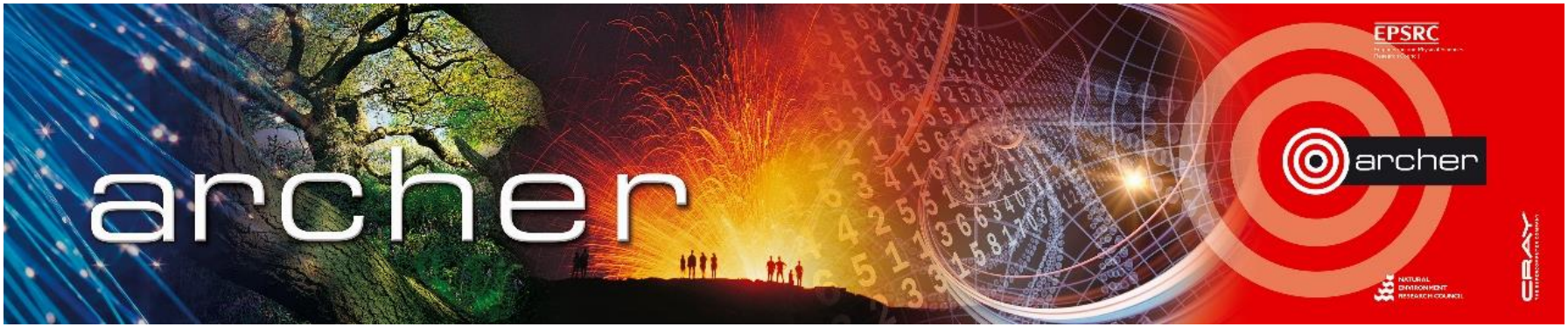


Welcome!

Virtual tutorial starts at 15:00 BST

archer

epcc

THE UNIVERSITY OF EDINBURGH

Make and Compilation

ARCHER Virtual Tutorial, Wed 11th April 2017

David Henty <d.henty@epcc.ed.ac.uk>

Reusing this material

archer

epcc

THE UNIVERSITY OF EDINBURGH

Compiling multiple files

- Compiling a simple code may be easy
  - cc program.c
  - cc –o program.exe program.c
- All but simplest programs have more than one source file
  - cc –o program.exe file1.c file2.c file3.c …
- This is wasteful so compile independently
  - cc –c file1.c
  - cc –c file2.c
  - …
- Then link the object files
  - cc –o program.exe file1.o file2.o file3.o …

The problems

- What if I changed file2.c (and maybe other files …)
  - cc –c file2.c
  - cc –o program.exe file1.o file2.o file3.o …
  - an error-prone procedure!
- Let's be safe
  - rm *.o
  - cc –c file1.c
  - cc –c file2.c
  - …
  - cc –o program.exe file1.o file2.o file3.o …
  - wasteful again!

More problems …

- Source files often depend on others, e.g. include files
- What if I edit include3.h
  - how do I know which files to recompile?
- Recompiling all files is slow and unnecessary
- Failing to recompile a file is disastrous
  - if your executable program does not reflect the current source code then debugging is impossible!
- Need a tool which:
  - remembers dependencies between files (in human readable form)
  - recompiles all files that need to be updated
  - recompiles the minimum number of files

Enter “make”

- User specifies ***pairwise*** dependencies between files
  - “program2.o ***depends on*** program2.c”
  - “program2.c ***depends on*** include3.h”
- Make works out the entire ***dependency tree***

- User specifies ***pairwise*** rules for resolving dependencies
  - “to update program2.o run the compiler on program2.c”
- All this information is stored in a ***Makefile***
  - tells make ***how*** to update files

- How does make know ***when*** to update?
  - Make compares the date stamps of files

Example 1: family1

- Three types of file:
  - david.self
  - david.parent
  - david.child
- Dependencies
  - self is younger than parent (created more recently)
  - child is younger than self
- One final output file
  - **davidfamily** contains a date-ordered listing of the source files
  - if correct, order should be: parent; self; child.
- Update rule is to copy: **cp david.self david.child**

Example 2: family2

- Imagine another family: sally

- Wasteful to specify ***explict*** rules all over again
  - file1.o: file1.c
    - cc –c file1.c
  - file2.o: file2.c
    - cc –c file2.c
  - file3.o: file3.c
    - cc –c file2.c
  - …
- Make also understands ***generic*** rules based on suffix
  - “this is how you create any child”
  - applies to david.child ***and*** sally.child

Example 3: C traffic code

- Illustrates use of variables
  - dependencies on header files
  - global change of C compiler by updating a single line
  - creation of one list of variables from another
- Some magic variables
  - e.g. “The thing on left hand side of expression you’re working on”
- Default rule
  - the first one in the Makefile, conventionally ***all***
- Dummy rules
  - housekeeping, e.g. delete junk with ***clean***
  - to find out object files in variable OBJ, put in a rule to print it out

Example 4: Fortran traffic code

- The same format as the C version
- Slightly complicated by use of modules
- Possible to create relatively simple generic Makefiles
  - extend as appropriate for real cases

The dirty linen

- Tabs have magic significance in Makefiles

- Can’t easily cut and paste them from the web!

- GNU make spots this:

```
user@archer> make david.child
Makefile: *** missing separator (did you
mean TAB instead of 8 spaces?). Stop.
```

Tricks and tips

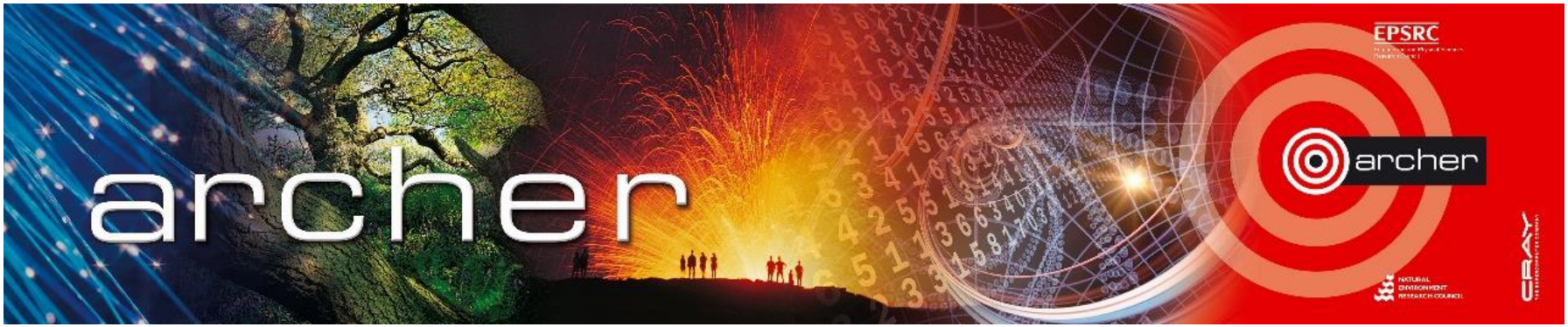
- You can make anything under control of make
  - e.g. **`make file.o`**
- **`make -n`** prints out ***what*** make would do without doing it
- **`make --debug`** prints out ***why*** make is doing what it does
  - can ask for more verbose output if you want
- update rules can print debug info
  - **`echo "updating $@ from $<"; cp $< $@`**

Complications

- Fortran modules
  - more sophisticated than C header files but harder to cope with
- What if I have hundreds of header files
  - tools like “makedepend” can write the rules for you
- GNU autotools (e.g. configure) produce Makefiles
  - unfortunately, not human understandable!
- Make has many ***implicit*** (default) rules and variables
  - I prefer makefiles to be explicit and not assume these

ARCHER

- Want Makefile that works for all programming environments
  - but different compilers have different options
- Can enquire environment variables within Makefile
  - e.g. whether $(CRAY_PRGENVCRAY)=loaded
- Change of compiler module invisible to make
  - module switch PrgEnv-cray PrgEnv-intel
  - make clean
  - make



Goodbye!

Virtual tutorial has finished

archer

epcc

THE UNIVERSITY OF EDINBURGH