Prácticas de Sistemas operativos

David Arroyo Guardeño

Escuela Politécnica Superior de la Universidad Autónoma de Madrid

Primera Semana: Shell + Ficheros

1 *Introducción*

2 *Normativa*
- *General*
- *Convalidaciones*
- *Sistemas de evaluación*
- *Requisitos de las entregas*
- *Estilo de programación*
  - *Makefile*
  - *gdb*

3 *Primera práctica*
- *Introducción a la shell*
- *Descriptores de ficheros*

Información general

- ✓ Correo $\leftrightarrow$ xxx.yyy@estudiante.uam.es
- ✉ [SOPER]
- ✓ Nota: $0.7 \times N_T + 0.3 \times N_P$
- ☠ COPIAS

Convalidaciones

- ✗ Prácticas 2013-14 aprobadas $\rightarrow N_P = 5$
- ✉ `david.arroyo@uam.es` $\rightarrow$ ASUNTO: [SOPER-PRACTICAS-CONVALIDAR]
- ✗ Hasta viernes 20 de febrero

Sistemas de evaluación

1. Evaluación continua
2. Evaluación *tradicional*

Evaluación continua

- ✓ Entregas dentro de plazo
- ☠ Número máximo de faltas de asistencia: 1
- ✓ Día Entrega $\rightarrow$ Examen de prácticas
  - 🕡 30 minutos
- ✓ Calificación
  - ✗ $N_{P_i} = 0.3 \times N_{Examen} + 0.7 \times N_{Entrega}$
  - ✗ $N_{Pf} = \frac{\sum_{i=1}^{4} N_{P_i}}{4}$

Evaluación tradicional

- ✓ Entregas dentro de plazo
- ☠ Examen final de prácticas
- ✓ Calificación
  - ✗ $N_{Entregas} = \frac{\sum_{i=1}^{4} N_{P_i}}{4}$
  - ✗ $N_f = (N_{Entregas} + N_{Examen})/2$
  - ☠ $N_{Entregas} \geq 5,\ N_{Examen} \geq 5$

☠☠ *Evaluación extraordinaria* ☠☠

- ✓ Entregar TODAS las prácticas con TODOS los optativos
- ✓ Examen final de prácticas

Requisitos de las entregas

⚠

Retraso $N_{Ef} = 0.8^{n_d} \times N_E,$

$n_d \equiv$ Número de días de retraso

Entrega moodle Límite: una hora antes del comienzo de la primera sesión de la siguiente práctica

Nombre entregable Gxxx_Pyy_z.tgz

en caso contrario, $\downarrow$ 1 punto

Contenido del fichero

1. Fichero de texto: nombre alumnos, email, grupo y fecha
2. Documentación en formato pdf
   - ✗ Respuesta a preguntas breves
   - ✗ Cómo se ha realizado cada ejercicio
   - ✗ Documentación ejercicio final: análisis del problema, algoritmos utilizados, módulos, estructuras de datos, pruebas realizadas
3. Listado de código fuente documentado
   - ✗ Programación modular y estructurada
   - ✗ Makefile
   - ☠ Compilación no correcta: 0!!!

Introduction to coding style

- ✓ Read carefully the coding standard document
- ✓ Read the set of mandatory rules for coding item Recode a supplied code according to the described coding standard

The unexpected relationship between writing code and writing

- ✓ Even more complicated than writing programs that a computer can understand
- ✓ Write programs that can be understood by both the computer and your fellow programmers

Anterior Siguiente 1 (1 de 15) Ajustar al ancho de página

Literate Programming

Donald E. Knuth
Computer Science Department, Stanford University, Stanford, CA 94305, USA

The author and his associates have been experimenting for the past several years with a programming language and documentation system called WEB. This paper presents WEB by example, and discusses why the new system appears to be an improvement over previous ones.

A. INTRODUCTION

The past ten years have witnessed substantial improvements in programming methodology. This advance, carried out under the banner of "structured programming," has led to programs that are more reliable and easier to comprehend; yet the results are not entirely satisfactory. My purpose in the present paper is to propose another motto that may be appropriate for the next decade, as we attempt to make further progress in the state of the art. I believe that the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be *works of literature*. Hence, my title: "Literate Programming."

I would ordinarily have assigned to student research assistants; and why? Because it seems to me that at last I'm able to write programs as they should be written. My programs are not only explained better than ever before; they also are better programs, because the new methodology encourages me to do a better job. For these reasons I am compelled to write this paper, in hopes that my experiences will prove to be relevant to others.

I must confess that there may also be a bit of malice in my choice of a title. During the 1970s I was coerced like everybody else into adopting the ideas of structured programming, because I couldn't bear to be found guilty of writing *unstructured* programs. Now I have a chance to get even. By coining the phrase "literate programming," I am imposing a moral commitment

THE ELEMENTS OF PROGRAMMING STYLE

SECOND EDITION

Kernighan and Plauger

C program layout: C program file

1. Header comment
2. #included files: first local, then global
3. #defines
4. local struct typedefs
5. local prototypes
6. global vars (☠)
7. main function (if present)
8. local functions

C header (.h) file

1. Header comment
2. #ifndef guard
3. #included files
4. #defines
5. struct typedefs
6. function declaration prototypes
7. (extern) global vars

Header comment

```
/**
 * fileName    Reglas.c
 * author darg

 * date    Tue Feb  5 00:52:35 2013
 *
 * brief  What this file is for (one line summary).
 *
 *
 */
```

Names

what	convention	example
functions	lowercase, more than one word names: use _	select_output_templ
variables		n_patterns
anonymous variables	single lowercase letter	i++;
new typedefed types	CamelCase, lowercase first letter unless abstract, uppercase first letter if ADT	typedef struct _dialogueRules DialogueRules;
constants	UPPER_CASE, words joined with _	#define MAX_LENGTH 5

Layout

- ✓ Use spaces for indents, not tabs
- ✓ Indent 3 spaces OR 4 spaces per level, but not a mix in any file
- ✓ Describe block structures using indentation, do not rely on { } alone
- ✓ MAY include one space before and/or after parentheses delimiting function parameter list
- ✓ Use compact placement

```
void process_input (int age) {
        printf ("Process!!!!!");
}
```

if, while I

- always use { }, even for single statement blocks
- place opening brace compactly (after condition rather than on following line)

```
while (cond == 0) {
     ....;
 }
```

- use compact braces around *else*

```
if (cond == 0) {
     ....;
     ....;
 } else {
     ....;
 }
```

if, while II

- ✓ do not further indent chained else if and else statements

```
if (cookies == 0) {
    ....;
} else if (cookies < 10) {
    ....;
} else if (cookies < 50) {
    ....;
} else {
    ....;
}
```

Declarations I

- ✓ Declare variables at the top of the function
- ✓ Declare one per line if initialised. Only use comma separated lists of variables in a declaration when they are related instances (e.g., int row, col;)
- ✓ In a block of related definitions vertically align the names, types, and initialisations
- ✓ initialise variables close to where they are first used.

Declarations II

- all structs and enums defined with typedefs
- place prototypes for functions which are only used locally at head of file
- place typedefs for types which are only used locally at head of file
- prototypes and typedefs for functions and types which are used in multiple files are placed in a meaningfully named .h file, which is included in the file in which the

Declarations III

function is defined and also in each file which uses the function or type

✓ don’t #include a .h file in another .h file

Functions

- ✓ put the return type on same line as function name
- ✓ function bodies should be kept short, and broken into sub-functions as required
- ✓ each function should have a single clearly defined purpose, this purpose should be reflected in its name

- ✓ First thing inside a function: verify input arguments!!!!!!!!
- ✓ Avoid the use of MACROS
- ✓ Try to ellude global variables: they are a source of side effects
- ✓ Consistent naming of variables, constants, and functions
- ✓ Within the code comments are best kept short and used to document unusual things

gcc compiler

- Given main.c generate the executable program basico1
- Generate calculator from main.c and functions.c

gcc compiler

- ✓ Given main.c generate the executable program basico1
  - ✗ gcc main.c -o basico1
  - ✗ Execution → ./basico1
- ✓ Generate calculator from main.c and functions.c

gcc compiler

- ✓ Given main.c generate the executable program basico1
- ✓ Generate calculator from main.c and functions.c
  1. gcc -ocalculator main.c functions.c
  2. Compiling & Linking
     - (a) gcc -c main.c
     - (b) gcc -c functions.c
     - (c) gcc -ocalculator main.o functions.o

Important gcc flags

-ansi	To enforce ANSI C standards
-pedantic	To issue all the warnings demanded by strict ISO C
-Wall	Enables all the warnings about constructions
-g	Generates debug information to be used by GDB debugger
-l$<$ *library* $>$	links with a library file
-L	looks in directory for library files
-c	Compiles source files without linking
-I	adds include directory of header files
-O	Set the compiler's optimization level
-help	Show help

Make

1. Macros declarations
   - ✗ Global constants
2. Targets
   - ✗ A target includes a set of dependencies
3. Dependencies
   - ✗ If one dependence does not exist, make looks for a target to generate it

Structure of a makefile

```
#Macros declaration
MACRO1=value

#Rules declarations

target1 : dependence1 dependence2 dependenceN
    command1 dependence1 dependence2
    command2 dependence2 dependenceN
```

makefile script I

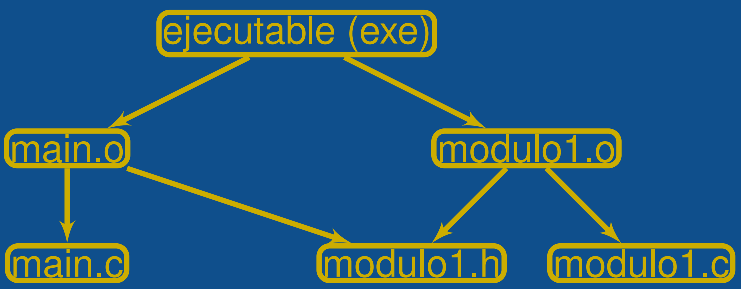
```
CC = gcc
CFLAGS = -O
OBJS = part1.o part2.o
libclassdll: $(OBJS)
	ar -rv libclassdll.a $(OBJS)
myprogram: $(OBJS)
	$(CC) -o myprogram $(CFLAGS) main.o -L. -lclassdll
part1.o: part1.c part1.h header.h
	$(CC) $(CFLAGS) -c part1.c
part2.o: part2.c header.h
	$(CC) $(CFLAGS) -c part2.c
main.o: main.c header.h
	$(CC) $(CFLAGS) -c main.c
clean:
	rm -f libclassdll.a $(OBJS) main.o
	@echo "all cleaned up!"
```

Local macros

```
$^    All the dependencies of a rule
$<    First dependence of a rule
$@    Target of a rule


executable: module1.o module2.o module3.o
            gcc -o$@ $^
instead of
 gcc -oexecutable moudule1.o module2.o module3.o
```

```
CC = gcc
CFLAGS = -g -Wall -O3
BUILD = ./build/
DIST = ./dist/
TEST = ./test/

EXECUTABLES = $(DIST)ex1 $(DIST)ex2
CLIENTS = $(TEST)tests

all: $(EXECUTABLES) $(TESTS)

$(DIST)ex1: $(BUILD)dep1.o $(BUILD)dep2.o

...

$(BUILD)dep1.o: dep1.c
	@echo Generating $@
	$(CC) $(CFLAGS) -c $< -o $@
```

Ejemplo



Makefile

3 Reglas

```
ejecutable: main.o modulo1.o
	gcc -o ejecutable main.o sum.o

main.o: main.c modulo1.h
	gcc -c main.c

modulo1.o: modulo1.c modulo1.h
	gcc -c modulo1.c
```

*.o: *.c dependencias

- ✓ Un fichero *.o depende (por defecto) del correspodiente fichero *.c
- ✓ Ejemplo: foo.o: foo.h
  - ✗ Acción implícita: **$(cc) −c foo.c −o foo.o**

```
ejecutable: main.o modulo1.o
     gcc -o ejecutable main.o modulo1.o
#ERROR: main.o no se compila aunque main.c sea más reciente!!!

main.o:  modulo1.h
     gcc -c main.c

modulo1.o: modulo1.c modulo1.h
     gcc -c modulo1.c
```

*.o: *.c dependencias

- ✓ Un fichero *.o depende (por defecto) del correspodiente fichero *.c
- ✓ Ejemplo: foo.o: foo.h
  - ✗ Acción implícita: **$(cc) –c foo.c –o foo.o**

```
ejecutable: main.o modulo1.o
	gcc -o ejecutable main.o modulo1.o


main.o: main.c modulo1.h
	gcc -c main.c

modulo1.o: modulo1.c modulo1.h
	gcc -c modulo1.c
```

Makefile equivalentes

```
OBJS=main.o modulo1.o

ejecutable: $(OBJS)
	gcc -o $@ $(OBJS)

%.o: %.c modulo1.h
	gcc -c $*.c
```

- **OBJS**: definición de variable
- **$(OBJS)**: obetenemos el valor de la variable
- **%.o**: patrón de regla
- **$*.c**: evalúa el correspondiente fichero *.c

Makefile: otro ejemplo

```
SRC = ../src
SRC_FILES = $(SRC)/foo.c $(COMMON_DIR)/bar.c main.c
OBJ_FILES = $(patsubst %.c,%.o,$(SRC_FILES))
CFLAGS  =  -Wall -g -ansi
CC = gcc
INCLUDES = -I$(SRC)

ejecutable: $(OBJ_FILES)
	$(CC) $(CFLAGS) $(OBJ_FILES) -lm -o $@
%.o:%.c
	$(CC) $(CFLAGS) $(INCLUDES) -c $*.c -o $*.o

clean:
	-rm $(OBJ_FILES) ejecutable
depend:
	@echo -e '\n' >> makefile
	$(CC) $(INCLUDES) -MM $(C_FILES) >> makefile
```

$(patsubst patrón,sustituto,texto): Función para la sustitución de patrones en el texto dado como tercer argumento

Depuración

1. Compile with the flag -g
2. ulimit -a $\rightarrow$ to check restrictions
3. ulimit -c unlimited $\rightarrow$ to disable restrictions
4. gdb name-executable
   - run $<$ *input – arguments* $>$
   - Add a break-point
     - b $<$ *file_name.c* $>$: *line_number*
     - c $\rightarrow$ continuing execution
     - n $\rightarrow$ next code line
     - s $\rightarrow$ show next code line or function (line by line)
   - bt $\rightarrow$ print a backtrace of the entire stack
   - p $<$ *variable* $>$ $\rightarrow$ print variable content

Documentación del código

- ✓ Cabeceras de los ficheros
- ✓ Funciones comentadas con sus correspondientes parámetros
- ✓ Comentarios en líneas de código si procede
- ✓ Librerias y dependencias utilizadas
- ✓ Descripción de las estructuras de datos principales, variables globales, exportación de datos

1 *Introducción*

2 *Normativa*

- *General*
- *Convalidaciones*
- *Sistemas de evaluación*
- *Requisitos de las entregas*
- *Estilo de programación*
  - *Makefile*
  - *gdb*

3 *Primera práctica*

- *Introducción a la shell*
- *Descriptores de ficheros*

Semana 1 Intro. Shell + Expresiones Regulares + Descriptores de Ficheros

Semana 2 Procesos + familia de funciones exec

Semana 3 Comunicación entre procesos mediante tuberías: entrega final

Introducción a la shell I

- Intérprete de comandos
- Arquitectura UNIX
  1. Hardware
  2. Núcleo (*kernel*)
  3. Programas estándar: ls, grep, cat, vi, etc.
  4. Aplicaciones. Ejemplo: gcc -o programa programa.c -lm
     1. Preprocesador de C (cpp)
     2. Compilador (comp)
     3. Ensamblador (as)
     4. Enlazador (ld)
- Distintos tipos

Introducción a la shell II

- ✗ bash
- ✗ sh, c-shell, tc-shell, etc.

✓ UNIX: todo son ficheros!!!

Utilidades accesibles vía shell

- Ayuda
- Navegación y exploración de archivos
- Visualización de ficheros
- Gestión de procesos y sistema
- Gestión de ficheros
- Redireccionamiento del flujo de datos

Ayuda → man

- ✓ man man
- ✓ man -k <palabra_clave>
- ✓ man n_sec <orden sobre la que se desea obtener ayuda>
  - ✗ Ej.: man passwd

Navegación

- ✓ Cambiar de directorio: cd
  - ✗ /
  - ✗ ~
  - ✗ ./
  - ✗ ../
- ✓ Listar el contenido de un directorio: ls
  - ✗ -l
  - ✗ -a
- ✓ Buscar ficheros
  - ✗ find
  - ✗ locate

Visualización de ficheros

- ✓ Concatenar e imprimir ficheros: cat
- ✓ Imprimir las primeras líneas de un fichero: head
- ✓ Imprimir las últimas líneas de un fichero: tail
- ✓ Visualizador de archivos: less

Gestión de procesos y sistema

- ✓ Mostrar información sobre procesos: top
- ✓ Mostrar el estado de los procesos: ps
  - ✗ -a
  - ✗ -l
  - ✗ -e
  - ✗ -f
- ✓ Mostrar espacio disponible en disco: df
- ✓ Mostrar el tamaño de un cierto fichero: du
- ✓ Consumo de memoria: free

Trabajo con ficheros

- ✓ Encontrar un patrón en un fichero: grep
- ✓ Contar el número de palabras en un fichero: wc
- ✓ Ordernar las líneas de un fichero: sort
- ✓ Eliminar filas repetidas de un fichero: uniq

Redireccionamiento del flujo de datos I

- Enviar a segundo plano un proceso: escribir detrás del comando &
- Tubería: $comando_1 | comando_2$
  - Redirige la salida del $comando_1$ a la entrada del $comando_2$
  - sort file | uniq | wc -l
- Redireccionar la salida estándar a un fichero: $>$
  - sort file | uniq $>$ file_uniq
- Redireccionar la entrada estándar de forma que se lee la información desde un fichero: $<$

Redireccionamiento del flujo de datos II

- ✓ Redireccionar la sálida estándar añadiendo al final de un fichero: $>>$
- ✓ Redireccionar la sálida estándar de error a un fichero: 2 $>$

<table>
<tr><td>*</td><td>Elemento precedente debe aparecer 0 o más veces</td></tr>
<tr><td>+</td><td>Elemento precedente debe aparecer 1 o más veces</td></tr>
<tr><td>.</td><td>Cualquier carácter excepto salto de línea</td></tr>
<tr><td>?</td><td>El elemento precedente es opcional</td></tr>
<tr><td>|</td><td>Operador binario: aparece uno u otro elemento</td></tr>
<tr><td>^</td><td>Comienzo de línea</td></tr>
<tr><td>[. . .]</td><td>Caracteres admitidos</td></tr>
<tr><td>[^. . .]</td><td>Caracteres no admitidos</td></tr>
<tr><td>$</td><td>Fin de línea</td></tr>
<tr><td>-</td><td>Conjunto de caracteres: [a-zA-Z0-9]</td></tr>
</table>

Ejemplos de expresiones regulares

- ✓ man -k read | grep ^read
- ✓ grep -r eth0 /etc/*
- ✓ grep -r etho /etc/* 2> salida.txt
- ✓ ps aux | grep xterm | grep -v grep
- ✓ ps aux | grep '[x]term'
- ✓ ps aux | grep 'xterm$'

Descriptores de ficheros

- Descriptor de fichero: número entero que identifica un fichero asociado a un proceso
- Una tabla de descriptores de fichero por proceso
- Entrada estándar (stdin): 0
- Salida estándar (stdout): 1
- Salida estándar de error (stderr): 2
- Funciones: open, close, read, write