Tema 5 - Compilación de grandes proyectos (DCA)

HISTORIAL DE REVISIONES			
NÚMERO	FECHA	MODIFICACIONES	NOMBRE

Índice

Logo DLSI

Tema 5 - Compilación de grandes proyectos
Curso 2018-2019

1. Preliminares

- Incluso el proyecto de software más sencillo necesita de varios ficheros que dependen entre ellos de distinta manera.
- Estos ficheros deben compilarse, enlazarse, etc... y si se modifica alguno, se ha de repetir el proceso. En proyectos pequeños esto no importa mucho...pero en proyectos grandes sí que lo hace. En este thread de StackOverflow tienes algunos otros ejemplos de tiempos de compilación de otras versiones de Windows. Dado que el código del S.O. Windows es *cerrado* puedes comparar con la alternativa *libre* llamada ReactOS.
- La herramienta **`make`** simplifica todo este proceso. Para ello tiene en cuenta las fechas de la última modificación de cada fichero.
- Podemos concluir por tanto que **`make`** es un generador de órdenes en base a marcas de tiempo.

2. Make (I)

- Nos da información de lo que hace y porqué con la opción **`-d`**.
- Con la opción **`-k`** continúa la ejecución aunque haya errores.
- Al invocar a **`make`** podemos dar valor a variables, por ejemplo: **`"make CC=clang"`**. Cambia el compilador de **`C`** guardado en la variable **`CC`**. El compilador de **`C++`** suele guardarse en la variable **`CXX`**.

3. Make (II). Fichero Makefile

- Es un fichero en formato ASCII.
- Se permiten comentarios, comienzan por un símbolo `#`.
- El `Makefile` más sencillo consta sólo de reglas:

```
objetivo ... : dependencias
(TAB)    orden
(TAB)    ...
(TAB)    ...
```

4. Make (III). Fichero Makefile

`objetivo o target`
Nombre de un fichero a generar o el de una acción a ejecutar (`clean`) — objetivo especial **`.PHONY`** — .

`dependencias`
Es una lista de ficheros de los que depende el `objetivo`.

orden
Son las acciones que `make` realiza para obtener el `objetivo`.

El `objetivo` se puede separar de las `dependencias` por un **"::"**. Si hay varias reglas con el mismo `objetivo`, se comportan como si fueran diferentes.

Para un mismo **objetivo** sólo podemos emplear un tipo de separador, o **":"** o **"::"**.

5. Make (IV). Ejemplo Makefile

```
edit : main.o kbd.o command.o display.o insert.o search.o files.o utils.o
       cc -o edit main.o kbd.o command.o display.o insert.o search.o \
             files.o utils.o

main.o : main.c defs.h
         cc -c main.c
kbd.o : kbd.c defs.h command.h
         cc -c kbd.c
command.o : command.c defs.h command.h
         cc -c command.c
.........
clean :
         rm edit main.o kbd.o command.o display.o \
            insert.o search.o files.o utils.o
```

6. Make (V). Ejemplo Makefile con variables

```
objects = main.o kbd.o command.o display.o insert.o search.o files.o utils.o

edit : $(objects)
         cc -o edit $(objects)

main.o : main.c defs.h
         cc -c main.c
kbd.o : kbd.c defs.h command.h
         cc -c kbd.c
command.o : command.c defs.h command.h
         cc -c command.c
.........
    .PHONY : clean
clean :
         rm edit $(objects)
```

VARIABLES ESPECIALES:

$?
Representa la lista de dependencias de una regla que son *más jóvenes* que el objetivo actual.

$^
Representa la lista completa de dependencias.

$@
Representa el objetivo actual.

7. Make (VI).

OBJETIVOS ESPECIALES

- Cada proyecto tendrá un Makefile general asociado, y éste siempre tendrá (al menos) los siguientes objetivos:
  - **all**: Será el primero y representa el objetivo por defecto.
  - **clean**: Borra todos los ficheros intermedios que se puedan volver a generar al ejecutar `make`.
  - **dist**: Crea un fichero `.tar.gz` que contiene la *distribución* del proyecto.
  - **install**: Instala el resultado de crear el proyecto.
  - **uninstall**: Desinstala el resultado de crear el proyecto.

APROVECHAMIENTO DE LOS NUCLEOS DEL PROCESADOR

- Make puede lanzar compilaciones en paralelo.
- Para ello debes emplear la opción **-j** seguida del numero de trabajos en paralelo a lanzar, por ejemplo: **`make -j2`**
- Lanzar trabajos en paralelo suele descubrir fallos a la hora de especificar las dependencias, no es lo mismo satisfacerlas de forma secuencial que en paralelo.

8. Make (VII). Multinivel

- Si el proyecto que desarrollamos tiene la entidad suficiente, lo dividiremos en varios módulos, estando el código de cada uno de ellos en un subdirectorio.
- Cada subdirectorio tendrá su propio *`Makefile`* y el directorio principal del proyecto tendrá uno que gestiona los *sub-`Makefile`* de cada subdirectorio.
- **`make`** permite llamadas recursivas, de manera que el **`make`** ejecutado en el directorio principal irá invocando un **`make`** por cada subdirectorio.
- Hay desarrolladores que descartan el uso de `make` recursivo, introduce complejidades y puede ser más lento que usar un solo `Makefile` en el directorio principal del proyecto.

9. Ccache (I)

- Se trata de una herramienta basada en una **idea muy sencilla**: *una caché para el compilador*. De ahí su nombre: ccache.
- **¿Y cómo funciona?** : Se *cachean* (guardan — ~/.ccache/ —) los resultados de compilaciones previas (ficheros "`.o`") los cuales se proporcionan instantaneamente cuando se detecta que se debe volver a compilar el mismo código otra vez. Por lo tanto no se deben volver a regenerar (*tiempo ahorrado*).
- La versión actual de `ccache` soporta los lenguajes: "`C`", "`C++`", "`Objective-C`" y "`Objective-C++`".
- `ccache` está hecho de manera que produce la misma salida del compilador que obtendríamos si no lo estuviéramos usando.
- Esto es así hasta tal punto, que la única manera de saber que lo estamos usando es por los tiempos de compilación obtenidos.

10. Ccache (II). Características

- Mantiene estadísticas de aciertos/fallos
- Gestión automática del tamaño de la cache
- Puede cachear compilaciones con "`warnings`".
- Es fácil de instalar
- Añade una sobrecarga mínima al proceso de compilación
- Opcionalmente puede usar enlaces *duros* cuando sea posible para evitar copias
- Opcionalmente comprime los archivos en la cache para ahorrar sitio

11. Ccache (III). Limitaciones

- Sólo cachea los resultados de compilaciones de ficheros individuales de C/C++/Objective-C/Objective-C++.
- Solo funciona con GCC o compiladores que se comporten de forma similar.
- No se soportan algunas opciones de compilación. Si se detecta alguna de esas opciones, `ccache` invocará automáticamente al compilador real.

12. Ccache (IV). Formas de ejecutarlo

Usándolo como prefijo en las órdenes de compilación que empleemos, p.e.:

```
ccache gcc main.c -o main
```

Suplantando al compilador.

```
1        cp ccache /usr/local/bin/
2        ln -s ccache /usr/local/bin/gcc
3        ln -s ccache /usr/local/bin/g++
4        ln -s ccache /usr/local/bin/cc
5        ln -s ccache /usr/local/bin/c++
```

13. Ccache (V). ¿Cómo funciona?

VEAMOSLO CON UN PAR DE EJEMPLOS:

- *apt-get source tcc* o desde su página web.

```
1    time make
2    make clean
3    time make
4    ...
5    make clean
6    time make CC="ccache gcc"
7    make clean
8    time make CC="ccache gcc"
```


Por cierto, prueba a compilar `tcc` con el propio `tcc`: **¿Qué tiempos de compilación observas?** ¿Qué pasa si tratas de usar `tcc` con `ccache`?

- *apt-get source wmaker* o desde su página web.

Repetimos los mismos pasos que antes.

14. Ccache (VI). ¿Cómo funciona?

- Interrogamos la `cache`: `ccache -s`
- La borramos completamente (ojo!)...: `ccache -C`
- A nivel interno la manera que tiene `ccache` de saber cuándo recompilar un fuente y cuando no, es calculando la suma `hash` de varias informaciones que deberían ser únicas en cada compilación.
- Emplea el algoritmo `MD4`...hoy en día en entornos criptográficos es débil, pero para lo que lo usa `ccache` (obtener un índice) es suficiente.
- `ccache` puede trabajar en uno de dos modos:
  - *modo directo* : es el usado por defecto
  - *modo preprocesador* : se emplea si se define la variable de entorno `CCACHE_NODIRECT`.
- El modo directo es el más rápido ya que no ejecuta el preprocesador

15. Distcc (I)

- `ccache` es muy útil compilando todo en una máquina.
- Cuando el volumen del código que queremos compilar es muy grande...sería interesante poder distribuir la compilación del código.
- Para que esta distribución de la compilación se haga de manera **eficiente** debemos ayudarnos de una herramienta como distcc
- `distcc` distribuye la compilación de código `C`, `C++`, `Objective C` y `Objective C++` entre diversas máquinas conectadas en red.
- Se compone de un cliente (`distcc`) y de un servidor (`distccd`).
- Es muy fácil de instalar y de usar.
- Está pensado para ser usado con `gcc` pero en determinadas situaciones también puede ser usado con el compilador de C++ de Intel y el de Oracle.
- Se basa en una premisa: `distcc` siempre debe generar el mismo resultado que una compilación local.

16. Distcc (II). Funcionamiento

- También tiene dos modos de funcionamiento: *sencillo* y *bombeo* (**pump**).
- En **modo sencillo** `distcc` envía al servidor el código preprocesado y los argumentos para el compilador.
- En **modo bombeo** envía al servidor el fichero a compilar y todos los archivos `#include` (de manera recursiva) que necesita para ser compilado. De estos `#include` se excluyen aquellos que se encuentran en los directorios de cabeceras estandar del sistema. Se estima que en este modo de funcionamiento puede distribuir los ficheros a compilar hasta 10 veces más rápido que en el *modo sencillo*.
- La compilación es controlada por la máquina que ejecuta el cliente (el `pc` del programador) y es el cliente `distcc` ejecutado en esta máquina el encargado de enviar el código a compilar a aquellas máquinas de la red que ejecutan el servidor `distccd`.

17. Distcc (III). Funcionamiento

- `distcc` puede funcionar sobre TCP y sobre SSH, en este último caso es un poco más lento.
- El servidor `distccd` se puede ejecutar manualmente por un usuario o desde `inetd`.
- `distcc` está pensado para hacer uso de la opción ***-j N*** de `make`. Como regla sencilla el valor de ***N*** se puede ajustar al doble del número de CPUs disponibles en la red.

18. Distcc (IV). Guía rápida

- En cada servidor de compilación ejecutamos:

```
distccd --daemon --allow IP-permitida --allow IP2 ...
```

- Ponemos el nombre del computador/ip del servidor de compilación en la variable de entorno `DISTCC_HOSTS`:

```
export DISTCC_HOSTS="localhost red green blue"
```

- Compilamos:

```
make -j8 CC=distcc
```

19. Distcc (V). A tener en cuenta

- En `DISTCC_HOSTS` colocamos por orden de preferencia o rapidez los servidores preferidos o más rápidos primero. Estos servidores también pueden estar en el archivo:

```
$HOME/.distcc/hosts
```

- En determinados S.O. la configuración del servidor al inicio suele estar en:

```
/etc/default/distcc
```

- Es conveniente tener la misma versión del compilador en todas las máquinas de compilación. Sobre todo si trabajamos en C++ ya que el `ABI` puede cambiar, incluso entre versiones distintas de un mismo compilador.

20. ¿Se puede integrar `distcc` con `ccache`?

- Se puede, es sencillo ya que pueden funcionar como programas independientes que interactúan.
- A tener en cuenta:
  - No usar en `distcc` el modo de bombeo (pump). Así ejecutamos el preprocesador una sola vez.
  - Es mejor ejecutar `ccache` antes que `distcc`.

- Basta con hacer una llamada al compilador así:

```
export CCACHE_PREFIX="distcc"
CC="ccache gcc"
```

21. Parecido pero no-igual

- GNU Parallel permite poner en marcha trabajos en paralelo.
- Puedes ver el tutorial de uso en este enlace.
- En la página de la wikipedia tienes ejemplos sencillos de uso.
- Y en YouTube tienes vídeos de ejemplo.

22. Trabajo en grupo en clase

- En grupos de 4 personas echad un vistazo a IceCream. Explicad qué es y en qué se diferencia de *distcc*.

23. Prácticas individuales.

MAKE:

- Bien con una práctica tuya, bien con un código descargado, comprueba tiempos de ejecución de la compilación con diversos valores para `N` en **`-jN`**. ¿A partir de qué valores de `N` ya no supone una mejora sustancial el incremento en el número de trabajos en paralelo?
- ¿Hay fallos de compilación con ejecuciones en paralelo? Si es así trata de ver por qué se producen y procura solucionarlos.

CCACHE:

- Bien con una práctica tuya, bien con un código descargado, comprueba tiempos de ejecución de la compilación usando y sin usar `ccache`.

DISTCC:

- Configura un par de máquinas del laboratorio para aceptar solicitudes de `distcc` y bien con una práctica tuya, bien con un código descargado, comprueba tiempos de ejecución de la compilación al usar `distcc` y comparalos cuando se compila todo el código en tu máquina.

ENTREGA:

- La práctica se entregará en pracdlsi en las fechas allí indicadas.

24. Aclaraciones

EN NINGÚN CASO ESTAS TRANSPARENCIAS SON LA BIBLIOGRAFÍA DE LA ASIGNATURA.

- Debes estudiar, aclarar y ampliar los conceptos que en ellas encuentres empleando los enlaces web y bibliografía recomendada que puedes consultar en la página web de la ficha de la asignatura y en la web propia de la asignatura.