### Silent Failure

Here's another makefile snippet that works some of the time:

```
clean:
→ @-rm *.o &> /dev/null
```

The `@` means that the command isn't echoed. The `-` means that any error returned is ignored and all output is redirected with `&>` to `/dev/null`, making it invisible. Because no `-f` is on the `rm` command, any failure (from say, permissions problems) will go totally unnoticed.

Usman's law strikes again.

### Recursive Clean

Many makefiles are recursive, and `make clean` must be recursive too, so you see the following pattern:

```
SUBDIRS = library executable

.PHONY: clean
clean:
→ for dir in $(SUBDIRS); do \
→ $(MAKE) -C $$dir clean;   \
→ done
```

The problem with this is that it means `make clean` has to work correctly in every directory in `SUBDIR`, leading to more opportunity for error.

Usman's law strikes again.

## Pitfalls and Benefits of GNU make Parallelization

Many build processes run for hours, so build managers commonly type `make` and go home for the night. GNU `make`'s solution to this problem is parallel execution: a simple command line option that causes GNU `make` to run jobs in parallel using the dependency information in the makefile to run them in the correct order.

In practice, however, GNU `make` parallel execution is severely limited by the fact that almost all makefiles are written with the assumption that their rules will run in series. Rarely do makefile authors *think in parallel* when writing their makefiles. That leads to hidden traps that either cause the build to fail with a fatal error or, worse, build "successfully" but result in incorrect binaries when GNU `make` is run in parallel mode.

This section looks at GNU `make`'s parallel pitfalls and how to work around them to get maximum parallelism.

### Using -j (or -jobs)

To start GNU make in parallel mode, you can specify either the -j or --jobs option on the command line. The argument to the option is the maximum number of processes that GNU make will run in parallel.

For example, typing make --jobs=4 allows GNU make to run up to four subprocesses in parallel, which would give a theoretical maximum speedup of 4×. However, the theoretical time is severely limited by restrictions in the makefile. To calculate the maximum actual speedup, you use Amdahl's law (which is covered in "Amdahl's Law and the Limits of Parallelization" on page 154).

One simple but annoying problem found in parallel GNU make is that because the jobs are no longer run serially (and the order depends on the timing of jobs), the output from GNU make will be sorted randomly depending on the actual order of job execution.

Fortunately, that problem has been addressed in GNU make 4.0 with the --output-sync option described in Chapter 1.

Consider the example in Listing 4-9:

```
.PHONY: all
all: t5 t4 t1
→ @echo Making $@

t1: t3 t2
→ touch $@

t2:
→ cp t3 $@

t3:
→ touch $@

t4:
→ touch $@

t5:
→ touch $@
```

*Listing 4-9: A simple makefile to illustrate parallel making*

It builds five targets: t1, t2, t3, t4, and t5. All are simply touched except for t2, which is copied from t3.

Running Listing 4-9 through standard GNU make without a parallel option gives the output:

```
$ make
touch t5
touch t4
touch t3
cp t3 t2
touch t1
Making all
```

The order of execution will be the same each time because GNU make will follow the prerequisites depth first and from left to right. Note that the left-to-right execution (in the all rule for example, t5 is built before t4, which is built before t1) is part of the POSIX make standard.

Now if make is run in parallel mode, it's clear that t5, t4, and t1 can be run at the same time because there are no dependencies between them. Similarly, t3 and t2 do not depend on each other, so they can be run in parallel.

The output of a parallel run of Listing 4-9 might be:

```
$ make --jobs=16
touch t4
touch t5
touch t3
cp t3 t2
touch t1
Making all
```

Or even:

```
$ make --jobs=16
touch t3
cp t3 t2
touch t4
touch t1
touch t5
Making all
```

This makes any process that examines log files to check for build problems (such as diffing log files) difficult. Unfortunately, there's no easy solution for this in GNU make without the --output-sync option, so you'll just have to live with it unless you upgrade to GNU make 4.0.

### Missing Dependencies

The example in Listing 4-9 has an additional problem. The author fell into the classic left-to-right trap when writing the makefile, so when it's run in parallel, it's possible for the following to happen:

```
$ make --jobs=16
touch t5
touch t4
cp t3 t2
cp: cannot stat `t3': No such file or directory
make: *** [t2] Error 1
```

The reason is that when run in parallel, the rule to build t2 can occur before the rule to build t3, and t2 needs t3 to have already been built. This didn't happen in the serial case because of the left-to-right assumption: the rule to build t1 is t1: t3 t2, which implies that t3 will be built before t2.

But no actual dependency exists in the makefile that states that `t3` must be built before `t2`. The fix is simple: just add `t2: t3` to the makefile.

This is a simple example of the real problem of missing or implicit (left-to-right) dependencies that plagues makefiles when run in parallel. If a makefile breaks when run in parallel, it's worth looking for missing dependencies straightaway because they are very common.

### The Hidden Temporary File Problem

Another way GNU `make` can break when running in parallel is if multiple rules use the same temporary file. Consider the example makefile in Listing 4-10:

```
TMP_FILE := /tmp/scratch_file

.PHONY: all
all: t

t: t1 t2
→ cat t1 t2 > $@

t1:
→ echo Output from $@ > $(TMP_FILE)
→ cat $(TMP_FILE) > $@

t2:
→ echo Output from $@ > $(TMP_FILE)
→ cat $(TMP_FILE) > $@
```

*Listing 4-10: A hidden temporary file that breaks parallel builds*

Run without a parallel option, GNU `make` produces the following output:

```
$ make
echo Output from t1 > /tmp/scratch_file
cat /tmp/scratch_file > t1
echo Output from t2 > /tmp/scratch_file
cat /tmp/scratch_file > t2
cat t1 t2 > t
```

and the `t` file contains:

```
Output from t1
Output from t2
```

But run in parallel, Listing 4-10 gives the following output:

```
$ make --jobs=2
echo Output from t1 > /tmp/scratch_file
echo Output from t2 > /tmp/scratch_file
cat /tmp/scratch_file > t1
cat /tmp/scratch_file > t2
cat t1 t2 > t
```

Now t contains:

```
Output from t2
Output from t2
```

This occurs because no dependency exists between t1 and t2 (because neither requires the output of the other), so they can run in parallel. In the output, you can see that they are running in parallel but that the output from the two rules is interleaved. Because the two echo statements ran first, t2 overwrote the output of t1, so the temporary file (shared by both rules) had the wrong value when it was finally cated to t1, resulting in the wrong value for t.

This example may seem contrived, but the same thing happens in real makefiles when run in parallel, resulting in either broken builds or the wrong binary being built. The yacc program for example, produces temporary files called y.tab.c and y.tab.h. If more than one yacc is run in the same directory at the same time, the wrong files could be used by the wrong process.

A simple solution for the makefile in Listing 4-10 is to change the definition of TMP_FILE to TMP_FILE = /tmp/scratch_file.$@, so its name will depend on the target being built. Now a parallel run would look like this:

```
$ make --jobs=2
echo Output from t1 > /tmp/scratch_file.t1
echo Output from t2 > /tmp/scratch_file.t2
cat /tmp/scratch_file.t1 > t1
cat /tmp/scratch_file.t2 > t2
cat t1 t2 > t
```

A related problem occurs when multiple jobs in the makefile write to a shared file. Even if they never read the file (for example, they might write to a log file), locking the file for write access can cause competing jobs to stall, reducing the overall performance of the parallel build.

Consider the example makefile in Listing 4-11 that uses the lockfile command to lock a file and simulate write locking. Although the file is locked, each job waits for a number of seconds:

```
LOCK_FILE := lock.me

.PHONY: all
all: t1 t2
→ @echo done.

t1:
→ @lockfile $(LOCK_FILE)
→ @sleep 10
→ @rm -f $(LOCK_FILE)
→ @echo Finished $@
```

```
t2:
→ @lockfile $(LOCK_FILE)
→ @sleep 20
→ @rm -f $(LOCK_FILE)
→ @echo Finished $@
```

*Listing 4-11: Locking on shared files can lock a parallel build and make it run serially.*

Running Listing 4-11 in a serial build takes about 30 seconds:

```
$ time make
Finished t1
Finished t2
done.
make 0.01s user 0.01s system 0% cpu 30.034 total
```

But it isn't any faster in parallel, even though `t1` and `t2` should be able to run in parallel:

```
$ time make -j4
Finished t1
Finished t2
done.
make -j4 0.01s user 0.02s system 0% cpu 36.812 total
```

It's actually slower because of the way `lockfile` detects lock availability. As you can imagine, write locking a file could cause similar delays in otherwise parallel-friendly makefiles.

Related to the file locking problem is a danger concerning archive (`ar`) files. If multiple `ar` processes were to run simultaneously on the same archive file, the archive could be corrupted. Locking around archive updates is necessary in a parallel build; otherwise, you'll need to prevent your dependencies from running multiple `ar` commands on the same file at the same time.

One way to prevent parallelism problems is to specify `.NOTPARALLEL` in a makefile. If this is seen, the entire `make` execution will be run in series and the `-j` or `--jobs` command line option will be ignored. `.NOTPARALLEL` is a very blunt tool because it affects an entire invocation of GNU `make`, but it could be handy in a recursive `make` situation with, for example, a third-party makefile that is not parallel safe.

### The Right Way to Do Recursive make

GNU `make` is smart enough to share parallelism across sub-makes if a makefile using `$(MAKE)` is careful about how it calls sub-makes. GNU `make` has a message passing mechanism that works across most platforms (Windows support was added in GNU `make` 4.0) and enables sub-makes to use all the available jobs specified through `-j` or `--jobs` by passing tokens across pipes between the `make` processes.