusuario puede cambiar la fecha
- clear: limpia la pantalla
- pwd: muestra el directorio de trabajo actual.
- cd: para cambiar de directorio
- cat: para editar archivos.
cat > mio
Al terminar de editar ctrl-D.
- Para listar el contenido de un archivo:
cat mio.

Introducción a UNIX

Algunas órdenes de UNIX:

- more, cat, less, head y tail: lista el contenido de un archivo
- ls: listar el contenido de un directorio.
- mv: renombrar o mover archivos.
- mkdir: crear un directorio.
- cp: copiar archivos
- rmdir: borrar un directorio
- rm: borrar archivos
- lpr, lp: imprimir
- lpq, lpstat: ver cola de impresión
- lprm: desencolar un archivo que se envió a imprimir.
- wc -lwc filename
- file filename: describe el tipo de archivo
- chmod: cambia los permisos a un archivo.
chmod o-rw filename u,g,o,a + - rwx
- Otros: find, env, exit, who, **ps**, **top**.
- Caracteres especiales: ctrl-C, ctrl-D, ctrl-S, ctrl-Q y ctrl-Z

Introducción a UNIX

Redireccionamiento en el Shell: Las facilidades de redireccionamiento en el shell permiten:

- Almacenar la salida de un proceso en un archivo (redireccionamiento de salida)
- Usar el contenido de un archivo como entrada a un proceso (redireccionamiento de entrada)

Veamos ejemplos de ambos casos:

- **Redireccionamiento de salida:** Permite guardar en un archivo texto la salida de un proceso. Esta salida puede luego ser editada, listada, impresa o usada como entrada para otro proceso futuro. Para redireccionar la salida se usan los metacaracteres `>` o `>>`.

Las secuencias usadas son: `command > filename`

Se crea un nuevo archivo con ese nombre o sobrescribe el contenido si el archivo ya existía.

```
cat > texto.txt
```

```
ls > archivos.txt
```

```
programa > salida
```

```
command >> filename
```

Se crea un nuevo archivo con ese nombre o se adiciona la salida al contenido anterior del archivo, si éste ya existía.

```
cat >> texto.txt
```

Introducción a UNIX

- **Redireccionamiento de entrada:** Permite pre-preparar la entrada de un proceso y almacenarla en un archivo para su uso posterior. Para redireccionar la salida se usan los metacaracteres `<` o `<<`.

Las secuencias usadas son:

```
command < filename
```

Ejecuta `command` usando el contenido del archivo `filename`.

```
mail yudith < mensaje
```

Introducción a UNIX

Cómo hacer un Makefile

- Compilar un programa en C:

Después de crear un programa en C (con emacs, vi o cat), éste debe compilarse para luego ejecutarlo.

```
void main() {  
    printf(`Hello Word\n");  
}
```

`$> gcc myprogram.c ==>` En este caso el ejecutable se genera en **a.out**

Para ejecutar el programa: `$> ./a.out`

Otra forma de compilación: **gcc myprogram.c -o myprogram**

En este caso el ejecutable es **myprogram**

Para ejecutar el programa sólo se indica el nombre del ejecutable:

`$> ./myprogram`

Introducción a UNIX

Cómo hacer un Makefile

- Compilar un programa conformado por varios módulos:

Cuando un programa está formado por varios módulos separados es necesario compilar cada módulo, enlazarlos y crear el ejecutable. Estos módulos independientes pueden ser reusables en otros programas.

Por ejemplo si tenemos los siguientes archivos:

```
/* reverse.h */  
  
void reverse (char *before, char *after);
```

```
/* reverse.c */  
  
#include <stdio.h>  
#include ``reverse.h''  
  
void reverse (char * before, char *after)  
{  
    /* codigo de la funcion */  
  
}
```

```
/* main1.c */  
  
#include <stdio.h>  
#include ``reverse.h''  
  
main ()  
{  
    char str[100];  
  
    reverse(``cat",str);  
    reverse(``ylayaly",str);  
}
```

Compilación:

```
$> gcc -c reverse.c
```

```
$> gcc -c main1.c
```

Alternativamente:

```
$> gcc -c reverse.c main1.c
```

En ambos casos la compilación genera **reverse.o** y **main1.o**. Luego para **enlazarlos** en un ejecutable ``main1" se ejecuta:

```
$> gcc reverse.o main1.o -o main1
```

Para ejecutarlo: \$>./main1

Introducción a UNIX

Cómo hacer un Makefile

- Otro ejemplo con más archivos:

```
/* reverse.h */  
  
void reverse (char *before, char *after);
```

```
/* reverse.c */  
  
#include <stdio.h>  
#include ``reverse.h''  
  
void reverse (char * before, char *after)  
{  
    /* codigo de la funcion */  
}
```

```
/* main2.c */  
  
#include <stdio.h>  
#include ``palindrome.h''  
  
main ()  
{  
    char str[100];  
    ...  
    palindrome(``cat");  
    palindrome(``ylyaly");  
}
```

```
/* palindrome.h */  
  
int palindrome (char *str);
```

```
/* palindrome.c */  
  
#include <stdio.h>  
#include ``palindrome.h''  
#include ``reverse.h''  
  
int palindrome (char *str);  
  
{  
    /* codigo de la funcion */  
}
```

Compilación:

```
$> gcc -c palindrome.c reverse.c main2.c
```

```
$> gcc reverse.o palimdrome.o main1.o -o main2
```

Para ejecutarlo: \$>./main2

Introducción a UNIX

Cómo hacer un Makefile

- Compilación usando Makefiles:

Un Makefile es un archivo que contiene las interdependencias que existen entre los archivos que son usados para generar un ejecutable. Las reglas del ``makefile" son de la forma:

```
targetList dependencyList  
  <TAB> commandList
```

Donde:

targetList es el nombre del archivo al cual se le están definiendo las dependencias

dependencyList lista de los archivos del cual *targetList* depende.

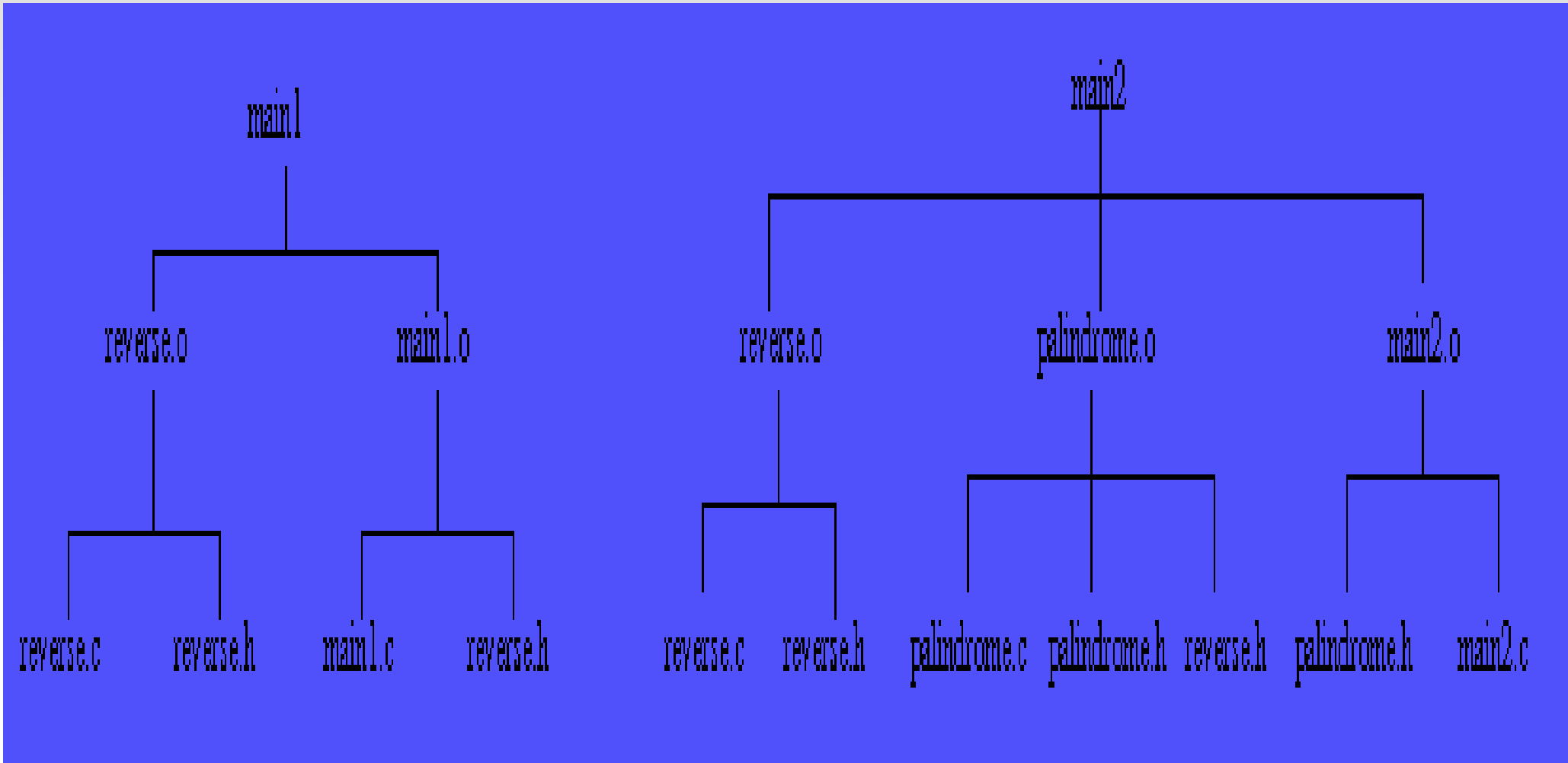
commandList es una lista de cero o más comandos, separados por ``newline", que reconstruyen el *targetList* a partir de las dependencias.

El orden cómo se definen las reglas es muy importante, debe crearse siguiendo un árbol de interdependencias.

Introducción a UNIX

Cómo hacer un Makefile

- Árboles de interdependencias de los ejemplos anteriores:



Introducción a UNIX

Cómo hacer un Makefile

- Makefiles de los ejemplos anteriores:

```
#Makefile de main1
main1: main1.o reverse.o
    gcc main1.o reverse.o -o main1
main1.o: main1.c reverse.h
    gcc -c main1.c
reverse.o: reverse.c reverse.h
    gcc -c reverse.c
```

==> Este archivo se llama makefile

Se ejecuta **\$> make** (automáticamente busca en . un archivo llamado makefile y lo ejecuta)

```
# Makefile de main2
main2: main2.o reverse.o palindrome.o
    cc main1.o reverse.o palindrome.o -o main2
main2.o: main2.c palindrome.h
    cc -c main2.c
reverse.o: reverse.c reverse.h
    cc -c reverse.c
palindrome.o: palindrome.c palindrome.h reverse.h
    cc -c palindrome.c
```

==> Este archivo se llama main2.make

Se ejecuta:

\$> make -f main2.make