

Building, installing and running software

Day three

Bob Dowling
University Computing Service

www.ucs.cam.ac.uk/docs/course-notes/unix-courses
training.cam.ac.uk/ucs/

UCS

1

Welcome back to the third session of the course.

Progress so far

`${HOME}/sw`

Builds:

- configure
- make
- make install

Makefile

Libraries

- build time
- run time

UCS



- Using libraries
- Simple make
- Configured builds
- Unpacking
- Location

2

So let's take stock of what we have done.

We have our own location for our own software. We have built software using the `configure` system and seen easy examples of `make` without the support of a `configure` system. We have seen how to use libraries installed in our personal `${HOME}/sw` directory trees.

Before I move on to this session's material, are there any questions?

Today

1. Real world example
A bad Makefile
2. Recursive make
3. Rewrite Makefile from scratch

What we will do today is to take what we learnt yesterday and put it to good use. We will take a `Makefile` from a real world build system and make it work. This make system is a good example of a bad example. We are going to see what is required in practice to fix other people's `Makefiles` that don't live up to the high ideals we laid down previously.

We will look at one extra technique in `make`, “recursive make”. This is useful for large build systems that span several subdirectories each with their own targets.

Finally we will see if what we learnt previously, combined with this new technique is sufficient to completely replace the poor quality `Makefiles` from our real world example.

Meet the enemy

LAPACK **Linear Algebra Package**

BLAS **Basic Linear Algebra Subprograms**
(Fortran)

Well respected pair of libraries

Extensively used

Absolute \$@%#! to build

UCS

4

So let's meet this example of a really bad build system.

The Linear Algebra Package, “lapack”, is a well-respected set of Fortran routines. It is built on top of the Basic Linear Algebra Subprograms library, “blas”, whose C language equivalent, cblas, we met in the previous session.

The routines are excellent. The whole world uses them with confidence. The build system, however, is appalling. To be fair, it is a mish-mash of historical relics from different people with different styles all glued together. The software managers have simply not done the necessary maintenance on it.

Worked example: unpacking

```
$ cd /tmp/building
```

```
$ tar -x -f ${HOME}/LAPACK.tgz
```

```
$ cd LAPACK
```

We'll unpack it in the usual way.

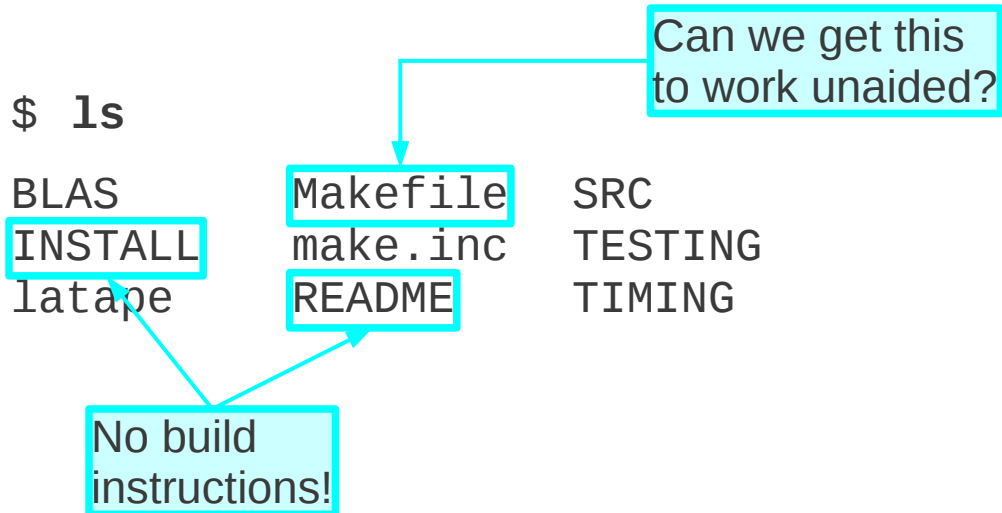
Don't forget that the TAB key will extend partial file names for you as best they can be.

```
$ cd /tmp/building
```

```
$ tar -x -f ${HOME}/LAPACK.tgz
```

```
$ cd LAPACK/
```

Worked example: first look



Inside the LAPACK directory we have no configure script. We do have a `Makefile`, however, so we can expect to be running `make` to build the software.

There is a `README` but it contains no build instructions!

The `INSTALL` node is actually a directory rather than a file of instructions as is common in more modern software.

So, can we get the `Makefile` to work on our own?

INSTALL *directory*

```
$ ls INSTALL/
```

```
diamch.f          Makefile
diamchtst.f      make.inc.ALPHA
dsecnd.f         make.inc.HPPA
dsecnd.f.RS6K   make.inc.IRIX64
dsecndtst.f     make.inc.LINUX
lawn81.pdf       make.inc.O2K
lawn81.ps       make.inc.pghpf
lawn81.tex      make.inc.RS6K
lsame.f         make.inc.SGI5
lsametst.f     make.inc.SUN4
```

Per-architecture
include files



UCS

7

The INSTALL directory does contain some interesting files. We see, in particular, a collection of the “make.inc” files we saw in the top-level directory. If we look inside the LINUX version we see it sets make macros for use on a Linux system. There is some limited documentation in the lawn81.* files but they’re not build instructions as we would understand them.

LAPACK Makefile

```
# Top Level Makefile for LAPACK  
# Version 3.0  
# June 30, 1999
```

Comments

```
include make.inc
```

Include the
contents of
another file

```
all: install lib testing ...
```

Standard
target

UCS

8

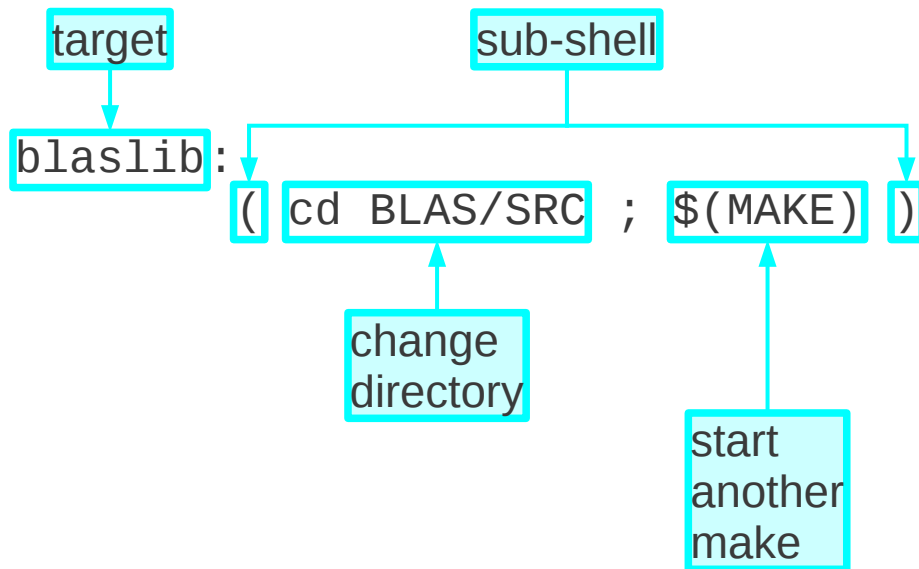
So let's return to the top-level Makefile and feel our way.

The first active line of the Makefile tells us what the `make.inc` file is for; it's an "include" file to set a whole bunch of parameters.

The next active line is the default target: "all". At least they got that right.

It's first dependency is called "install" so they must be using that for something other than the standard. Ho hum.

LAPACK Makefile



UCS

9

Further down the Makefile we see a bunch of lines like this.

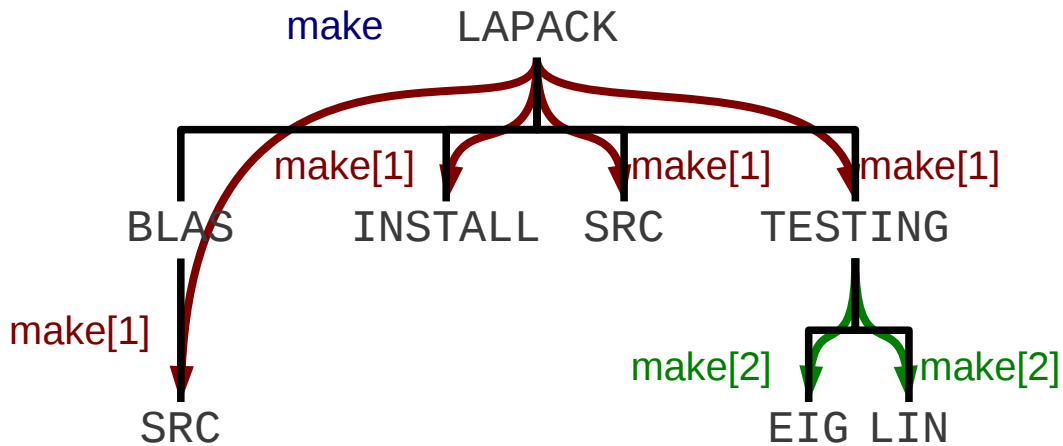
```
blaslib:  
    ( cd BLAS/SRC; $(MAKE) )
```

This is an example of a “recursive make”. The action for the target involves moving to another directory and running make again.

Makefiles tend to use “\$(MAKE)” rather than “make” internally because the make program being used may not be called that. “gmake” is a not uncommon name for “**GNU make**”. “**pmake**” is a “**parallel make**” that can build several targets simultaneously, etc. The MAKE macro always takes on the name of the currently running make program.

The parentheses (round brackets) start a sub-shell. They are entirely unnecessary. Every action line in a Makefile is automatically run in its own sub-shell.

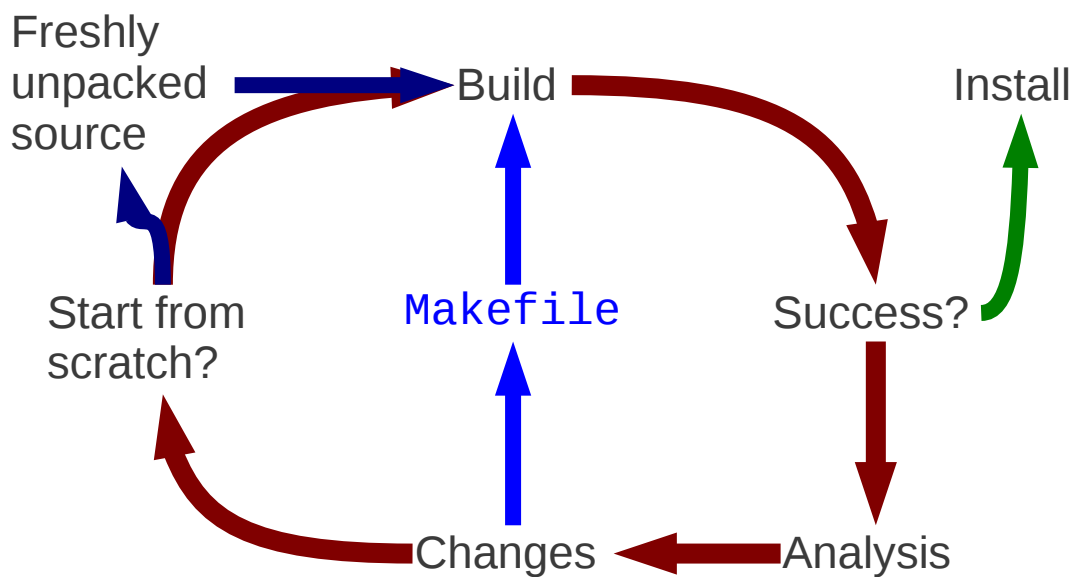
Recursive make



Of course, once `make` jumps to another directory and runs `make` there there is nothing to stop that `Makefile` doing the same again. This is what makes it truly recursive. We will see that the top-level `make` enters a bunch of directories (including one two levels down in the directory tree) and runs `make` in each of them. All these are first-level sub-makes and will be represented in the output as “`make[1]`”. One of these, the one that enters the `TESTING` directory, then calls `make` a couple more times. These are second level sub-makes and are represented as “`make[2]`” in the output.

The “level” of a sub-make is a function of how many times `make` has called `make`, not of how many subdirectories deep the process has descended.

Iterative development



UCS

11

Our builds aren't going to succeed first time. We are going to have to hit them quite a few times. The trick is to be systematic about it. Treat this as a series of experiments in a cool-headed, scientific fashion and you won't go wrong.

We start with a clean unpacking of the source code.

Then we try a build with the current `Makefile`, recording the output in a log file.

Then we look to see if it succeeded. If it did we move on to the installation.

If it failed we analyse the results (the log file and the build directory's contents) to find out why.

Based on this analysis we draw two conclusions:

What change or changes do we need to make to the `Makefile`?

Can we pick up with our current build tree or do we need to start from a freshly unpacked build tree?


Then we try again.

This is where your lab books really come into their own. Keeping a written record of what you do and what the build process does in retaliation really helps.

Default make.inc

```
# The machine (platform) identifier to  
# append to the library names
```

```
PLAT = _SUN4SOL2
```



Sun4 hardware
Solaris 2 operating system

If we look at the default make.inc file we see that it is not suitable for our (Linux) system but for Solaris 2 systems.

Build #1



Record change
to file in lab book

```
$ cp INSTALL/make.inc.LINUX make.inc
```

```
$ make &> make01.log
```

Log for *this* attempt

Output, errors,
everything

We copy the Linux `make.inc` file into place, record that we have done so in the lab book, and then run `make`.

We want to capture *all* the output of the `make` process in a log file. We will number our log files to keep each build attempt quite separate. I use two digits in my versioning because I'm a pessimist. We won't go that far this afternoon. Note that the redirector “&>” redirects both the standard output and the error messages to the same file.

If you can't get it to work and turn to an expert for help then they should ask to see your logs and your lab notes. If you can't show them both then they have every right to laugh in your face and show you the door. (Though bribing them with chocolates has been known to work.)

Success? #1

```
$ make &> make01.log
```

Fails almost
instantly

We need to
read the log

The build attempt fails almost immediately. So we will read the log to see what went wrong.

Reading a make log

make stops as soon as it hit an error

Start at the end & work backwards

“Rewind”

There is a knack to reading a log file from make. Because make (unless otherwise directed) stops on the first error the error is always at the end of the file the best approach is to start at the last line and to work backwards, finding how we got to that state.

Analysis #1: last line

```
make: *** [lapacklib] Error 2
```

target

Error message

Error 2: Error in a sub-make or sub-shell

So let's look at the last line of the log file.

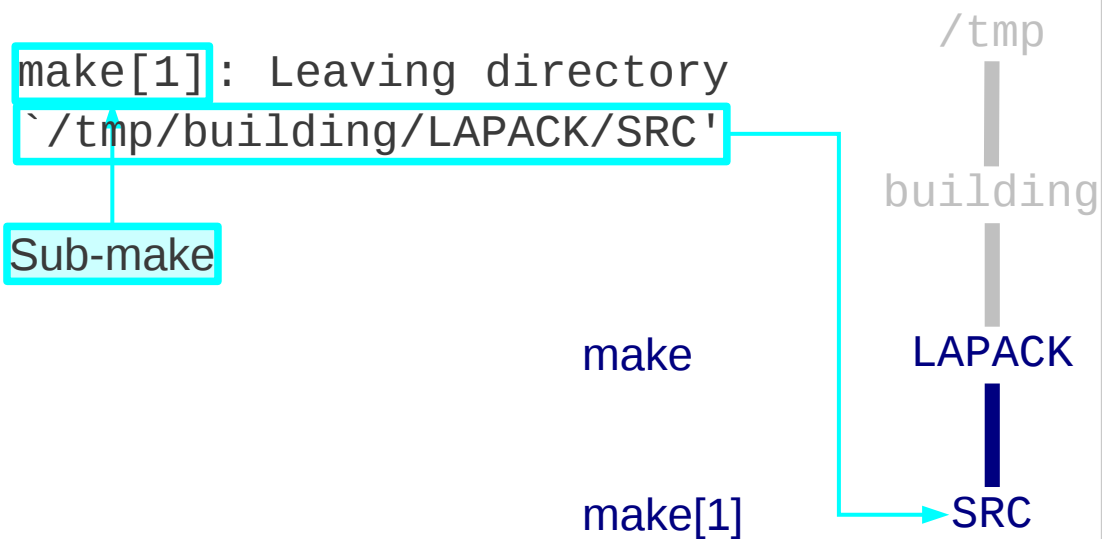
This is a message from make itself, as opposed to from one of the various programs make launches.

The word in square brackets is the target being built by make when it caught the error. The text that follows it is make's error message.

So in this case the target make was trying to build was "lapacklib".

Its error message is the less than helpful "Error 2". This is code for an error in a sub-make or a sub-shell and we need to move backwards into the log file to find out what happened in the sub-system to cause the original error.

Analysis #1: penultimate line



Analysis #1: antepenultimate line

```
make[1]: *** [[sgbbrd.o] Error 127
```

Same
sub-make

target

error message

Error 127: "Command not found"

UCS

18

So we work backwards one more line.

This time we see that the sub-make was trying to build the `sgbbrd.o` target and failed with "Error 127".

Error code 127 means "command not found". This isn't a make-specific code. You can get the same from the plain shell.

```
$ qwerty
```

```
-bash: qwerty: command not found
```

```
$ echo $?
```

```
127
```

(The automatically maintained variable "?" carries the return code from the last command run.)

Analysis #1: previous line

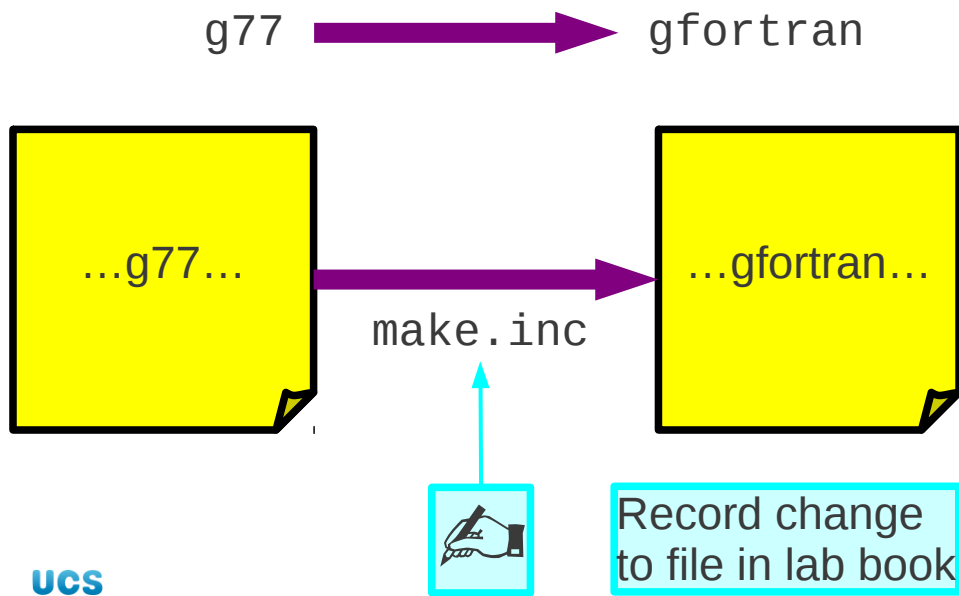
make[1]: g77: Command not found

Told you so! 😊

This command

To find out which command wasn't found we move back one more line. Here we see that the sub-make couldn't find the "g77" command. "g77" is the old name for the **GNU Fortran '77** compiler. The **GNU Fortran** compiler now compiles much more than Fortran '77 so has been renamed "gfortran".

Changes #1: previous line



The Fortran compiler is one of the macros set in the `make.inc` file. We will change this and record the fact that we have had to in the lab book.

Start from scratch?

Nothing wrong with what's done so far.

Whenever we make a change to the Makefile (as we have just done indirectly) we must ask ourselves whether there is a need to start from scratch or whether we can just call make again.

All that has happened is that we have failed to find a compiler. There's nothing wrong with what we have built so far (mostly because we haven't built anything) so we can just pick up where we left off.

Build #2

```
$ make &> make02.log
```

5 (ish)
minutes

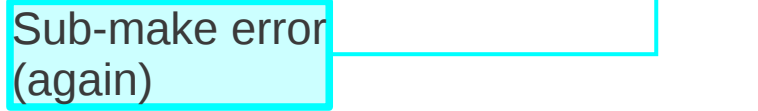
Distinct
log file

So we relaunch, logging to a different file, `make02.log`. This one should roughly five minutes.

Analysis #2: Last line

```
make: *** [testing] Error 2
```

Sub-make error
(again)



Again, we turn to the final line of the make log to see that the error happened in a sub-make again. It's time to start working backwards through the log file again.

Analysis #2:

make[1]: Leaving directory

``/tmp/building/LAPACK/TESTING'`

make

make[1]

/tmp

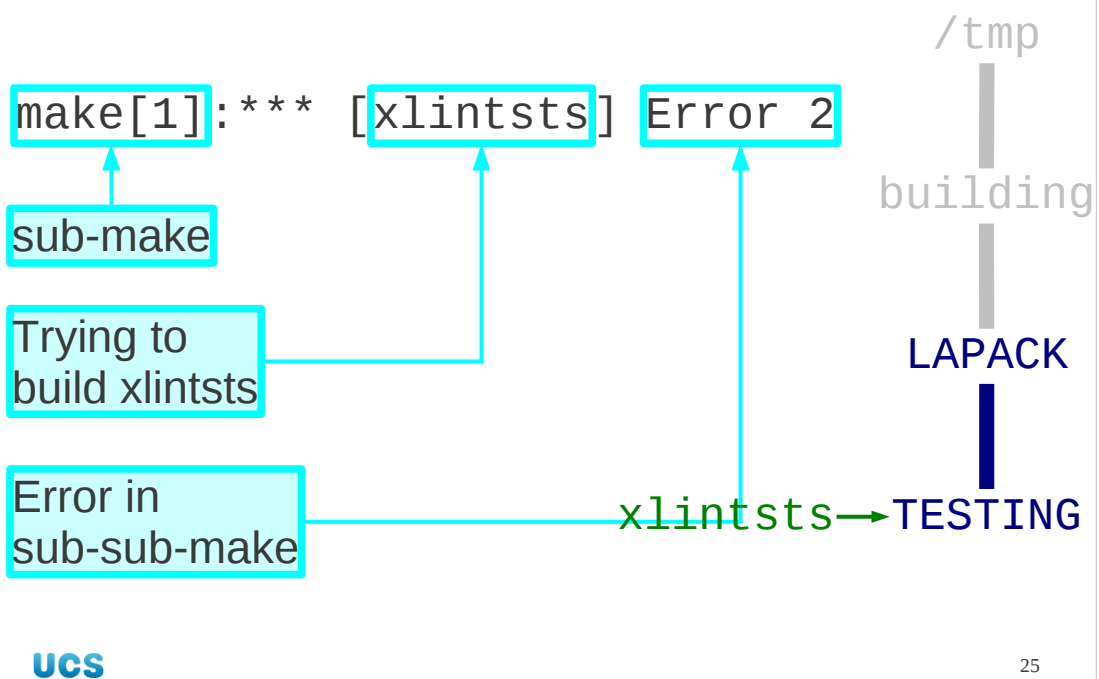
building

LAPACK

TESTING

Again the previous line is the announcement of the departure from the directory where the sub-make was working.

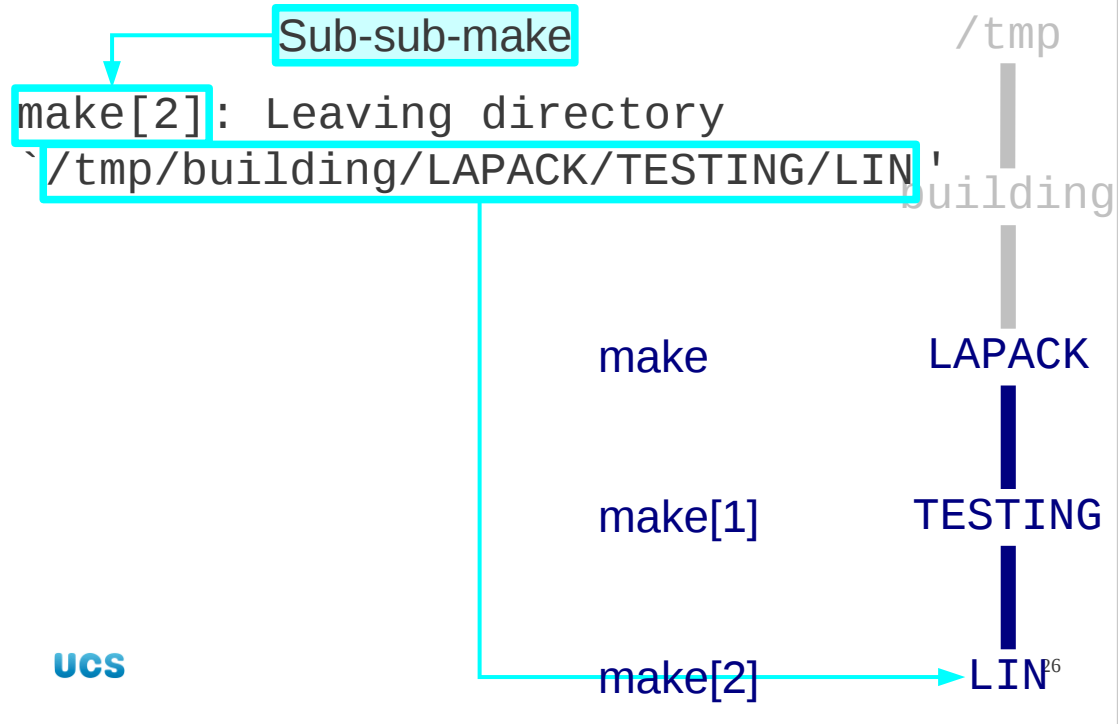
Analysis #2:



The line prior to that is the line identifying the target being built and the error reported by the sub-make. The target being built is `xlintsts` in the `/tmp/building/LAPACK/TESTING` directory so is the file `/tmp/building/LAPACK/TESTING/xlintsts`.

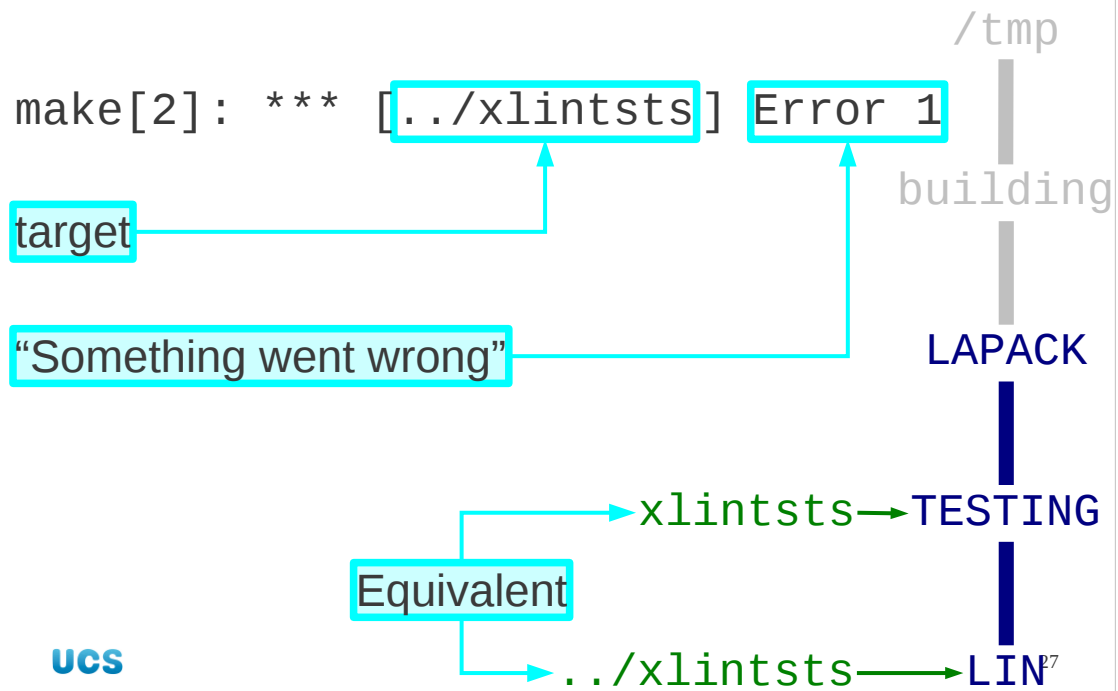
The error being reported by sub-make is that there was an error in its own sub-sub-make.

Analysis #2:



Here we meet the signing off from the second-level sub-make and its departure from the LIN sub-directory of the TESTING directory where the first-level sub-make had been operating.

Analysis #2:



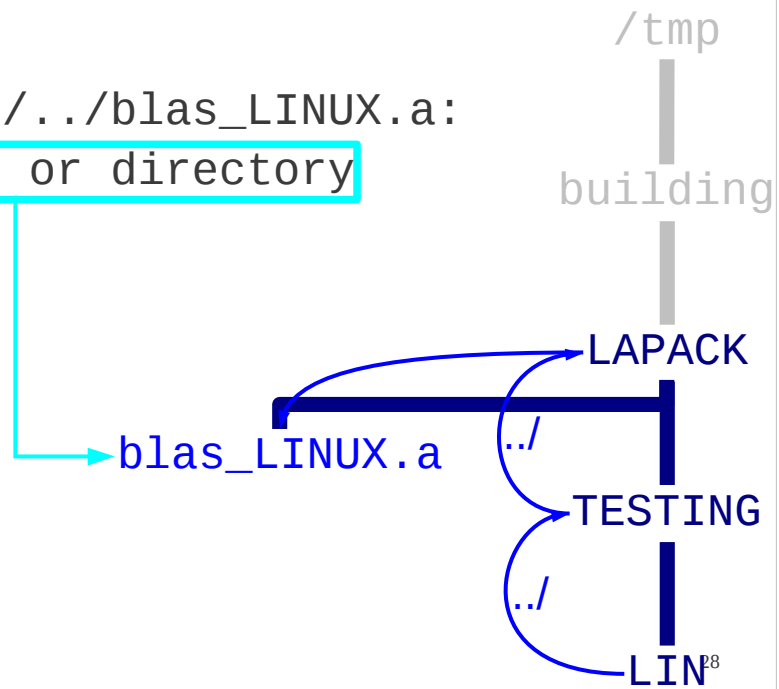
So we go back another line. This tells us that the sub-sub-make was trying to build a target “../xlintsts”. This was done from the directory /tmp/building/LAPACK/TESTING/LIN so is an attempt to build the file /tmp/building/LAPACK/TESTING/xlintsts. This is exactly the same file as corresponded to the target of the sub-make we had before.

The error message is “Error 1” which is simply “something went wrong with the command”. It is progress, however, because we are no longer sinking through layers of make recursion. We have reached our target, the command that failed.

Analysis #2:

```
gfortran: ../../blas_LINUX.a:  
No such file or directory
```

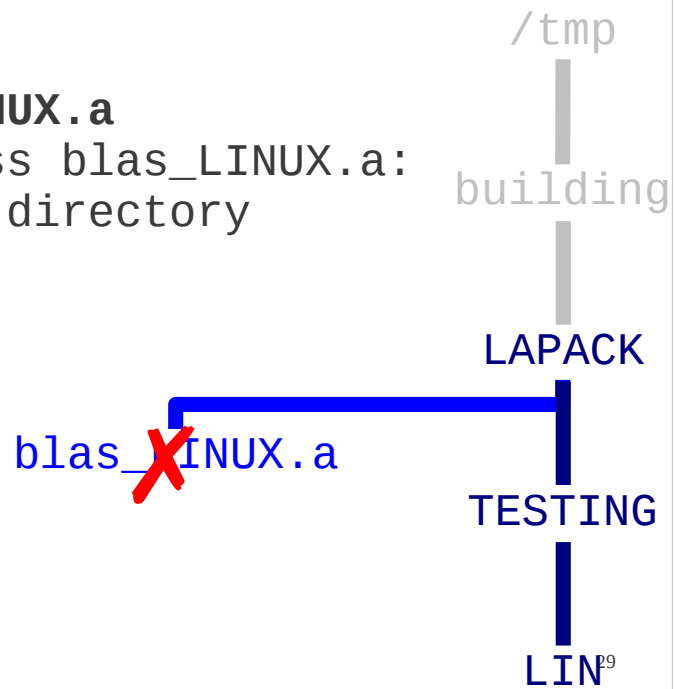
UCS



The error message comes from `gfortran` and reports that the compiler was unable to find a file `../../blas_LINUX.a`. The sub-sub-make was running in `/tmp/building/LAPACK/TESTING/LIN` so the relative file `../../blas_LINUX.a` has absolute file name `/tmp/building/LAPACK/blas_LINUX.a`. That's the name of the file that should exist but doesn't.

Analysis #2:

```
$ ls -l blas_LINUX.a
ls: cannot access blas_LINUX.a:
No such file or directory
```



UCS

So first we should check that the file really doesn't exist. It doesn't.

We ought to check that it hasn't been built in some other location

```
$ find /tmp/building/LAPACK -name blas_LINUX.a -print
```

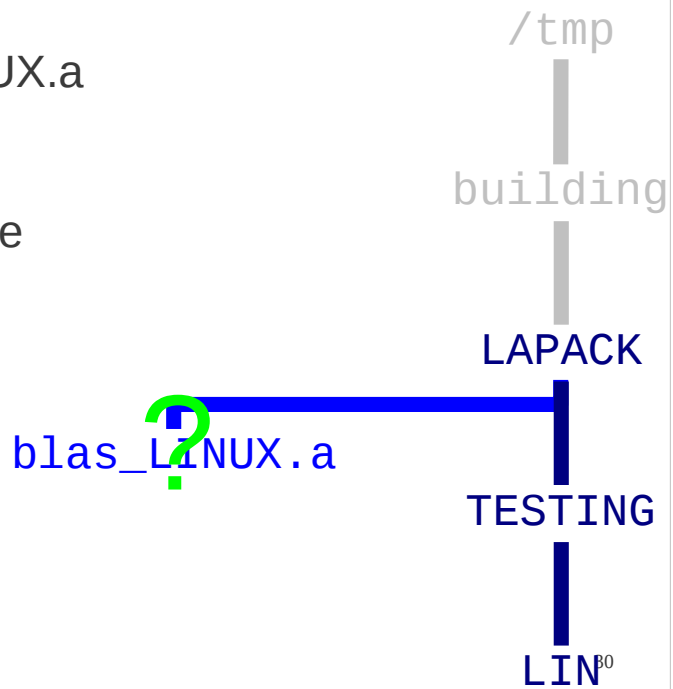
```
$
```

It hasn't been.

Analysis #2:

Why does blas_LINUX.a not exist?

Because the Makefile doesn't build it!



UCS

So why doesn't it exist?

The typical answer — and the correct one in this case — is that the build system was never asked to make it. In a correctly written Makefile there should be a dependency requiring that this library exist before any targets get built that use it. This has not happened here.

So why isn't it built?

Analysis #2:

Why doesn't the
Makefile build it?

Because it's
commented out!

```
lib: lapacklib tmglib  
#lib: blaslib lapacklib tmglib
```

Makefile

UCS

31

If we look near the top of the Makefile we see a suspicious pair of lines

```
lib: lapacklib tmglib  
#lib: blaslib lapacklib tmglib
```

and the name “blaslib” is very suggestive.

This pair of lines suggests that there are two ways to build this package. One line presumes that there is already a BLAS library (the default) and the other line that we need to build it (commented out).

Changes #2

```
lib: lapacklib tmglib  
#lib: blaslib lapacklib tmglib
```

Makefile



```
#lib: lapacklib tmglib  
lib: blaslib lapacklib tmglib
```

Makefile

32

So, again we make a change which we record in our lab books.

Start from scratch?

Nothing wrong with what's done so far.

So, do we need to build from scratch?

No. All that has happened is that building stopped early because a library was not present. If we build that library then the process should be able to continue apace.

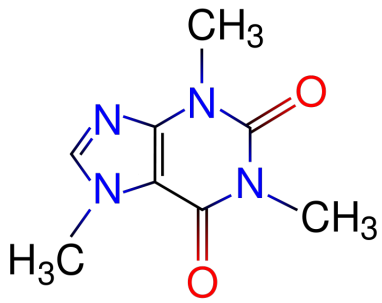
Build #3

```
$ make &> make03.log
```

**5-10
minutes**

Third log file

So we start the make process for a third time, into a third log file. This one will take between five and ten minutes. Start the build and then take a short coffee break.



“Sleep when
you're dead.”

Ten minutes

UCS



Ten minutes coffee break.

Analysis #3: Last line

```
make: *** [timing] Error 2
```

target

sub-make error

So we return to our (third) make log file. This time the error happened in the “timing” target and (again) the error happened in a sub-make.

Analysis #3

make[1]: Leaving directory

`~/tmp/building/LAPACK/TIMING`

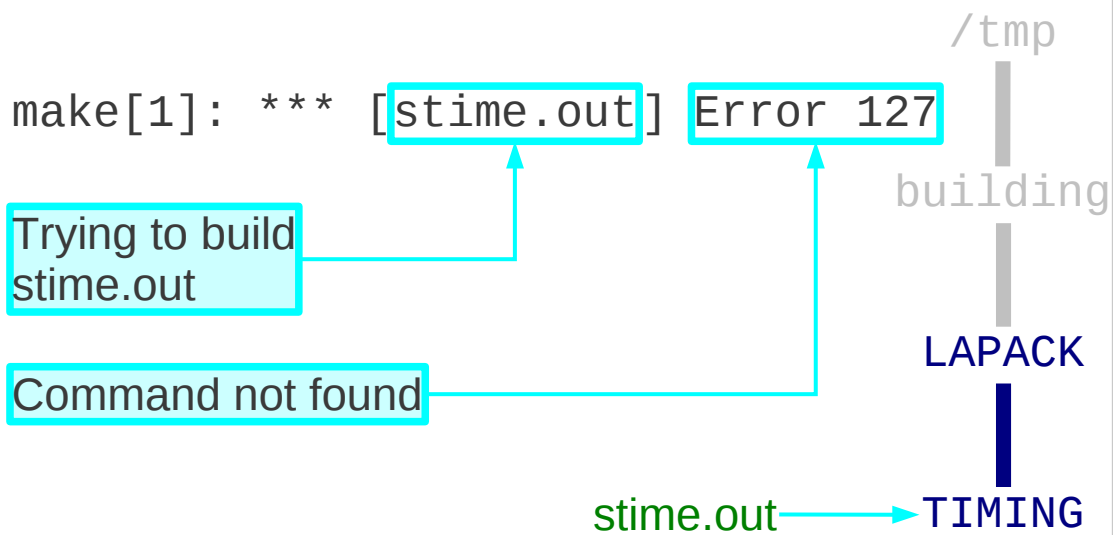
make

make[1]



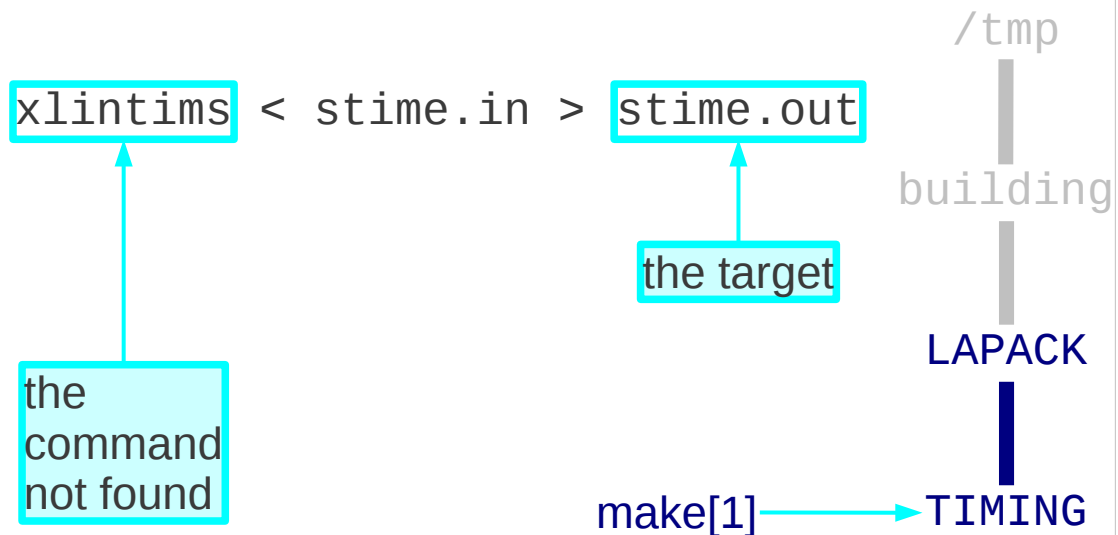
The sub-make filed when it was running in the directory `~/tmp/building/LAPACK/TIMING`.

Analysis #3



The sub-make was trying to build `stime.out` and suffered an “error 127” which we now know to mean “command not found”.

Analysis #3



If we roll back one more line we find the command the sub-make was trying to run. We see the command being run, `xlintims`, which is the command we worked to build last time round. This is the command that can't be found.

We also see that its output is the file `stime.out` which is the target that the sub-make is trying to build.

Analysis #3

xlintims ?

```
$ find . -name xlintims
```

We are here

/tmp
|
building

LAPACK

make[1]

TIMING

So why couldn't the file be found? Does it exist?

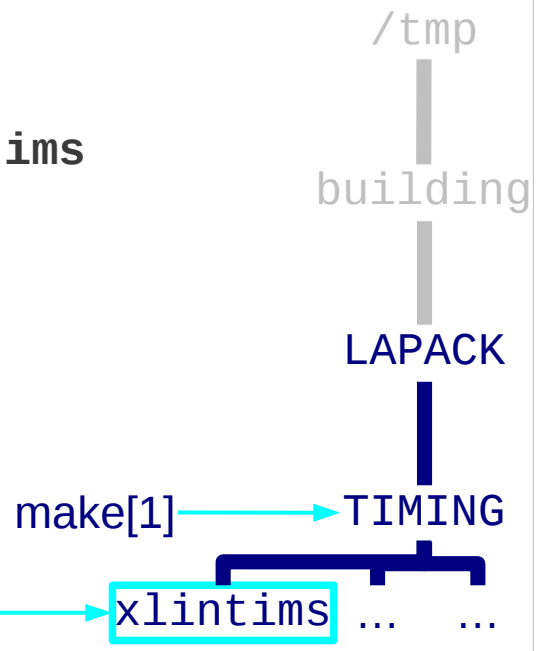
Analysis #3

`xlintims` ?

```
$ find . -name xlintims
```

```
./TIMING/xlintims
```

It's there!

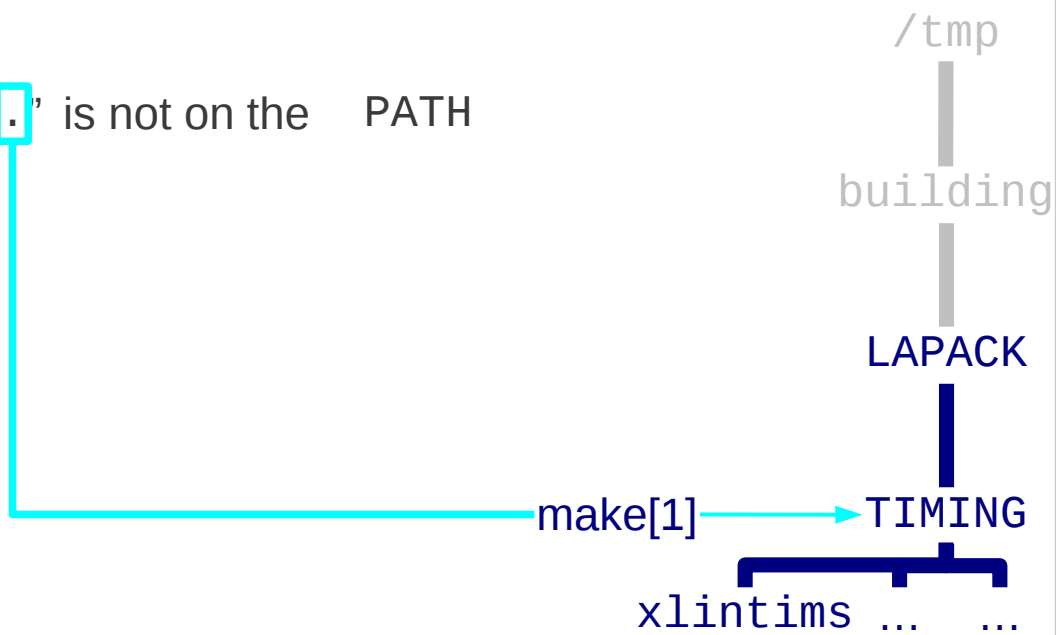


UCS

Yes. It does exist. It's in the directory `/tmp/building/LAPACK/TIMING` where the command was running.

Analysis #3

“.” is not on the PATH



UCS

42

The command is in the directory `/tmp/building/LAPACK/TIMING` where the command was running. And that's the nub of the problem. For good reasons, we don't have `.` (the current working directory) on our `PATH` along which commands are searched for.

Changes #3

```
$ export PATH="${PATH}:"
```



Add "." to PATH



UCS

Do *not* make this change permanent!

43

So the improvement we need to make this time is to add "." to our current PATH. Note that we are not changing our permanent PATH by adding this instruction to our `~/.bashrc` files. This is not a safe setting.

If you ever need to run a command out of the current working directory (e.g. in a `Makefile`) refer to it as `./command` and you will never need this hack.

This is a change to your environment required to build the software. It can't be recorded in a saved `Makefile` or other configuration file; you must record this in your lab book's notes.

Start from scratch?

Nothing wrong with what's done so far?

There *shouldn't* be anything wrong...

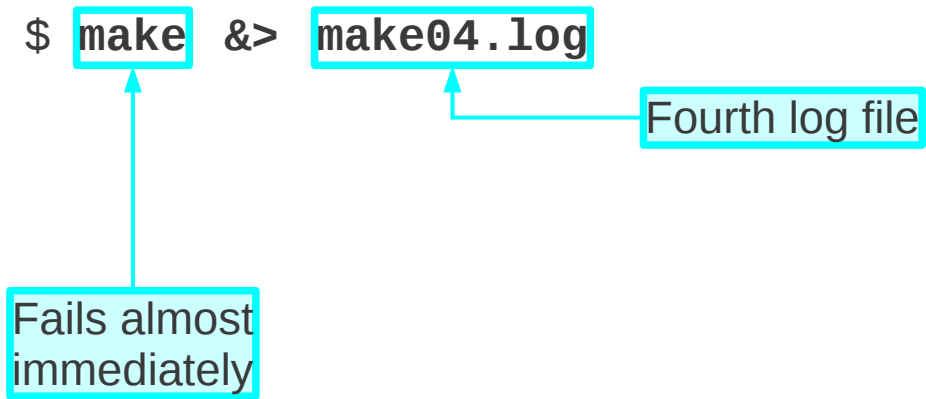
Once again we ask ourselves the question “do we need to start from scratch?” We shouldn't need to restart from scratch. An action failed. In a properly written Makefile this means that the target file is cleanly not created. We can pick up where we left off.

But this isn't a well-written Makefile...

Build #4

```
$ make &> make04.log
```

Fails almost
immediately



```
graph TD; A[Fails almost immediately] --> B[make]; B --> C[make04.log]; D[Fourth log file] --> C;
```

Fourth log file

We run make a fourth time, logging to a fourth log file. It fails almost immediately.

Analysis #4: Last line

```
make: *** [blas_testing] Error 2
```

↑
Top-level
make

Here we go again!

There was an error in a sub-make or a sub-shell when make tried to build the `blas_testing` target.

Analysis #4: The error message

```
At line 130 of file zblat2.f  
Fortran runtime error: File exists
```

Need to look at the larger error message.

Need to look at the action being taken.

Need to look at the zblat2.f file.

We move back one line and immediately run into the error message from the Fortran system. This is a runtime error, not a build error. Our make system has built a program which it runs later in the process. When it runs it fails. To be precise it fails at line 130 of the program in zblat2.f.

Analysis #4: The Makefile

```
blas_testing:
    ...
    ( cd BLAS; ./xblat2s < sblat2.in ; \
      ./xblat2d < dblat2.in ; \
      ./xblat2c < cblat2.in ; \
      ./xblat2z < zblat2.in )
    ...
```

Makefile

/tmp/building/LAPACK/BLAS

Source of the last
error message

This is new so we will look in the Makefile to see the instruction for the `blas_testing` target.

You may take this as a textbook example of how not to write a rule in a Makefile. If we look at the actions we see five commands (including the change of directory) all chained together with no mechanism to stop if something gives an error. There is also no mention of any files they create so `make` can't clean up if anything does go wrong.

Analysis #4: The program

```
...  
*      Read name and unit number for  
*      summary output file and open file.  
      READ( NIN, FMT = * )SUMMRY  
      READ( NIN, FMT = * )NOUT  
      OPEN( NOUT, FILE = SUMMRY, STATUS = 'NEW' )  
      NOUTC = NOUT  
*  
...
```

zblat2.f

Line 130

File must not
already exist!

UCS



49

Let's look at line 130 of the offending file.

Here I will have to share some Fortran with you. The option “STATUS= 'NEW' ” means that the file must not already exist. But because the Makefile doesn't mention what this file is it has no way to know not to run this program for a second time.

We will record this detail in our lab books and, with a heavy heart, we will start from scratch.

Changes #4

None!

Just start from scratch!

A well-written Makefile reduces the number of times this happens. Starting from scratch should only be necessary when changes are made that cause previously created files to clash.

Start from scratch!

```
$ cp make*.log ..
```

Copy the log files out of LAPACK

```
$ cd ..
```

Get out of LAPACK ourselves

```
$ rm -rf LAPACK/
```

Remove LAPACK

```
$ tar -xf ${HOME}/LAPACK.tgz
```

Create a fresh copy of LAPACK

First, we will copy our existing log files out of the LAPACK directory. Then we get out ourselves and remove the entire directory tree. We can't trust "make clean" with a Makefile this bad. Finally we will unpack a fresh copy.

Build #5

Repeat the previous changes!

1. INSTALL/make.inc.LINUX → make.inc
2. g77 → gfortran
3. “lib” target in Makefile
4. Check “.” on PATH
5. **make install lib &> make05.log**

Cheat to save time

>1hr → ~10mins

UCS

52

Now it's time to read the notes you have been recording in your lab books. You have been, haven't you?

```
$ cd LAPACK
```

```
$ cp INSTALL/make.inc.LINUX make.inc
```

```
$ vi make.inc
```

```
$ vi Makefile
```

```
$ echo ${PATH}
```

```
/home/rjd4/sw/bin:...:.
```

```
$ make install lib &> make05.log
```

g77 → gfortran

swap libs target

“.” is on the PATH

We cheat in the last step. The build will succeed. However the testing and the timing take an hour or more so we just do the build. If you want to check this then please feel free to do the build in your own time and simply run “make”.

**A “multiple purpose
generally recognized
as safe food substance”**

Five minutes

UCS



Let's have a ten minute break.

The quote comes from the USA Code of Federal Regulations 21CFR182.1180.
[http://edocket.access.gpo.gov/cfr_2003/aprqtr/21cfr182.1180.htm]

Installation

```
$ make install
```

```
$ ls *.a
```

```
blas_LINUX.a  
lapack_LINUX.a  
tmglib_LINUX.a
```

basic linear algebra
subprograms

linear algebra
package

timing library

We'll leave it running and move to a copy I prepared earlier.

We already know that the “install” target does nothing of the sort. Our build has made three libraries, the BLAS library, the LAPACK library and a timing library which we're not interested in.

Installation

```
$ install blas_LINUX.a  
  ${HOME}/sw/lib/libblas.a
```



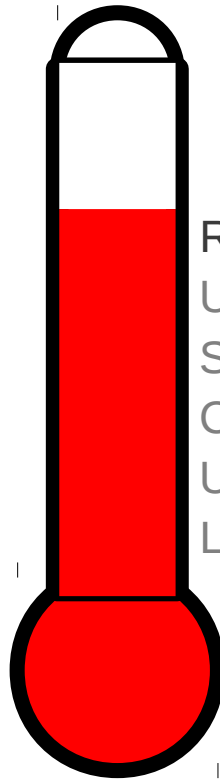
```
$ install lapack_LINUX.a  
  ${HOME}/sw/lib/liblapack.a
```



We will do our own installation with the good old-fashioned “install” command, taking the opportunity to give the libraries more traditional names. Note that these commands would have failed if the directory `${HOME}/sw/lib` did not already exist.

Progress

Real world
example
(almost)



Real world example
Using libraries
Simple make
Configured builds
Unpacking
Location

And that brings us to the end of the most painful part of the course. We have successfully bludgeoned a real-world, poor-quality build system into life.

Makefiles from scratch

LAPACK
Makefiles
are *awful*

It's a lot
of work

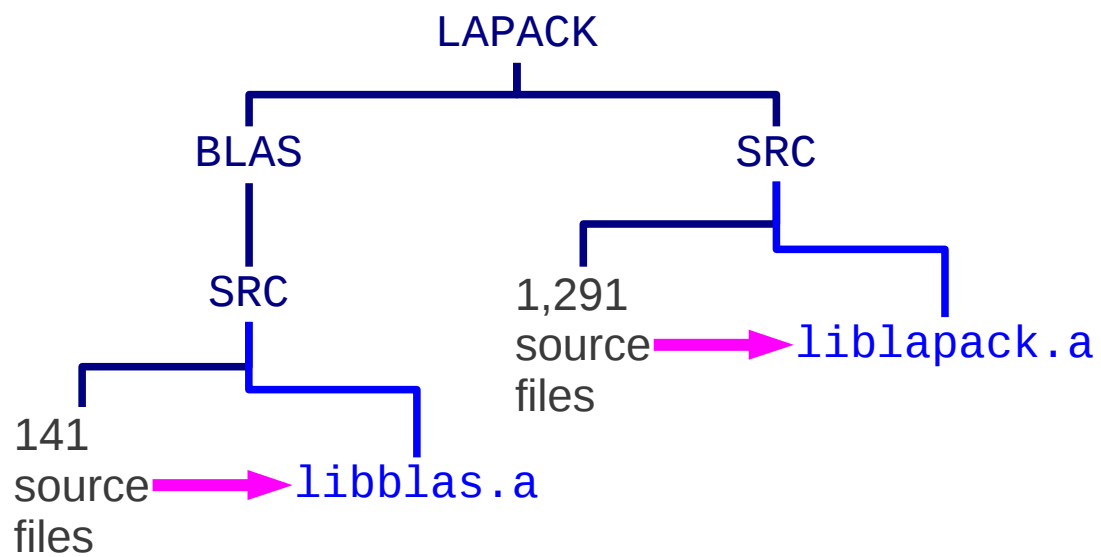
VS.

We know
better
principles

Should be a
last resort

To finish off this course we are going to rewrite the Makefiles from scratch. We are going to do this to convince ourselves that we can, that we are good enough. We only have one more trick to learn and this example will provide us with the opportunity to learn it. I want to emphasise that we are doing this for the course. In the “real world” (as if the University could ever be described as such) this is very much a last resort. It can be a lot of work.

What are we building?



We only want two libraries. We could perform the self tests and timing tests if we wanted, but this is a teaching exercise so I'm keeping its size under control. We will take these two libraries one at a time.

libblas.a

1. Keep the lists of file names
2. Ditch the rest

```
SBLAS1 = isamax.o sasum.o saxpy.o scopy.o \  
         sdot.o snrm2.o srot.o srotg.o \  
         sscal.o sswap.o  
...  
ZBLAS3 = zgemm.o zsymm.o zsyrk.o zsyr2k.o \  
         ztrmm.o ztrsm.o zhemm.o zherk.o \  
         zher2k.o
```

Makefile

/tmp

building

LAPACK

BLAS

SRC

59

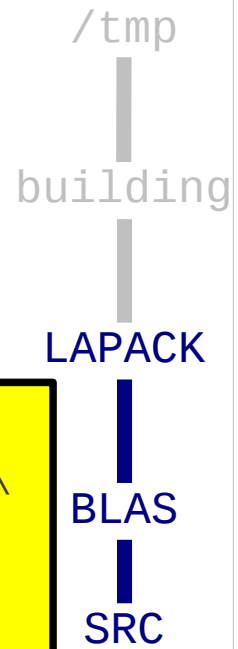
We start with libblas as it is the base library that liblapack depends on. There are 141 Fortran source files in its directory. To save us a good deal of typing, we will salvage from the provided Makefile just the definitions of the macros that list file names. We will discard everything else.

libblas.a

3. Glue all the object files together for convenience

```
OBJECTS = $(SBLAS1) $(CBLAS1) $(DBLAS1) \  
          $(ZBLAS1) $(CB1AUX) $(ZB1AUX) \  
          $(ALLBLAS) $(SBLAS2) $(CBLAS2) \  
          $(DBLAS2) $(ZBLAS2) $(SBLAS3) \  
          $(CBLAS3) $(DBLAS3) $(ZBLAS3)
```

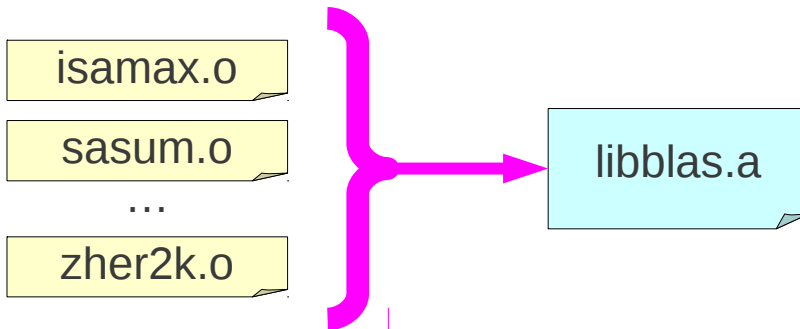
Makefile



If we look in the Makefile we have salvaged we see a collection of macros defining subsets of the Fortran files. We will define one more macro to join them all together. Note that we aren't trying to be clever with “* .f”, somehow asking for “all the Fortran files”. This is because we haven't manually compared the macro definitions with the files present and we run the risk of adding in extra files unexpectedly. Relying on “* .f” is a hostage to fortune if new files get added and I advise against it.

Tool for building static libraries

/usr/bin/ar Builds an object file “archive”



```
libblas.a: isamax.o ... zher2k.o
TAB $(AR) $(ARFLAGS) $@ $^
```

We have a collection of Fortran files. We know that make has defaults for building object files from them. All that we need now is the extra to combine them into a single library.

There is a tool to build the library. Usually it is called “ar” (“archiver”). It has flags to tell it what to do. We will rely on make's macros that correspond to these: AR and ARFLAGS.

libblas.a

4. Add the rule to the Makefile

```
libblas.a: $(OBJECTS)  
TAB      $(AR) $(ARFLAGS) $@ $^
```

Makefile

/tmp

building

LAPACK

BLAS

SRC

62

Rather than have the long list of object files in the rule's dependencies we will use the OBJECTS macro we defined.

libblas.a

5. Add the standard targets

```
PREFIX=${HOME}/sw
TARGET=libblas.a

all: $(TARGET)

install: all
    install -d $(PREFIX)/lib
    install $(TARGET) $(PREFIX)/lib

clean:
    $(RM) $(TARGET) $(OBJECTS)
```

Makefile

/tmp

building

LAPACK

BLAS

SRC

63

Finally we will add our standard targets. We're building a *good* Makefile. Note that our install target ensures that the directory exists before trying to install into it.

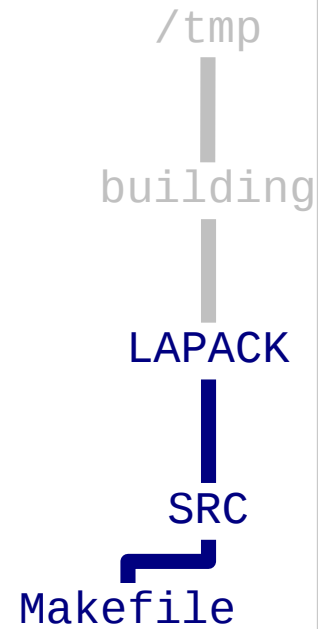
And that's it!

```
$ make FC=gfortran ← Identify Fortran compiler
gfortran -c -o isamax.o isamax.f
gfortran -c -o sasum.o sasum.f
...
ar: creating libblas.a
a - isamax.o
...
a - zher2k.o
```

And that's all we need. We just have to specify the new-fangled Fortran compiler and it "just works".

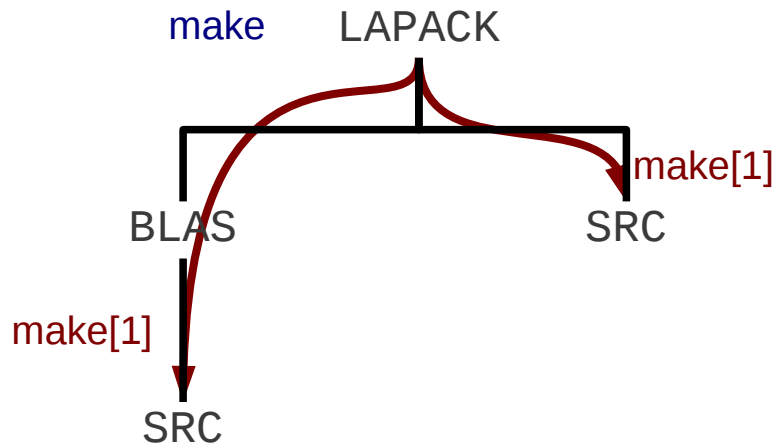
Same for liblapack.a

1. Keep the lists of file names
2. Ditch the rest
3. Glue all the object files together for convenience
4. Add the library build rule to the Makefile
5. Add the standard targets



We will now do exactly the same for liblapack.a in its Makefile.

Gluing it together



We can now make the two libraries by going to their respective directories and typing "make". What we want to do next is to build an over-arching Makefile that will let us do it with a single command.

Top-level Makefile

Pure recursion

```
all:
    cd BLAS/SRC && $(MAKE) all
    cd SRC && $(MAKE) all

clean:
    cd BLAS/SRC && $(MAKE) clean
    cd SRC && $(MAKE) clean

install: all
    cd BLAS/SRC && $(MAKE) install
    cd SRC && $(MAKE) install
```

UCS

67

Our top-level Makefile is purely recursive. All its targets are defined in terms of calling the same targets in the various subdirectories. The only thing to note is that we still have the “install” target depend on “all”.

Top-level Makefile

Common macros

```
export FC = gfortran

all :
    cd BLAS/SRC && $(MAKE) all
    cd SRC && $(MAKE) all

clean :
    cd BLAS/SRC && $(MAKE) clean
    cd SRC && $(MAKE) clean
```

UCS

68

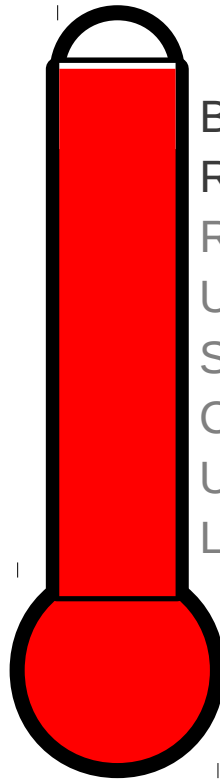
We can specify the Fortran compiler in this top-level Makefile if we want. Its value will be carried into the sub-makes if we add the “export” keyword..

And that's it!

```
$ cd /tmp/building/LAPACK  
$ make
```

And that's all it takes. We can simply move to the top-level directory and run “make”.

Progress



Build from scratch
Recursive make
Real world example
Using libraries
Simple make
Configured builds
Unpacking
Location

What we have done is to understand recursive make and then apply it to a complete rewrite of an existing build system.

Conclusion

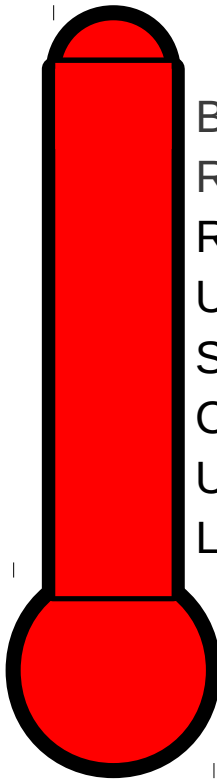
A tough course

You can build software:
 configure-style
 Makefile-style

Personal copies:
 no system
 privileges

Congratulations!

UCS



Build from scratch
Recursive make
Real world example
Using libraries
Simple make
Configured builds
Unpacking
Location

And that's all the course has. It has been a very full course; I believe it is the toughest course we offer.

You now know how to build software with a configure script or with a plain Makefile. You know how to maintain your own copies of software with no need for system privileges.

Congratulations.