Proyecto 2: `rautomake`: El generador de `Makefile`s

CI3825 (Sistemas de operación I)

Enero–Marzo 2012

Índice general

Este documento es solo una propuesta y no especifica requerimientos para ninguna evaluación. No debe suponerse que alguna evaluación vaya a ser basada en él a menos que sea publicado oficialmente por los profesores a cargo del curso; en ese caso, la versión normativa del documento será la que se publique, que no contendrá esta nota, y esta versión será considerada inválida y no deberá ser usada como referencia para ningún propósito.

Este documento es un borrador, y se espera que tenga deficiencias de diseño, del redacción, orthografícas y de t**ipo**g$^{\text{r}}$a~~f~~$_{\text{ía}}$.

1 Introducción

Esta sección es informativa.

1.1 Objetivos

1. Familiarizarse con las llamadas al sistema para la creación y acceso a las estructuras del sistema de archivos.
2. Desarrollar un mecanismo de comunicación entre un proceso y múltiples hijos concurrentes utilizando los estados de salida de los hijos.
3. Utilizar la familia de llamadas al sistema `exec` para que un proceso ejecute a otro programa como parte de su ejecución.
4. Fortalecer el conocimiento de herramientas de automatización del proceso de desarrollo de software como `make`.

1.2 *Build system*s

1. El proceso de desarrollo de software con cualquier conjunto de herramientas implica realizar ciertas tareas repetitivas para transformar los datos crudos y el código fuente desarrollados por los programadores en un producto terminado que se pueda distribuir y utilizar. Si se utilizan lenguajes de programación compilados, este proceso incluye la ejecución de compiladores para transformar código fuente en ejecutables. Esta tarea puede resultar tediosa en el contexto de un proyecto de complejidad considerable, y como el tiempo de un desarrollador de software es mejor aprovechado cuando se utiliza para desarrollar software, sería deseable eliminar esa carga de trabajo con procesos automatizados.
2. Un ***build system*** es un sistema de automatización para las tareas de producción de paquetes de software a partir de código y datos fuente. Esta definición general incluye sistemas de alcances diversos:
    1. Los `Makefile`s simples para compilar alguno de sus proyectos pueden ser considerados *build system*s: son programas que describen el proceso de construcción de los ejecutables de sus proyectos.
    2. El programa que interpreta sus `Makefile`s y ejecuta acciones descritas en ellos, `make`, también puede considerarse un *build system*.
    3. Existen *build system*s que trabajan a un nivel más alto, como GNU Automake y GNU Autoconf que son dos de los componentes del GNU build system, también conocido como *Autotools*. Este sistema puede generar `Makefile`s automáticamente ajustados a las peculiaridades de cada tipo de sistema UNIX y cada proyecto, lo que lo hace adecuado para proyectos de gran escala que deban funcionar en múltiples plataformas.
    4. Fuera del mundo de C es común encontrar *build system*s asociados a lenguajes y plataformas específicas, como Apache Ant para Java, Scons para Python, Rake para Ruby, Cabal para Haskell, etc. Casi todas estas herramientas pueden ser usadas independientemente del lenguaje o la plataforma para la cual esté escrito el proyecto al que son aplicadas, y ofrecen diversos grados de expresividad y automatización: algunas hasta proveen lenguajes de programación completos para describir el proceso que automatizan.

1.3 Compilación separada por directorios

1. En un proyecto de software de escala no trivial resulta conveniente dividir la fuente del proyecto en categorías y agrupar los archivos de cada una en subdirectorios del directorio del proyecto. Por ejemplo, un sistema de documentación de código escrito en C con soporte para múltiples lenguajes podría tener una estructura de directorios en su código fuente como esta:

```
multidoc/             # Directorio principal del proyecto
multidoc/src          # Código fuente del proyecto
multidoc/src/c        # Uso de documentación en código C
multidoc/src/cc       # Uso de documentación en código C++
multidoc/src/asm      # Uso de documentación en código assembly
multidoc/src/asm/x86  # Lenguaje de máquina x86
multidoc/src/asm/mips # Lenguaje de máquina MIPS
multidoc/src/java     # Uso de documentación en código Java
multidoc/lib          # Código de una biblioteca incluida
multidoc/doc          # Documentación del proyecto
multidoc/test         # Pruebas de correctitud del proyecto
```

2. El directorio `src` y cada uno de sus subdirectorios podrían contener varios archivos de código fuente que se compilen de la misma manera a archivos de objeto y se combinen finalmente en un ejecutable de todo el proyecto. En ese caso, sería conveniente tener en cada subdirectorio un `Makefile` que describa cómo compilar los archivos de código fuente de ese subdirectorio, y si contiene a otro subdirectorio, que llame a los `Makefiles` contenidos en ellos. Por ejemplo, el `Makefile` del directorio `multidoc/src` llamará al `Makefile` del directorio `multidoc/src/asm`, que a su vez llamará al `Makefile` del directorio `x86`. Claro que cada `Makefile` no debe llamar a solo *uno* de los `Makefiles` de sus subdirectorios, sino a *todos*.

3. A partir de los archivos de código fuente (los `.c`) de cada subdirectorio serán generados sus correspondientes archivos de objeto (los `.o`). El objetivo del proceso de compilación es reunir todo el código generado (todos los `.o`) para crear finalmente un archivo ejecutable en la raíz de la jerarquía de directorios del proyecto (que es el producto final del proyecto ficticio `multidoc`). Para facilitar este proceso, en cada subdirectorio debería crearse un archivo `.a` que contenga la *colección de todos los* `.o` de ese subdirectorio (y de los subdirectorios que contenga). Luego, el *directorio padre* de ese subdirectorio podría hacer lo mismo con sus archivos de código fuente, y en su colección incluir a sus archivos de objeto y a las *colecciones* (los `.a`) de todos sus directorios hijos.

4. Por ejemplo, suponga que en `multidoc/src/asm/x86` existen dos archivos de código fuente a compilar: `parse.c` y `opcodes.c`. Suponga además que `opcodes.c` incluye a un archivo de encabezado `opcodes.h`, y que `parse.c` incluye a `parse.h`, y `parse.h` a su vez también incluye a `opcodes.h`. El directorio `multidoc/src/asm/x86` debería contener un `Makefile` que diga, por ejemplo,

```
.PHONY: all
all: x86.a

x86.a: parse.o opcodes.o
	ar rT $@ $?

parse.o: parse.c parse.h opcodes.h
opcodes.o: opcodes.c opcodes.h
```

5. El comando

   ```
   ar rT archivo.a elemento1 elemento2 ...
   ```

   crea un archivo de colección llamado `archivo.a` que contendrá referencias al código compilado de `elemento1`, `elemento2`, etc. Si alguno de esos elementos es a su vez un archivo de colección, se almacenarán referencias a sus elementos por separado en vez de al archivo de colección completo como uno de los miembros. La secuencia `$@` en la *receta* de una regla de un `Makefile` se sustituye por el nombre del archivo que causó que se corriera esa regla; en el caso de la regla cuya receta ejecuta al comando `ar`, `$@` se sustituye por `x86.a`. `$?` se sustituye por aquellas dependencias de la regla que deban actualizarse: por ejemplo, si acabamos de compilar de nuevo a `opcodes.c` y obtuvimos una nueva versión de `opcodes.o`, se actualizará la referencia a `opcodes.o` en el archivo de colección `x86.a`, pero si no se actualizó `parse.o`, no se sustituirá su referencia en la colección (porque no es necesario hacerlo).

6. Suponga ahora que se hizo lo mismo en el directorio `multidoc/src/asm/mips`. En el directorio `multidoc/src/asm` sucede lo mismo, excepto que al archivo de colección habrá que insertarle las referencias a los elementos de las colecciones de sus subdirectorios:

   ```
   .PHONY: all force
   all: asm.a

   asm.a: x86/x86.a mips/mips.a asm.o marmota.o ...
           ar rT $@ $?

   x86/x86.a: force
           $(MAKE) -C x86

   mips/mips.a: force
           $(MAKE) -C mips

   asm.o: asm.c asm.h marmota.h archs.h
   marmota.o: marmota.c marmota.h archs.h
   ...
   ```

7. El nuevo archivo de colección `x86/x86.a` contendrá referencias a todo el código compilado correspondiente a ese directorio y a todos sus subdirectorios (recursivamente).

8. Note que la regla *phony* "force" se utiliza como prerequisito para los archivos de colección de los subdirectorios; el efecto de esto es que **siempre** se harán las llamadas recursivas a `make` para los subdirectorios. Esto es necesario porque el `Makefile` de un directorio no tiene información sobre si es necesario actualizar el código compilado en un subdirectorio (si su `.c` o alguna de sus dependencias cambió). Sin embargo, el `Makefile` que está dentro de ese subdirectorio sí será capaz de determinar esto. Por lo tanto, la llamada recursiva se hace incondicionalmente, y si había algo que actualizar dentro de un subdirectorio, la invocación recursiva de `make` hará el trabajo y actualizará el archivo de colección de ese subdirectorio.

9. Los directorios `multidoc/src/c`, `multidoc/src/cc` y `multidoc/src/java` tendrían `Makefile`s muy similares a los de los directorios `multidoc/src/asm/x86` y `multidoc/src/asm/mips`: como no tienen subdirectorios, solo especifican reglas con las dependencias de sus archivos de código fuente, y la regla que dice cómo se construye el archivo de colección de ese directorio. Como el directorio `multidoc/src` sí tiene subdirectorios con código fuente, tendría un `Makefile` similar al de `multidoc/src/asm`:

```
.PHONY: all force
all: src.a

src.a: c/c.a cc/cc.a asm/asm.a java/java.a multidoc.o xml.o ...
        ar rT $@ $?

c/c.a: force
        $(MAKE) -C c

cc/cc.a: force
        $(MAKE) -C cc

asm/asm.a: force
        $(MAKE) -C asm

java/java.a: force
        $(MAKE) -C java

multidoc.o: multidoc.c multidoc.h xml.h asm/archs.h
xml.o: xml.c xml.h asm/archs.h
...
```

10. Finalmente, en el directorio principal del proyecto se compilarán todos los archivos compilables que estén directamente en él, y se enlazarán todos sus archivos de objeto y todos los archivos de colección de los subdirectorios del proyecto. El resultado de esto debe ser un ejecutable. En el directorio principal del proyecto podría haber, por ejemplo, este `Makefile`:

```
.PHONY: all force
all: multidoc

multidoc: src/src.a lib/lib.a main.o args.o ...
        $(CC) $(CPPFLAGS) $(CCFLAGS) -o $@ $^

src/src.a: force
        $(MAKE) -C src

lib/lib.a: force
        $(MAKE) -C lib

main.o: main.c main.h args.h
args.o: args.c args.h
...
```

11. La secuencia `$^` en una receta de una regla se sustituye con los nombres de todos los prerequisitos de la regla separados por espacio (sin repeticiones). La receta mostrada en el ejemplo de `Makefile` anterior genera el ejecutable final del proyecto que contiene todo el código compilado de todos los archivos fuente del proyecto.

12. Note que no se incluyó `doc/doc.a` ni `test/test.a` porque en el caso particular de este ejemplo esos serían directorios que no se usan para almacenar código fuente (ni directamente, ni en algún subdirectorio descendiente de ellos directa o indirectamente). Esto implica que no interesaría escribir `Makefile`s en estas ramas de la jerarquía de directorios, y claro, si no hay `Makefiles`, no habrá tampoco llamada recursiva de `make` en ellos.

1.4 Generación de dependencias

1. Las líneas de un `Makefile` que indican las dependencias de un archivo de código fuente compilable (un `.c`) son tediosas de generar y de mantener actualizadas. Si se agrega `#include "foo.h"` al archivo `bar.h`, debería agregarse `foo.h` a todos los archivos que incluyan a `bar.h` como dependencia directa o indirecta; además, si `foo.h` incluye a otros archivos, habrá que agregarlos a todos, y a todos los que éstos incluyan, y así sucesivamente. Este proceso recursivo de resolución de dependencias implica procesar los archivos `.c` y `.h` para encontrar sus dependencias; por ejemplo, si una inclusión está condicionada con `#if` y/o `#ifdef` que dependa de valores que solo pueda conocer el compilador, entonces podríamos incluir más dependencias de las necesarias si nos limitamos a incluir dependencias de la manera ingenua: cada vez que veamos una línea que comience con `#include "`.

2. Afortunadamente, los compiladores de C casi universalmente soportan opciones especiales que hacen que en vez de compilar un archivo de código fuente compilable, generen el código correspondiente a la regla que debería contener un `Makefile` para compilarlo y mantener su archivo de objeto actualizado correctamente. Por ejemplo, en el caso del archivo de ejemplo `parse.c` anteriormente mencionado, podría ejecutarse en su directorio el comando

   ```
   gcc -E -MMD parse.c
   ```

   y se generará otro archivo llamado `parse.d` cuyo contenido será

   ```
   parse.o: parse.c parse.h opcodes.h
   ```

   que es precisamente lo que debe contener el `Makefile` que esté en el directorio de `parse.c`. Hay varias variaciones de estas opciones; todas comienzan con `-M`, pero algunas generan un archivo con un nombre fijo, otras generan la salida en un archivo cuyo nombre es parte de la opción, otras generan el texto de la regla en la salida estándar del compilador, etc. El manual de `gcc` describe en detalle todas las opciones que provee para cálculo de dependencias para `Makefile`s.

3. Como este análisis es efectuado por el compilador de C (en particular por su preprocesador), las dependencias generadas serán precisamente las necesarias, tomando en cuenta la posibilidad de inclusiones condicionadas a variables del preprocesador. Toda la complejidad de este análisis se puede delegar al compilador, y el `Makefile` puede simplemente contener el texto de las dependencias generado por el compilador.

1.5 Estados de salida

1. Cuando un proceso termina su ejecución, su entrada en la tabla de procesos del sistema operativo se reduce significativamente, pero no se elimina por completo. Los recursos que el proceso tuviera reservados son liberados, incluyendo su memoria con su código ejecutable y sus datos. Se mantiene, sin embargo, un mínimo de información referente a la historia de la ejecución del proceso, incluyendo varios datos estadísticos (cuánto tiempo estuvo en ejecución, cuál fue su carga promedio, etc), su identificador de proceso, el identificador de proceso de su padre, etc. Entre esos datos está el *estado de salida* del proceso.

2. El *estado de salida* de un proceso que ha terminado es un número entero de 8 bits que un proceso produce cuando retorna de su función `main` o cuando ejecuta la función `exit`. Del entero retornado, o del entero que es pasado como parámetro a `exit`, se toman los ocho bits menos significativos para obtener el estado de salida del proceso; todo el resto del número se ignora completamente. Cuando un proceso termina, su proceso padre puede obtener su *estado de salida* usando las funciones `wait`, `waitpid` o `waitid`.

3. Este mecanismo permite que un proceso delegue parte de su trabajo a un proceso hijo y obtenga un resultado simple luego de su terminación que indique alguna información sobre ese trabajo. Aunque los datos que se pueden comunicar mediante este mecanismo son muy limitados (porque solo se puede pasar un entero de 8 bits), es muy conveniente: no es necesario establecer `pipe`s ni manejar señales para "retornar" un valor simple al proceso padre. Los programas del sistema típicamente usan sus estados de salida para indicar que su ejecución fue exitosa produciendo un estado de salida igual a cero, y si hubo un error, producen estados de salida distintos de cero cuyos valores particulares pueden indicar el tipo de error encontrado.

1.6 Ejecución de programas con `exec`

1. La familia de llamadas al sistema `exec` permite que un proceso sustituya su propio contenido por el de algún programa disponible en el sistema y comience a ejecutarlo. `exec` es el mecanismo básico que usan los programas de un sistema UNIX para producir a otros programas: primero se duplican llamando a `fork`, y el proceso hijo procede inmediatamente a ejecutar una de las funciones de la familia `exec` para ejecutar a otro programa.
2. Este código demuestra un uso mínimo de `fork`, `wait` y `execlp` para ejecutar un programa instalado en el sistema (en particular, un compilador de C que calculará las dependencias del código fuente del mismo programa) y capturar su estado de salida.

```c
#include <stdio.h>     /* printf, puts, perror */
#include <stdlib.h>    /* exit */
#include <sys/wait.h>  /* wait */
#include <unistd.h>    /* fork, execlp */

#define do { PERROR_EXIT(s) perror(s); exit(EXIT_FAILURE); } while (0)

int main() {
        int status;

        switch(fork()) {
                case -1:
                        PERROR_EXIT("fork");

                case  0:
                        execlp("cc", "cc", "-E", "-MM", __FILE__, NULL);
                        PERROR_EXIT("execlp");

                default:
                        break;
        }

        if (wait(&status) == -1) PERROR_EXIT("wait");

        if (!WIFEXITED(status)) puts("El hijo terminó de forma irregular.");
        else {
                printf(
                        "El hijo terminó con el estado de salida %d\n",
                        WEXITSTATUS(status)
                );
        }
        exit(EXIT_SUCCESS);
}
```

2 Enunciado

Esta sección es normativa.

2.1 Requerimientos

1. Usted debe desarrollar un programa llamado `rautomake` escrito en el lenguaje de programación C.
2. `rautomake` será ejecutado dentro de un cierto directorio. Por ejemplo, suponga que el ejecutable de `rautomake` (que será el resultado de compilar su proyecto) reside en

   `/home/marmota/USB/CI3825/2012EM/P2/src/rautomake`

   y usted ejecuta el comando `../src/rautomake` en un intérprete de línea de comando cuyo directorio actual es

   `/home/marmota/USB/CI3825/2012EM/P2/multidoc`

   entonces *ese último directorio* será el directorio donde se ejecuta `rautomake`.
3. Es necesario que `rautomake` cree un `Makefile` en un subdirectorio del directorio donde se ejecutó, o en ese mismo directorio, si `rautomake` tuviera permiso de lectura, escritura y acceso a ese directorio y a todos sus padres hasta el directorio donde fue ejecutado `rautomake`, inclusive, y si hubiera sido necesario que `rautomake` creara un `Makefile` en algún subdirectorio del directorio en cuestión o en él existen archivos para los que `rautomake` tenga permiso de lectura y cuyos nombres terminen en `.c`.
4. Cada `Makefile` que `rautomake` deba crear deberá contener una regla para cada archivo en el directorio de ese `Makefile` cuyo nombre termine en `.c`. El contenido de la regla será el mismo que algún compilador de C generaría al pedirle que genere la regla correspondiente a ese archivo para un `Makefile`; debe tener como prerequisitos al menos a todos los archivos que al compilarlo serían incluidos directa o indirectamente por el archivo en cuestión que estén dentro del directorio en el que fue ejecutado `rautomake`, y no debe tener como prerequisito a ningún archivo que no pueda ser incluido ni directa ni indirectamente por el archivo en cuestión.
5. Cada `Makefile` que `rautomake` deba crear en un directorio distinto a donde fue ejecutado deberá contener las reglas necesarias para crear un archivo de colección con el programa `ar` que contenga referencias a todo el código de objeto generado al compilar cada uno de los archivos de ese directorio cuyo nombre termine en ".c", junto con referencias a todo el código de objeto referido por archivos de colección de sus subdirectorios en los que `rautomake` deba crear `Makefile`s.
6. Si `rautomake` debió crear algún `Makefile` en cualquier directorio, entonces `rautomake` deberá crear un `Makefile` en el directorio donde fue ejecutado. Ese `Makefile` deberá contener una regla que cree un archivo ejecutable cuyo nombre sea igual al nombre del directorio donde `rautomake` fue ejecutado; ese archivo ejecutable será el resultado de enlazar todos los archivos de objeto correspondientes a archivos en el directorio en cuestión cuyos nombres terminen en ".c", junto con el código referido por los archivos de colección correspondientes a los subdirectorios del directorio en cuestión en los que `rautomake` debiera generar un `Makefile`. Los `Makefile`s que `rautomake` debe crear deben permitir que se genere el ejecutable mencionado en este párrafo si se ejecuta la orden `make` en el mismo directorio donde `rautomake` fue llamado, suponiendo que todos los archivos de código fuente mencionados puedan compilarse a archivos de objeto, y que ese ejecutable se pueda generar enlazando a todos los archivos de objeto resultantes de esas compilaciones.
7. Un proceso de `rautomake` no puede visitar ni abrir más de un directorio a lo largo de su ejecución, a menos que ese proceso sea un compilador de C. Si un proceso de `rautomake` que no sea un compilador de C requiere visitar o abrir más de un directorio, deberá copiarse a sí mismo y una de sus copias podrá visitar o abrir un directorio distinto al visitado o abierto por el proceso original. Verificar los permisos de un directorio no constituye haberlo visitado ni abierto.

8. La comunicación entre varios procesos resultantes de una ejecución de `rautomake` solo podrá hacerse a través de estados de salida y señales del sistema operativo, a menos que uno de los procesos involucrados sea un compilador de C, en cuyo caso es aceptable la comunicación a través de archivos.

9. A menos que este documento haga una excepción explícita particular, `rautomake` **debe** manejar explícitamente absolutamente todos los efectos y resultados de llamadas al sistema o a bibliotecas que puedan afectar al flujo de ejecución de su implementación de `rautomake`. En particular, si su implementación de `rautomake` deja de revisar una posible condición de error en el retorno de una llamada al sistema o a alguna biblioteca que utilice, incluyendo la biblioteca estándar de C, se esperará que usted pueda demostrar que el flujo de ejecución de su programa nunca depende de esa condición.

2.2 Condiciones de ejecución

1. Esta subsección define propiedades que `rautomake` puede suponer que cumple el ambiente donde será ejecutado, y no es necesario que sean verificadas explícitamente.

2. `rautomake` podrá suponer que no existe ningún archivo o subdirectorio dentro del directorio donde es ejecutado, ni dentro de ninguno de sus subdirectorios, cuyo nombre no sea una secuencia de caracteres alfanuméricos de ASCII, o el caracter "FULL STOP", también llamado "punto" (`U+002E`, "`.`"); podrá suponer lo mismo, además, del nombre del directorio donde es ejecutado.

3. `rautomake` podrá suponer que no existe ningún archivo dentro del directorio donde es ejecutado, ni dentro de ninguno de sus subdirectorios, cuyo nombre termine en ".d". Si tales archivos existen, `rautomake` podrá eliminarlos o sobreescribirlos para cualquier fin. `rautomake` no será ejecutado en un directorio donde existan archivos con nombres de esa forma sobre los que no tenga permiso de lectura y escritura. `rautomake` no será ejecutado en un directorio que tenga algún subdirectorio directo o indirecto cuyo nombre termine en ".d".

4. `rautomake` podrá suponer que no existe ninguna entrada del directorio donde es ejecutado, ni de ninguno de sus subdirectorios, que no sea un archivo regular o un directorio.

2.3 Entrega

1. El código principal de `rautomake` debe estar escrito en el lenguaje de programación C en cualquiera de sus versiones, debe poder compilarse con las herramientas GNU disponibles en la versión estable más reciente al momento de la entrega de Debian GNU/Linux; en particular, debe poder compilarse y ejecutarse en las computadoras del LDC.

2. La entrega consistirá de un archivo en formato `tar.gz` o `tar.bz2` que contenga todo el código que haya desarrollado y sea necesario para compilar su programa. No debe incluir archivos compilados. Su proyecto debe poder compilarse extrayendo los archivos del paquete de su entrega, entrando en el directorio generado por la extracción, y ejecutando el comando `make`; es decir que *debe* incluir al menos un `Makefile` para automatizar la compilación de su proyecto.

3. Aunque en general es preferible que su código sea compatible con los documentos de estandarización más recientes publicados para cada tecnología que utilice, es aceptable que requiera y use extensiones propias de la implementación de las herramientas de la plataforma GNU/Linux tanto del lenguaje de programación C como de la interfaz con el sistema operativo. Es aceptable que requiera alguna versión de las herramientas del sistema diferente de las que están instaladas y disponibles en la versión estable más reciente al momento de la entrega de Debian GNU/Linux, y en particular de lo que esté instalado y disponible en las computadoras del LDC, pero deberá justificar estos requerimientos adicionales.

4. Deberá enviar su entrega a la plataforma asociada al curso, en la sección que será creada para este fin, antes de las 13:00 HLV[1] del lunes 2012–03–26 (semana 12 del trimestre en curso).

[1] Hora legal de Venezuela (UTC−04:30).

3 Información adicional

Esta sección es informativa.

1. Este proyecto está diseñado para instruir en el uso del lenguaje C, herramientas de desarrollo de software como `make` y `gcc`, y la interfaz de programación de los sistemas operativos que implementan los estándares POSIX[2], y particularmente todo lo relacionado con el acceso a archivos y directorios y la comunicación entre procesos. El diseño del proyecto, y por lo tanto de la herramienta que usted debe implementar, está orientado a ese propósito educativo y no necesariamente obedece a los criterios de diseño adecuados para la creación de un *build system*.

2. En particular, aunque el uso recursivo de `make` que este proyecto propone es sin duda muy similar a las prácticas comunes de muchos proyectos de software de gran envergadura, esta técnica sufre de problemas importantes y hay buenas razones para *no* usarla. Varios de estos problemas se exploran en Miller, 1997[3] (disponible en PDF); ese artículo es un buen comienzo si es de su interés conocer la manera *correcta* de usar `make` con proyectos divididos en múltiples directorios.

[2] Documentos de POSIX.1-2008 (también llamado "IEEE Std 1003.1™-2008", o "The Open Group Technical Standard Base Specifications, Issue 7")

[3] Peter Miller — Recursive make considered harmful (1997)