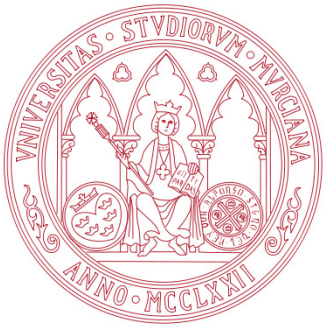

Universidad de Murcia
Facultad de Informática

Título de Grado en
Ingeniería Informática

Ampliación de Sistemas Operativos

Tema 1: Interfaz de un Sistema Operativo

Boletín 1: Implementación del *shell* `simplesh`

Curso 2017 / 18

Departamento de Ingeniería y Tecnología de Computadores
Área de Arquitectura y Tecnología de Computadores

Índice general

1 Boletines de prácticas

B1.1. El *shell* `simplesh`

B1.1.1. Introducción

Esta sección describe el *shell* `simplesh` que se debe modificar y extender en las prácticas de Ampliación de Sistemas Operativos. Este *shell* es capaz de leer y procesar una línea de órdenes de entrada desde el teclado que contenga uno o varios comandos.

Los comandos, con sus correspondientes argumentos, pueden estar delimitados por "`;`" o por "`|`". En el primer caso, los comandos se ejecutarán secuencialmente uno después de otro. En el segundo caso, la salida estándar de cada comando se redirigirá a la entrada estándar del siguiente comando. Además, `simplesh` permite la creación de *subshells* delimitados por "`(`" y "`)`". A su vez, cada *subshell* puede incluir una secuencia de comandos como la descrita anteriormente. La entrada y la salida estándar de un comando se pueden redirigir desde o hacia un fichero mediante "<" y "<" o ">>". Por último, un comando se puede ejecutar en segundo plano con "`&`". Por ejemplo:

```
$ ( cat /etc/lsb-release | grep DESCRIPTION | cut -d = -f 2 ) > distribucion &
```

La función `main` de `simplesh` consiste en un bucle que lee cada línea de órdenes introducida por el usuario, la procesa identificando todos los comandos y demás elementos arriba descritos y, por último, la ejecuta.

B1.1.2. Lectura de la línea de órdenes

El *prompt* de `simplesh` utiliza la librería `readline()` que proporciona la funcionalidad básica de edición, así como un historial de comandos desde el inicio de la sesión. En concreto, la función `get_cmd()` devuelve una cadena de caracteres que contiene la línea de órdenes tecleada por el usuario, sin incluir el retorno de carro.

B1.1.3. Procesamiento y ejecución de la línea de órdenes (*parsing*)

Tras haber leído una línea de órdenes con `get_cmd()`, `simplesh` llama a la función `parse_cmd()` que la procesa reconociendo cada uno de los comandos así como las listas, las tuberías, los *subshells*, las redirecciones o la ejecución en segundo plano. `parse_cmd()` llama a su vez a la función `parse_line()` que realiza el análisis sintáctico de los comandos de manera recursiva con la ayuda de las funciones `parse_pipe()`, `parse_exec()`, `parse_subs()` y `parse_redr()`. Este proceso genera una estructura de datos en forma de árbol que determina de manera inherente el orden de ejecución de los comandos.

La ejecución de la línea de órdenes la lleva a cabo la función `run_cmd()`, de manera recursiva, a partir de la estructura de datos construida por la función `parse_cmd()` en el paso anterior.

B1.1.4. Estructuras de datos

En la estructura de datos arbórea construida por la función `parse_cmd()`, cada comando está representado por la estructura `struct execcmd`. Además, existen estructuras adicionales que representan las redirecciones (`struct redircmd`), las tuberías (`struct pipecmd`), las listas (`struct listcmd`), la ejecución en segundo plano (`struct backcmd`) o los *subshells* (`struct subscmd`).

B1.1.5. Depuración

El código de `simplesh` proporciona funcionalidad básica de depuración, que ayuda a entender como se ejecutan las líneas de órdenes (véase `DBG_CMD`), y un mecanismo extensible para seguir la secuencia recursiva de llamadas a funciones (véase `DBG_TRACE`).

B1.1.6. Compilación

Para compilar `simplesh` se necesita la librería `libreadline` y los ficheros de cabecera `readline/readline.h` y `readline/history.h`. En Ubuntu, estos dos ficheros está incluidos en los paquetes `readline-common` y `libreadline-dev`, respectivamente.

B1.2. El depurador gdb

El depurador gdb (GNU Debugger) es un programa que permite depurar otros programas escritos en cualquier lenguaje de programación que hayan sido compilados con el compilador gcc (GNU Compiler Collection, antes GNU C Compiler). Depurar significa poder ejecutar el programa de manera que permita controlar y estudiar el flujo de ejecución, la pila de llamadas, los valores de las variables, etcétera.

Para ilustrar el uso de gdb, vamos a introducir un hipotético programa en C que contiene errores.

B1.2.1. El programa

Un programador principiante en C ha escrito el siguiente programa para crear un buffer de cadenas de caracteres. El programa contiene dos errores. Uno que se podría considerar de escritura (el límite superior del for, al que se le ha añadido un cero de más), y otro de concepto (los punteros internos del buffer, a los que no se ha asignado memoria).

```
1  #include <stdio.h>
   #include <stdlib.h>
3  #include <string.h>

5  char** inicializa()
   {
7      char** tmp;
       int i;
9
       tmp = malloc(2000 * sizeof (char *));
11     for(i = 0; i < 20000; i++)
           tmp[i] = 0;
13
       return tmp;
15 }

17 void copia(char** buffer)
   {
19     strcpy(buffer[0], "ASO");
   }
21
   int main(void)
23 {
       char** buffer;
25
       buffer = inicializa();
27     copia(buffer);
       printf("%s\n", buffer[0]);
29
       return EXIT_SUCCESS;
31 }
```

B1.2.2. El indicio

Un intento de ejecución del programa nos da un resultado inesperado:

```
usuario@maquina:~$ ./depurame
Violación de segmento (core generado)
```

Este error suele ser un indicio bastante claro de que el programa no funciona correctamente. Indica que el programa ha accedido a una posición de memoria no válida, lo que provoca una violación de segmento. Además, se crea un fichero adicional llamado core cuya utilidad explicaremos más adelante.

B1.2.3. El estudio

Para encontrar el o los problemas en el código, utilizaremos gdb. Para que se pueda utilizar el depurador gdb, se debe usar la opción de compilación -g (también -ggdb3). Así, para generar nuestro programa:

```
usuario@maquina:~$ gcc -g -o depurame depurame.c
```

A continuación, para depurar nuestro programa con `gdb`:

```
usuario@maquina:~$ gdb ./depurame
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
Para las instrucciones de informe de errores, vea:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Leyendo símbolos desde ./depurame...hecho.
(gdb) r
Starting program: ./depurame
Program received signal SIGSEGV, Segmentation fault.
0x00000000004005b0 in inicializa () at depurame.c:12
12              tmp[i] = 0;
```

Aquí ya vemos que el programa falla en la línea 12. Sin embargo, vayamos viendo las características de `gdb` ejecutando el programa paso a paso.

El primer concepto importante en `gdb` es la posibilidad de incluir puntos de ruptura (*breakpoints*). Podemos incluir uno en la función main de la siguiente forma:

```
(gdb) b main
Punto de interrupción 1 at 0x4005e9: file depurame.c, line 26.
```

Si ejecutamos el programa de nuevo, `gdb` se detendrá en la función especificada:

```
(gdb) r
Starting program: ./depurame
Breakpoint 1, main () at depurame.c:26
26          buffer = inicializa();
```

Para continuar con la ejecución, tenemos dos posibilidades: la orden `n`, para que pase a la siguiente línea sin entrar en la función (aunque la llamada sí se ejecuta), o la orden `s`, que sigue con la ejecución dentro de la función. Como sabemos que el código falla dentro de la función:

```
(gdb) s
inicializa () at depurame.c:10
10          tmp = malloc(2000 * sizeof (char *));
```

Al pulsar `n`, se ejecuta la llamada a `malloc`:

```
(gdb) n
11          for(i = 0; i < 20000; i++)
```

Entonces, podemos imprimir el valor de `tmp`:

```
(gdb) print tmp
$1 = (char **) 0x602010
```

Y también listar el código con `l`:

```
(gdb) l
6       {
7           char** tmp;
8           int i;
9
10          tmp = malloc(2000 * sizeof (char *));
11          for(i = 0; i < 20000; i++)
12              tmp[i] = 0;
13
14          return tmp;
15      }
```

La línea que se va a ejecutar a continuación (11) se muestra en el centro del listado. Una inspección rápida nos dice que hay un error en el límite superior del `for`. Para evitar este tipo de errores, resulta conveniente utilizar constantes, por ejemplo, incluyendo `#define BUFFER_LENGTH 2000` al principio del fichero.

B1.2.4. Depuración *post-mortem*

El programa contenía dos errores. Por tanto, al arreglar el fallo detectado anteriormente, y volver a ejecutar el programa, fallará de nuevo en la línea en que se llama a la función `strcpy`:

```
(gdb) r
Starting program: ./depurame
Program received signal SIGSEGV, Segmentation fault.
0x00000000004005e7 in main () at depurame.c:19
19          strcpy(buffer[0] , "ASO");
```

Depurar los programas una vez que han fallado resulta muy útil para localizar los errores. Además, es lo que normalmente se hace, porque depurar paso a paso un programa hasta encontrar el error puede resultar muy tedioso. Tras este mensaje, podemos pedirle a `gdb` que nos muestre la pila de llamadas con `backtrace`:

```
(gdb) backtrace
#0  0x00000000004005d9 in copia (buffer=0x602010) at depurame.c:19
#1  0x0000000000400603 in main () at depurame.c:27
```

Este listado indica que la función `copia` se quedó ejecutando la línea 19 que corresponde a `strcpy`. La pila de llamadas guarda todas las funciones que han sido invocadas por el programa. En cualquier momento podemos seleccionar una función de la pila, lo que nos permitiría explorar las variables locales de esa función. En este caso nos interesa `copia`. Para seleccionarla, utilizaremos la orden `frame`:

```
(gdb) frame 0
#0  0x00000000004005d9 in copia (buffer=0x602010) at depurame.c:19
21          strcpy(buffer[0], "ASO");
```

Si imprimimos el valor de `buffer[0]`, veremos que ha sido el causante del error:

```
(gdb) print buffer[0]
$2 = 0x0
```

El error se debe a que `buffer[0]` es un puntero nulo. Se puede resolver el problema reservando memoria para `buffer` en la función `inicializa()`, es decir, con una llamada a `malloc` para cada elemento de `tmp`.

Otra forma de conseguir una depuración post-mortem es utilizando el fichero `core`. Cuando el programa falla, produce este fichero que es un volcado de memoria del programa en ejecución:

```
-rw------- 1 usuario usuario 405504 sep 16 17:30 core
-rwx------ 1 usuario usuario   9991 sep 16 17:30 depurame
```

Si este fichero no es generado automáticamente, se puede obligar al sistema a hacerlo escribiendo este comando antes de ejecutar un programa: `ulimit -c 1024` (para generar un fichero `core` de hasta 1MB).

Ejecutando `gdb` con la opción `-core`, conseguimos lo mismo que ejecutándolo con `r` y esperando a que se detenga, es decir, dejar el programa justo en el punto de fallo:

```
usuario@maquina:~$ gdb -core core
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
Para las instrucciones de informe de errores, vea:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
[Nuevo LWP 31713]
El núcleo se generó por «./depurame».
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x00000000004005d9 in ?? ()
```

A continuación, con el comando `file` le indicamos a GDB de qué fichero ejecutable debe leer los símbolos de depuración:

```
(gdb) file ./depurame
Leyendo símbolos desde depurame...hecho.
```

Finalmente, podemos continuar la depuración post-mortem igual que antes:

```
(gdb) backtrace
#0  0x00000000004005d9 in copia (buffer=0x1125010) at depurame.c:19
#1  0x0000000000400603 in main () at depurame.c:27
```

B1.2.5. Resumen de órdenes de `gdb`

Orden	Abrev.	Significado
`run [<argumentos>]`	r	Ejecuta el programa desde el principio pasándole opcionalmente los argumentos como si le fueran dados en la línea de comandos.
`print <expresión>`	p	Imprime el valor de la expresión.
`break <función>` `break <fichero>:linea`	b	Establece un punto de ruptura.
`info break`	b	Lista los puntos de ruptura.
`delete <B>`	d	Elimina el punto de ruptura número `B`.
`continue`	c	Continúa la ejecución después de una interrupción.
`step`	s	Ejecuta la siguiente instrucción (entrando a funciones llamadas).
`next`	n	Ejecuta la siguiente instrucción (sin entrar a funciones llamadas).
`list [<fichero>:linea]` `list [<función>]`	l	Lista el código en el lugar especificado.
`backtrace`	bt	Muestra la pila de llamadas en un momento dado (también post-mortem).
`frame <F>`	f	Selecciona la función en la posición F de la pila de llamadas.
`file <fichero>`	-	Carga un fichero ejecutable para leer los símbolos que se utilizarán al depurar un fichero `core`.
`quit`	q	Sale de `gdb`.

B1.3. El *framework* `valgrind`

`valgrind` es un conjunto de herramientas para depurar y analizar el rendimiento de otros programas. En particular, la herramienta `memcheck` de `valgrind` ayuda a identificar problemas relacionados con la gestión de memoria como *fugas* de memoria, lecturas/escrituras más allá de los límites de un array o uso de variables no inicializadas. Para ilustrar su uso, vamos a usar de nuevo el ejemplo anterior:

```
usuario@maquina:~$ valgrind ./depurame
==5562== Memcheck, a memory error detector
==5562== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==5562== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==5562== Command: ./depurame
==5562==
==5562== Invalid write of size 8
==5562==    at 0x400599: inicializa (depurame.c:12)
==5562==    by 0x4005DC: main (depurame.c:26)
==5562==  Address 0x5207ec0 is 0 bytes after a block of size 16,000 alloc'd
==5562==    at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==5562==    by 0x400577: inicializa (depurame.c:10)
==5562==    by 0x4005DC: main (depurame.c:26)
==5562==
==5562== Invalid write of size 4
==5562==    at 0x4005C2: copia (depurame.c:19)
==5562==    by 0x4005EC: main (depurame.c:27)
==5562==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==5562==
==5562== Process terminating with default action of signal 11 (SIGSEGV)
==5562==  Access not within mapped region at address 0x0
==5562==    at 0x4005C2: copia (depurame.c:19)
==5562==    by 0x4005EC: main (depurame.c:27)
==5562==  If you believe this happened as a result of a stack
==5562==  overflow in your program's main thread (unlikely but
==5562==  possible), you can try to increase the size of the
==5562==  main thread stack using the --main-stacksize= flag.
==5562==  The main thread stack size used in this run was 8388608.
==5562==
==5562== HEAP SUMMARY:
==5562==     in use at exit: 16,000 bytes in 1 blocks
==5562==   total heap usage: 1 allocs, 0 frees, 16,000 bytes allocated
==5562==
==5562== LEAK SUMMARY:
==5562==    definitely lost: 0 bytes in 0 blocks
==5562==    indirectly lost: 0 bytes in 0 blocks
==5562==      possibly lost: 0 bytes in 0 blocks
==5562==    still reachable: 16,000 bytes in 1 blocks
==5562==         suppressed: 0 bytes in 0 blocks
==5562== Rerun with --leak-check=full to see details of leaked memory
==5562==
==5562== For counts of detected and suppressed errors, rerun with: -v
==5562== ERROR SUMMARY: 18001 errors from 2 contexts (suppressed: 0 from 0)
Violación de segmento (‘core’ generado)
```

B1.3.1. Lecturas/escrituras ilegales

El primer error que detecta `valgrind` en la línea 12 del programa de ejemplo se debe a que se intenta escribir más allá de los límites de la memoria que se reserva con `malloc`.

B1.3.2. Uso de variables no inicializadas

En la línea 19 del programa, `valgrind` notifica un intento de escribir en una posición de memoria no inicializada.

B1.3.3. *Fugas* de memoria

El resumen de la pila especifica que el programa realiza una llamada a `malloc` que reserva 16000 bytes (2000 punteros a carácter de 64 bits). A su vez, el resumen de *fugas* de memoria identifica un bloque de memoria de 16000 bytes como *still reachable*, es decir, reservado pero nunca liberado.

B1.3.4. Limitaciones de `valgrind`

Aunque la herramienta `memcheck` de `valgrind` es extraordinariamente útil, tiene algunas limitaciones que se deben tener en cuenta. En primer lugar, `valgrind` no detecta las lecturas/escrituras ilegales en variables globales o que se encuentren en la pila. En segundo lugar, `valgrind` analizar el comportamiento de los programas en tiempo de ejecución, es decir, si la ejecución de un programa no pasa por el fragmento de código que causa el problema, `valgrind` no lo detectará. Por último, `valgrind` puede ralentizar considerablemente la ejecución del programa.

B1.3.5. Análisis de `simplesh` con `valgrind`

Cuando se utiliza la herramienta `memcheck` de `valgrind` para intentar encontrar problemas relacionados con la gestión de memoria de un programa, `valgrind` también analiza todas las librerías que utiliza. Por defecto, `valgrind` omite los errores encontrados en la librería `glibc`. En nuestro caso, nos interesa además, poder ignorar cualquier problema detectado en la librería `readline`. Para hacerlo, tendremos que crear un fichero llamado `libreadline.supp` con el siguiente contenido:

```
{
    libreadline
    Memcheck:Leak
    ...
    fun:*
    obj:/lib/x86_64-linux-gnu/libreadline.so.*
    ...
}
```

Y, a continuación, invocar `valgrind` de la siguiente forma:

```
$ valgrind --leak-check=full --show-leak-kinds=all \
          --suppressions=libreadline.supp -v ./simplesh 2> valgrind.out
```

Para más detalles acerca de `memcheck`, véase *Memcheck: a memory error detector*.

B1.4. La utilidad `make`

La utilidad `make` es un programa de propósito general que construye un objetivo o meta a partir de prerrequisitos o dependencias. El objetivo puede ser un programa ejecutable, un documento PostScript o cualquier otra cosa. Los prerrequisitos pueden ser un fichero de código C, un fichero de texto LaTeX, etc.

Supongamos que queremos compilar los ficheros `main.c` y `edit.c` para generar el ejecutable `prog`:

```
gcc -o prog main.c edit.c
```

Si modificamos `edit.c`, ¿por qué tenemos que compilar también `main.c` de nuevo? La solución es guardar lo siguiente en un fichero que se debe llamar `makefile` o `Makefile`.

```
# Los comentarios se escriben de esta manera.
prog: main.o edit.o
<Tab> gcc -o prog main.o edit.o
main.o: main.c
<Tab> gcc -c main.c
edit.o: edit.c
<Tab> gcc -c edit.c
```

En la primera línea `prog` es un objetivo, y `main.o` y `edit.o` son dependencias. El significado es: "para obtener el fichero `prog` necesitas los ficheros `main.o` y `edit.o`". Una vez que los tengas, `prog` se construye con:

```
gcc -o prog main.o edit.o
```

Es muy importante observar el uso del tabulador. De lo anterior se deduce que en lugar de `gcc` se podría colocar cualquier otra orden. Tras crear el fichero anterior debemos ejecutar:

```
make
```

o bien

```
make prog
```

En el segundo caso se indica qué objetivo se debe construir. En el primero, se construye el objetivo por defecto, que es el primer objetivo que se indique en el fichero `Makefile`. El orden de los objetivos es indiferente, con la excepción del primero que se toma por defecto si sólo se ejecuta `make`.

B1.4.1. Macros

Un ejemplo de macro es `OBJECTS=main.o edit.o`, que se podría utilizar en el `Makefile` del ejemplo anterior:

```
# Los comentarios se escriben de esta manera.
prog: main.o edit.o
<Tab> gcc -o prog $(OBJECTS)
main.o: main.c
<Tab> gcc -c main.c
edit.o: edit.c
<Tab> gcc -c edit.c
```

Otro ejemplo:

```
DRIVERS = drivers/block/scsi/scsi.a
ifdef CONFIG-SCSI
    DRIVERS := $(DRIVERS) drivers/scsi/scsi.a
endif
```

`CONFIG-SCSI` se define con `CONFIG-SCSI=yes` dentro del fichero `Makefile`, con `make CONFIG-SCSI=yes objetivo`, o con un `export CONFIG-SCSI=yes` desde el *shell*. Observar el uso de `:=` en lugar de `=`. Esto se hace para poder utilizar en la asignación la propia macro.

B1.4.2. Reglas de sufijos y reglas de patrones

Imaginemos que en un fichero `Makefile` aparecen estas líneas:

```
.c.o:
    gcc -c $(CFLAGS) $<
```

La primer línea es una regla de sufijos, que significa que para obtener un fichero con sufijo `.o` es necesario tener un fichero del mismo nombre pero con sufijo `.c`. `$<` es una forma críptica de referirse al fichero de entrada. `$(CFLAGS)` es otro ejemplo de uso de macros, en este caso, para definir opciones de compilación.
Si ejecutásemos:

```
make CFLAGS="-O -g" edit.o
```

obtendríamos

```
gcc -c -O -g edit.c
```

Ahora supongamos que un fichero `Makefile` contiene las siguientes líneas:

```
%.o: %.c
    gcc -c -o $@ $(CFLAGS) $<
```

La primera línea es una regla de patrones, que tiene el mismo significado que la regla de sufijos anterior. El carácter `%` representa a cualquiers cadena. La macro predefinida `$@` hace referencia al fichero de salida y `$<` tiene el mismo significado que antes.

B1.4.3. Reglas implícitas

En algunos casos es posible omitir la regla que especifica como construir un objetivo. `make` intentará construirlo utilizando alguna de las reglas predefinidas que incorpora. En el caso anterior, si omitiésemos los prerrequisitos y las reglas para los objetivos `main.o` y `edit.o`, `make` utilizará una regla predefinida para compilar ambos ficheros:

```
$(CC) $(CPPFLAGS) $(CFLAGS) -c main.c
$(CC) $(CPPFLAGS) $(CFLAGS) -c edit.c
```

Se puede obtener una lista completa de las reglas predefinidas o implícitas de `make` con:

```
make -p
```

Por ejemplo:

```
%.o: %.c
        $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c $<

%: %.o
        $(CC) $(LDFLAGS) $(TARGET_ARCH) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

B1.4.4. Ejecución de comandos

Cualquier comando del *shell* se puede ejecutar en un fichero `Makefile`. No obstante, `make` ejecuta cada comando en un *shell* distinto. Por tanto, la siguiente secuencia de comandos no hace lo esperado:

```
cd obj
OBJ_DIR=/home/user/obj
mv *.o $OBJ_DIR
```

Sin embargo, si funcionaría con

```
cd obj; OBJ_DIR=/home/user/obj; mv *.o $OBJ_DIR
```

o con

```
cd obj;\
OBJ_DIR=/home/user/obj;\
mv *.o $OBJ_DIR
```

B1.4.5. Reglas falsas

Resulta muy habitual definir una regla sin prerrequisitos para eliminar tanto los objetivos como otros ficheros que pudiese haber generado `make`:

```
clean:
    rm -fr prog $(OBJECTS)
```

Sin embargo, ¿qué pasaría si existiese un fichero llamado `clean`? Si se diese ese caso, `make clean` no borraría nunca ni `prog` ni los ficheros `*.o`. Para evitarlo se utiliza lo que se conoce como una regla falsa o `PHONY`:

```
.PHONY: clean
clean:
    rm -fr prog $(OBJECTS)
```

Ahora, `make clean` borraría todos los ficheros especificados aunque existiese un fichero `clean`.

B1.4.6. Otras consideraciones

- `make` puede ser recursivo. La macro predefinida `$(MAKE)` hace que se invoque a `make`.
- `make` permite que la salida de una orden se guarde en una macro: `HOST=$(shell uname -n)`.
- `@`: este símbolo al principio de una línea hace que no se produzca eco de la misma cuando se ejecuta el `Makefile`.
- `-`: un guión al principio de una orden permite que continúe la ejecución del fichero `Makefile` aunque falle la orden (útil para órdenes como `cp` o `mv`). Por defecto, `make` termina inmediatamente si una orden falla.

2 Ejercicios

1. Modifica el programa `depurame` para subsanar los dos errores encontrados con `gdb`.
2. Elimina la *fuga* de memoria detectada en el programa `depurame` con `valgrind`.
3. Comprueba si `simplesh` tiene alguna *fuga* de memoria con `valgrind` y, si es así, elimínala.
4. Extiende la funcionalidad de depuración de `simplesh` (véase `DGB_TRACE`) a todas las funciones `parse_*` y, a continuación, determina la secuencia de llamadas a funciones `parse_*` con la línea de órdenes:

   ```
   $ ls > listado
   ```

5. Con la ayuda de `gdb`, verifica que el resultado del ejercicio anterior es correcto.

   Nota: Cuando `gdb` encuentra un `fork()` sigue ejecutando el código del proceso padre. Para hacer que `fork()` siga ejecutando el código del proceso hijo:

   ```
   (gdb) set follow-fork-mode child
   ```

3 Referencias

- `gdb`: `https://www.gnu.org/software/gdb/documentation/`
- `valgrind`: `http://valgrind.org/docs/manual/index.html/`
  - Cprogramming.com: *Using Valgrind to Find Memory Leaks and Invalid Memory Use*
- `make`: `https://www.gnu.org/software/make/manual/`