Building BSD with *meta* mode

To be presented at BSDCan 2011.

Abstract

Junos is a BSD derived OS.

For over 10 years, it has been built using bmake (the portable version of NetBSD's make), it has been through several evolutions taking advantage of techniques such as automated dependency collection both on a per directory, and tree-wide to facilitate parallel building. The author has added many features to NetBSD's make in that time.

This talk will discuss the latest evolution of the Junos build, taking advantage of *meta* mode in bmake - which has been contributed to NetBSD. The key makefiles discussed are available as part of the generic mk-files distribution that accompanies bmake.

A previous talk at BSDCan described John Birrell's `build` tool for FreeBSD.

The basic idea, of capturing *meta* info as targets are built remains, but the implementation has been completely overhauled and enhanced as part of the merging to bmake. The result retains the benefits without sacrificing any flexibility.

The original prototype for the FreeBSD build leveraged `DTrace` to track *interesting* (successful) system calls during the building of each target. Since `DTrace` required elevated privileges, we asked John to implement a simple kernel module (`filemon`), which `make` can use to get the same (and in some cases better) results.

The *meta* info thus collected allows for much more accurate update builds, as well as capturing tree based dependencies, to allow a clean tree build to be initiated from any point of interest.

Introduction

This talk is not about Junos (sorry). Rather, it is about building a large code base - like BSD (take your pick), for multiple architectures, in a reliable and efficient manner. Junos is simply an example, (any numbers I quote will be vague but erring on the conservative side).

First a couple of words about myself. Among other things, I have been building software and build systems for over 20 years. I have maintained the `bmake` distribution since the early '90s, and have been one of the maintainers of NetBSD's `make` for over 10 years.

Almost any mention of `bmake` below, should be assumed to hold for the `make` found in NetBSD. For the last several years the `bmake` version represents the date of the last sync from NetBSD.

These days `bmake` adds a rather thin film of additional portability over NetBSD's make, I test each distribution on the platforms I have access to which is currently NetBSD, FreeBSD, Linux, SunOS and Darwin. In the past I've also used it on AIX, HPUX, Irix,

OSF1, Ultrix and more. Based on the occasional bug report, bmake and the accompanying mk-files get used on some pretty obscure systems ;-)

BSD is a good example of a large, well organized code base, and to a large extent the BSD build is a good example too. The build magic is mostly in `bsd.*.mk`, and the top-level makefile. With the result that the bulk of the makefiles that people work with can be quite simple. That's not to say that the BSD build cannot be improved.

Out of curiosity, I had a colleague (who does such things) time a FreeBSD 8.x `make -j24 universe`, it took 89 minutes to produce just under 14Gb in `/usr/obj/`.

The same machine, doing a Junos production build takes about 260 minutes. That sounds bad until you consider it produces over 8 times the output - about 3 times the Gb/hour - without *meta* mode. There is still room for improvement.

Note: With regard to the the key makefiles discussed later, Juniper are contributing these to the community, via the mk-files distribution that accompanies bmake. I've tweaked some of them to be a bit more generic, and provided generic versions of the makefiles not suitable for distribution.

Teaser

Here's a quick example, building `/bin/sh` in FreeBSD current, in a clean tree:

```
$ mk destroy
(cd /c/sjg/work/FreeBSD/current/src && rm -rf /c/sjg/work/FreeBSD/current/obj/i386)
$ time mk -j12 -C bin/sh
[Creating objdir /c/sjg/work/FreeBSD/current/obj/i386/bin/sh...]
Checking /c/sjg/work/FreeBSD/current/src/stage for i386 ...
Checking /c/sjg/work/FreeBSD/current/src/stage for host ...
[Creating objdir /c/sjg/work/FreeBSD/current/obj/i386/stage...]
[Creating objdir /c/sjg/work/FreeBSD/current/obj/freebsd9-i386/stage...]
..
Building /c/sjg/work/FreeBSD/current/obj/i386/stage/stage0
Building /c/sjg/work/FreeBSD/current/obj/i386/stage/stage_root
Building /c/sjg/work/FreeBSD/current/obj/i386/stage/stage_include
Checking /c/sjg/work/FreeBSD/current/src/include for i386 ...
Checking /c/sjg/work/FreeBSD/current/src/usr.bin/rpcgen for host ...
[Creating objdir /c/sjg/work/FreeBSD/current/obj/freebsd9-i386/usr.bin/rpcgen...]
Building /c/sjg/work/FreeBSD/current/obj/freebsd9-i386/usr.bin/rpcgen/.dirdep
Building /c/sjg/work/FreeBSD/current/obj/freebsd9-i386/usr.bin/rpcgen/rpc_main.o
..
Building /c/sjg/work/FreeBSD/current/obj/freebsd9-i386/usr.bin/rpcgen/rpcgen
Building /c/sjg/work/FreeBSD/current/obj/freebsd9-i386/usr.bin/rpcgen/stage_files.prog
Checking /c/sjg/work/FreeBSD/current/src/usr.bin/rpcgen/Makefile.depend.host: .depend
Checking /c/sjg/work/FreeBSD/current/src/include/rpcsvc for i386 ...
[Creating objdir /c/sjg/work/FreeBSD/current/obj/i386/include...]
[Creating objdir /c/sjg/work/FreeBSD/current/obj/i386/include/rpcsvc...]
Building /c/sjg/work/FreeBSD/current/obj/i386/include/rpcsvc/key_prot.h
Building /c/sjg/work/FreeBSD/current/obj/i386/include/.dirdep
Building /c/sjg/work/FreeBSD/current/obj/i386/include/rpcsvc/key_prot.h
..
Building /c/sjg/work/FreeBSD/current/obj/i386/include/rpcsvc/crypt.h
Building /c/sjg/work/FreeBSD/current/obj/i386/include/rpcsvc/.dirdep
Building /c/sjg/work/FreeBSD/current/obj/i386/include/rpcsvc/stage_files.RPCHDRS
Building /c/sjg/work/FreeBSD/current/obj/i386/include/rpcsvc/stage_files.prog
Building /c/sjg/work/FreeBSD/current/obj/i386/include/rpcsvc/stage_files.INCS
Building /c/sjg/work/FreeBSD/current/obj/i386/include/stage_files.INCS
Updating .depend: key_prot.h.meta klm_prot.h.meta mount.h.meta
Checking /c/sjg/work/FreeBSD/current/src/include/rpcsvc/Makefile.depend.i386: .depend
Updating .depend: .dirdep.meta osreldate.h.meta stage_symlinks.INCS.meta
```

```
Checking /c/sjg/work/FreeBSD/current/src/include/Makefile.depend.i386: .depend
Checking /c/sjg/work/FreeBSD/current/src/lib/msun for i386 ...
Checking /c/sjg/work/FreeBSD/current/src/lib/libedit for i386 ...
Checking /c/sjg/work/FreeBSD/current/src/lib/ncurses/ncurses for i386 ...
[Creating objdir /c/sjg/work/FreeBSD/current/obj/i386/lib/msun...]
[Creating objdir /c/sjg/work/FreeBSD/current/obj/i386/lib/ncurses/ncurses...]
[Creating objdir /c/sjg/work/FreeBSD/current/obj/i386/lib/libedit...]
Building /c/sjg/work/FreeBSD/current/obj/i386/lib/libedit/help.h
Building /c/sjg/work/FreeBSD/current/obj/i386/lib/libedit/.dirdep
Building /c/sjg/work/FreeBSD/current/obj/i386/lib/libedit/common.h
..
Building /c/sjg/work/FreeBSD/current/obj/i386/lib/libedit/readline.o
Building /c/sjg/work/FreeBSD/current/obj/i386/lib/msun/b_exp.po
Building /c/sjg/work/FreeBSD/current/obj/i386/lib/libedit/readline.po
Building /c/sjg/work/FreeBSD/current/obj/i386/lib/msun/b_log.po
Building /c/sjg/work/FreeBSD/current/obj/i386/lib/libedit/editline.c
Building /c/sjg/work/FreeBSD/current/obj/i386/lib/ncurses/ncurses/.dirdep
Building /c/sjg/work/FreeBSD/current/obj/i386/lib/ncurses/ncurses/stage_files.DOCS
..
..
Building /c/sjg/work/FreeBSD/current/obj/i386/lib/libedit/libedit.a
Building /c/sjg/work/FreeBSD/current/obj/i386/lib/ncurses/ncurses/make_hash
..
Checking /c/sjg/work/FreeBSD/current/src/lib/msun/Makefile.depend.i386: .depend
Checking /c/sjg/work/FreeBSD/current/src/lib/libc for i386 ...
[Creating objdir /c/sjg/work/FreeBSD/current/obj/i386/lib/libc...]
Building /c/sjg/work/FreeBSD/current/obj/i386/lib/libc/fork.S
Building /c/sjg/work/FreeBSD/current/obj/i386/lib/libc/read.S
..
Building /c/sjg/work/FreeBSD/current/obj/i386/lib/libc/freebsd4_uname.o
Updating .depend: .dirdep.meta stage_files.DOCS.meta curses.head.meta
Checking /c/sjg/work/FreeBSD/current/src/lib/ncurses/ncurses/Makefile.depend.i386: .depend
Building /c/sjg/work/FreeBSD/current/obj/i386/lib/libc/sysarch.o
..
Building /c/sjg/work/FreeBSD/current/obj/i386/lib/libc/crypt_clnt.So
Building /c/sjg/work/FreeBSD/current/obj/i386/lib/libc/libc.a
building static c library
Building /c/sjg/work/FreeBSD/current/obj/i386/lib/libc/libc_p.a
building profiled c library
Building /c/sjg/work/FreeBSD/current/obj/i386/lib/libc/libc.so.7
Building /c/sjg/work/FreeBSD/current/obj/i386/lib/libc/libc_pic.a
building shared library libc.so.7
..
Building /c/sjg/work/FreeBSD/current/obj/i386/lib/libc/stage_libs
Updating .depend: fork.S.meta read.S.meta write.S.meta open.S.meta
Checking /c/sjg/work/FreeBSD/current/src/lib/libc/Makefile.depend.i386: .depend
Checking /c/sjg/work/FreeBSD/current/src/bin/sh for i386 ...
Building /c/sjg/work/FreeBSD/current/obj/i386/bin/sh/.dirdep
Building /c/sjg/work/FreeBSD/current/obj/i386/bin/sh/builtins.c
Building /c/sjg/work/FreeBSD/current/obj/i386/bin/sh/mkinit.o
Building /c/sjg/work/FreeBSD/current/obj/i386/bin/sh/mknodes.o
Building /c/sjg/work/FreeBSD/current/obj/i386/bin/sh/mksyntax.o
Building /c/sjg/work/FreeBSD/current/obj/i386/bin/sh/sh.1.gz
Building /c/sjg/work/FreeBSD/current/obj/i386/bin/sh/mksyntax
Building /c/sjg/work/FreeBSD/current/obj/i386/bin/sh/mkinit
Building /c/sjg/work/FreeBSD/current/obj/i386/bin/sh/syntax.c
Building /c/sjg/work/FreeBSD/current/obj/i386/bin/sh/init.c
..
Building /c/sjg/work/FreeBSD/current/obj/i386/bin/sh/parser.o
Building /c/sjg/work/FreeBSD/current/obj/i386/bin/sh/sh
Building /c/sjg/work/FreeBSD/current/obj/i386/bin/sh/stage_files.prog
Updating .depend: .dirdep.meta builtins.c.meta mkinit.o.meta
Checking /c/sjg/work/FreeBSD/current/src/bin/sh/Makefile.depend.i386: .depend
       67.03 real       196.12 user       170.12 sys
```

and what does a `Makefile.depend` file look like?

```
# Autogenerated - do NOT edit!

DEP_RELDIR := ${_PARSEDIR:S,${SRCTOP}/,,}

DEP_MACHINE := ${.PARSEFILE:E}

DIRDEPS = \
        include \
        lib/libc \
        lib/libedit \
        lib/ncurses/ncurses \


SRC_DIRDEPS = \
        bin/kill \
        bin/test \
        usr.bin/printf \


.include <dirdeps.mk>

.if ${DEP_RELDIR} == ${_DEP_RELDIR} && !exists(.depend)
# local dependencies - needed for -jN in clean tree
arith_yylex.o: syntax.h
arith_yylex.po: syntax.h
builtins.c: shell.h
builtins.o: builtins.c
...
.endif
```

All the real magic is in dirdeps.mk which we'll look at in more detail later, but a few things should be immediately obvious:

1. objdirs were created automatically. Note too that the objdirs are not within the src tree. Ensuring generated files and sources are kept in separate trees allows for targets like `make destroy` which are far more efficient than `make clean`.
2. no time was spent doing `make depend`
3. everything ran in parallel, but in the correct order. Each `Makefile.depend*` contains enough local dependency information to allow a successful parallel build in a clean tree.

The mk command

Everything you build seems to need its own make flavor and setup. The command `mk` is just a wrapper script which conditions the environment and makes it easy to build for different target machines, using the correct version of make:

```
$ cd bin/sh
$ mk -V .OBJDIR
/c/sjg/work/FreeBSD/current/obj/i386/bin/sh
$ mk-amd64 -V .OBJDIR
[Creating objdir /c/sjg/work/FreeBSD/current/obj/amd64/bin/sh...]
/c/sjg/work/FreeBSD/current/obj/amd64/bin/sh
$ mk-host -V .OBJDIR
[Creating objdir /c/sjg/work/FreeBSD/current/obj/freebsd9-i386/bin/sh...]
/c/sjg/work/FreeBSD/current/obj/freebsd9-i386/bin/sh
$ mk-host -V .MAKE
/homes/sjg/freebsd9-i386/bin/bmake-20110505
```

In its simplest form, the script simply searches upwards from the current directory until it finds the file `.sandbox-env`, which contains environment settings, and also marks the

location of the top of the tree (or *sandbox*, known as `$SB`). The `mk` script sources, `.sandboxrc` from various places and finally `$SB/.sandbox-env`, before running some flavor of `make`.

When I build NetBSD's `make` I typically use `mk-host` to cause the build to use the host's toolchain, headers and libs:

```
$ cd usr.bin/make
$ mk-host -V .OBJDIR
/var/obj/NetBSD/current/usr.bin/make
$ mk-host -V .MAKE
make
```

This setup is great if you're an Emacs user, since you typically only need to remember `M-x compile` (I bind that to `C-c m`) and have it run `mk`, and whether you are building FreeBSD, NetBSD, Junos or whatever, it *just works*.

Some definitions

Considering the venue, I expect most of you are familiar the the BSD build, and its makefile syntax. Below are some definitions, some of which should be familiar.

`.CURDIR`

> Is the value returned by getcwd(3) when make first starts. It is typically where the makefile is read from.

`.OBJDIR`

> Is the directory make is in when it starts building things. Make's predilection for finding an object dir causes confusion for those unfamiliar with it.
>
> The basic algorithm is (in Bourne shell):
>
> ```
> for __objdir in ${MAKEOBJDIRPREFIX}${.CURDIR} \
>       ${MAKEOBJDIR} \
>       ${.CURDIR}/obj.${MACHINE} \
>       ${.CURDIR}/obj \
>       ${.CURDIR}
> do
>         if [ -d ${__objdir} -a ${__objdir} != ${.CURDIR} ]; then
>                 break
>         fi
> done
> ```
>
> In `bmake`, makefiles can set `.OBJDIR`, this makes automated objdir creation possible (from `auto.obj.mk`):
>
> ```
> .if !defined(NOOBJ) && ${MKOBJDIRS:Uno} == auto
> # Use __objdir here so it is easier to tweak without impacting
> # the logic.
> __objdir?= ${MAKEOBJDIR}
> .if ${.OBJDIR} != ${__objdir}
> # We need to chdir
> .if !exists(${__objdir}) && \
>         (${.TARGETS} == "" || ${.TARGETS:Nclean*:N*clean:Ndestroy*} != "")
> # This will actually make it... Mkdirs is in sys.mk
> __objdir:=${__objdir:!umask ${OBJDIR_UMASK:U002}; \
>         ${ECHO_TRACE} "[Creating objdir ${__objdir}...]" >&2; \
>         ${Mkdirs}; Mkdirs ${__objdir}; echo ${__objdir}!}
> ```

```
.endif
# This causes make to use the specified directory as .OBJDIR
.OBJDIR: ${__objdir}
.endif
.endif
```

The `Mkdirs` function mentioned, works around the fact that in some implementations; `mkdir -p` do not handle race conditions. When attempting to build `bin/cat` and `bin/sh` in parallel and a clean tree, both makefiles will run a `mkdir -p` command that will want to create `${OBJTOP}/bin`, only one will succeed, the other throws an error.

`MAKESYSPATH`

A colon separated list of directories which `bmake` will search for `sys.mk`. The token `.../` means here or above, which means that `bmake -m .../share/mk` should *just work* in a typical BSD tree.

This is useful in conjunction with the *sandbox* idea, when building multiple branches on the one machine.

`MACHINE`

Identifies the specific machine or CPU that we are building for.

`MACHINE_ARCH`

The architecture that matches `${MACHINE}`, for example if `MACHINE` is `xlr` then `MACHINE_ARCH` would be `mips`.

`SB`

As mentioned above `${SB}` names the directory where `mk` found the file `.sandbox-env`. It is simply the top of a work space.

`SB_NAME`

We refer to the basename of `${SB}` often enough to give it its own variable.

`SB_SRC`

Usually `${SB}/src`, we typically set `SRCTOP` to this.

`SB_OBJROOT`

Usually `${SB}/obj/`, if `${SB}` is on NFS, `${SB_OBJROOT}` may be a symlink to local storage. We typically set `OBJTOP` to this with `${MACHINE}` appended.

`HOST_OBJTOP`

When building things for the host (the machine the build is running on), we use an object directory that uniquely identifies it. We append `${HOST_TARGET}` (eg. `freebsd7-i386`) to `${SB_OBJROOT}`.

`_CURDIR _OBJDIR`

If we are playing games with turning `.CURDIR` and `.OBJDIR` into relative paths, the original values are preserved in `_CURDIR` and `_OBJDIR`.

`RELDIR`

The relative path from `SRCTOP` to `.CURDIR`.

bmake additions

For those not familiar with the NetBSD build, `bmake` has a number of modifiers which may be new. There are lots, but I shall only mention the ones that appear sufficiently below to warrant introduction:

`:@`*temp*`@`*string*`@`

Is an in-line loop construct, which unlike `.for` is not evaluated when read, and does not limit expansion to the loop iterator. Each word of the variable is assigned to *temp* and then *string* is expanded. Insanely useful.

`:U`*value*

If a variable is undefined, use *value*. Note that's undefined not empty. One can however use:

`${"${`*VARIABLE*`}":?`*if-set*`:`*if-empty-or-undefined*`}`

`:ts`*c*

Use *c* as the separator between words.

`:tA`

Apply realpath(3) to each word.

I've added a lot of features to NetBSD's `make` to enable improvements in the Junos build (of course they had to be generically useful too ;-). I have been repeatedly surprised, at how quickly many of these improvements have been adopted by the NetBSD build.

I should point out that a number of NetBSD developers actively work on improving `make`, and Juniper has benefited greatly from their efforts too.

Note: the `:@` and `:U` mentioned above and a number of other useful modifiers were adapted from the `pmake` found in OSF. The OSF development environment (ODE) did some very cool things with them.

The following variables are also relevant to the discussion:

`.MAKE.LEVEL`

Defines the recursion level. The first instance of `bmake` will have `.MAKE.LEVEL == 0`, sub-makes will have an incremented value.

`.MAKE.MODE`

The means of putting `bmake` into *meta* mode. More detail later.

`.MAKE.MAKEFILE_PREFERENCE`

Default is `makefile Makefile`, it is the list, in order of preference, of makefiles that `bmake` will look for. Note that *makefile* describes any file read

by `make` which tells it what to do.

`.MAKE.DEPENDFILE`

The file that `bmake` automatically reads after the makefiles. Default is `.depend`, but for *meta* mode we use `${.CURDIR}/Makefile.depend.${MACHINE}`

`.PARSEFILE`

The basename of the makefile currently being read. That is the file the line containing the reference to `${.PARSEFILE}`. Any file read by `make` which tells it what to do is a makefile.

`.PARSEDIR`

The directory where the current makefile was found. Applying `:tA` to this can be very handy, as we shall see.

`MAKE_VERSION`

These days this is just the `YYYYmmdd` that represents the `bmake` version. A number of makefiles in mk-files need to check this, to know whether certain features will work.

For some crazy reason, when I first put this in NetBSD, the value was a tuple that included when it was built. I think the idea was to be able to distinguish `bmake` from the native `make`. Bad idea. Apart from being harder to utilize, that caused problems for people trying to re-produce builds. So now NetBSD's `make` only has it if the makefile defines it. In my own trees, I set it to the `mtime` of `CVS/Entries`.

The net result is that my generic `sys/NetBSD.mk` has to guess values for `MAKE_VERSION` if it is not defined.

History

I promised some history...

In the past, the typical BSD build process could be expressed as something like:

```
make obj
make includes
make depend
make libs
make all
make install
```

each of which would recurse their way down the tree processing `SUBDIR` in the order listed etc. The whole process might be wrapped up in a target like `make world` or `make release`.

The tree walks were originally beneficial - keeping the memory footprint of `make` reasonable.

Some complained about the cost of the multiple tree walks, so a couple of the steps were collapsed into:

```
make dependall
```

that is, as you walk the tree do `make depend all` in each directory. But it is still sub-optimal, and making it run well in parallel is a challenge.

With the increased popularity of cross-building (especially if the target has an anaemic CPU), the builds have grown additional phases like `make hosttools`.

When I started at Juniper at the beginning of 2000, FreeBSD 2.x was in use and we were working on Junos 4.x. I wrote a document which describes the evolution of the Junos build in terms of the major versions that changes were introduced with. The following sections provide a brief summary.

It is worth noting that Juniper routers, generally use separate CPUs for the control and data planes. The data plane is where all the super cool, proprietary ASICs live. The control plane is where the BSD bit of Junos runs, much of our discussion is only concerned with that. Until recently the data plane used its own build environment (using `gmake`) and none of the cool features below applied to it ;-)

Junos 4.x (2000)

The 4.x Junos build was sub-optimal, and caused a number of problems for developers.

Up to this time, Junos was built as a small number packages added to a stock FreeBSD install. All the bits were handled independently. The data plane build had been developed on SunOS, so bore no relationship to the BSD build.

One of my initial projects was to design an improved build which would integrate everything, and facilitate subset checkouts and builds, with zero impact to the current official builds.

While that zero impact requirement caused some contortions, the result was a big improvment.

This build introduced:

1. The concept of a *sandbox* and the commands;

    - `mk` to launch `make` after conditioning the environment
    - `mksb` to prepare a *sandbox* and checkout the sources.
    - `workon` to run a shell within the *sandbox*

    These tools were all inspired by the OSF Development Environment (ODE), but the implementations were simple scripts which allowed a lot more flexibility.

    Using the *sandbox* model, developers could switch from working on one branch to another and use the correct tools and configuration without thought.

2. A tree layout which was basically the BSD tree, with a new top-level directory containing the Juniper bits.

3. Use of `bmake`; I added a collection of variable modifiers from ODE to NetBSD `make`, to enable the design.

Junos 5.0 (2001)

This was a major set of changes, which coincided with a migration to FreeBSD 4.2 and from `a.out` to `ELF`. A number of interesting things were introduced:

1. To avoid bugs from people forgetting to `make includes` at appropriate times, the build was changed such that all headers were used from where they live.

   That is, if you were using libc, you might have a `-I${SRC_libc}/include` etc in `CFLAGS`.

   For Juniper libraries we introduced the notion that `${SRC_libfoo}/h` was where all the public headers would be - so we could avoid accidentally picking up private ones.

   `dpadd.mk` took care of working out all the above, based on `bsd.libnames.mk` entries (see below).

2. Similarly, libraries were used from where they were built. No more `make install` during the build.

3. We introduced the notion of packaging our s/w as compressed ISO images.

4. The idea of staging files to a `$DESTDIR` was dropped. The ISO images were built from a manifest which identified the location of an object file in the tree, and the desired location within the ISO image.

5. The top-level makefiles changed quite radically, introducing prefixes such that the same dependency information could be leveraged for checking out subsets of the code base, as for building it.

   These makefiles were not my idea btw. While not as elegant perhaps, they were easier to explain than the key makefile that drove my 4.x build, and they achieved a largely similar result. The basic idea is still in use today.

6. We used `autodep.mk` which leveraged `gcc -M*` to eliminate the need for a separate `make depend`.

The above all worked very well. Builds were faster due to avoiding lots of tree walks, and copying of binaries, people could easily build the entire system in about an hour, and life was good.

Manually maintaining the dependency information was error prone though, and eventually we hit problems with command line limits in FreeBSD 4.x. The NFS bugs that we had encountered in FreeBSD 2.x, were gone and we had again allowed folk to use NFS for building. The longer pathnames that resulted (from `amd`) blew the command line limits.

The solution was to turn `.CURDIR` and `.OBJDIR` into relative paths - saving the original values in `_CURDIR` and `_OBJDIR`. This meant that the command line lengths were independent of tree location, and the debug symbols that ended up in object files were (in some cases) more useful.

Junos 7.0 (2004)

A new set of top-level makefiles were introduced. The top-level Makefile included a

bunch of component makefiles that did specific things. The goal was to improve maintainability.

Object dir creation was automated using `auto.obj.mk`.

Support for backing sandboxes was introduced. This allowed a subset checkout, to leverage a pre-built tree to provide missing headers, libraries and binaries. This same functionality enabled the Junos SDK a couple of years later.

A means of automatically deriving the bulk of the tree dependencies from the the manifest files - which controlled what went into the ISO images, was added.

It is worth stressing that the Junos build had long since abandoned the notion of using `SUBDIR` and tree walks. Rather, the top-level makefiles generally visited the leaves of the tree directly, based on the dependency information known to the top-level.

Thus automating the collection of that dependency information allowed for a big improvement in build reliability. We stored this information in the SCM, in a single directory, and were able to leverage it for fine-grained subset checkouts. But this eventually became a hot-spot for conflicts as the number of developers increased.

At this time, there were three "architectures" that Junos was built for. The control plane software had to be built for i386 and mips, and the data-plane was its own little world. Allowing each of these to build in parallel allowed us to pretty well consume all the CPU available at the time. Which was handy, because much of the FreeBSD 4.x tree could not handle building in parallel - at least not without an explicit `make depend`.

Junos 9.3 (2008)

By this time we had upgraded the FreeBSD code base to 6.x

We removed the captured dependency information from the SCM. This avoided the conflict hot-spot, and allowed the data to be collected on a per `$MACHINE` basis (a number of new architectures had been added). The downside was that doing sub-set checkouts became a bit more complicated.

Apart from new architectures that we were building the control plane software for, a number of low-end hybrid systems had evolved which introduced dependencies between the the data and control plane builds, such that the separate build for the data plane was becoming an issue that would need to be fixed.

Today

We are in the second stage of migrating the build to *meta* mode. The `gmake` build for the data plane, has been replaced with `bmake` and *meta* mode and already showing benefits.

The BSD portion of the tree, can also run in *meta* mode, and the current build is useful for bootstrapping the `Makefile.depend` files, but final conversion requires re-working some key makefiles which currently break the one build product per directory dictum, and hence interfere with capturing complete and stable dependencies.

Once conversion is complete, we expect to achieve a much higher degree of parallelism than is currently possible. Any build performance improvements though are a bonus. The primary benefit is improved reliability of update builds and reduction of build complexity.

Desirable build features

The following sections discuss some features that have proven useful over a number of years. All of these are retained in our next evolution.

Separating sources and objects

Too much flexibility, can add greatly to the cost of supporting a build. While some people may *like* their objects and src in the same directory, we don't give them that option (any more ;-)

Keeping in mind we don't want the default objdir of `${.CURDIR}/obj/`, the simplest way to keep objects separate from the source, is by using `MAKEOBJDIRPREFIX`:

```
$ export MAKEOBJDIRPREFIX=/var/obj/$USER
$ mkdir -p $MAKEOBJDIRPREFIX
$ pwd
/homes/sjg/work/sb/src
$ make obj
$ make -V .CURDIR
/.amd/server/homes/sjg/work/sb/src/bin/cat
$ make -V .OBJDIR
/var/obj/sjg/.amd/server/homes/sjg/work/sb/src/bin/cat
```

While `MAKEOBJDIRPREFIX` is handy, the paths that result can be hideous especially if `.CURDIR` is on an automounted NFS.

Since `bmake` allows applying modifiers to `MAKEOBJDIR` we use that to achieve much neater results:

```
$ export MAKEOBJDIR='${.CURDIR:S,${SRCTOP},${OBJTOP},}'
```

with suitable definitions for `SRCTOP` and `OBJTOP`, for example:

```
$ export SRCTOP=$SB/src
$ export OBJTOP='${OBJROOT}${MACHINE}'
$ export OBJROOT=/var/obj/$USER/$SB_NAME
$ make -V .OBJDIR
[Creating objdir /var/obj/sjg/sb/i386/bin/cat...]
/var/obj/sjg/sb/i386/bin/cat
$ make -V .CURDIR
/.amd/server/homes/sjg/work/sb/src/bin/cat
```

One downside of both `MAKEOBJDIRPREFIX` and `MAKEOBJDIR` is that they need to be set in the environment. While `bmake` has the ability to export and even completely wipe the environment, getting the right value for `MAKEOBJDIR` exported by `bmake` is a challenge. If you use a wrapper like `mk` anyway, this isn't an issue.

Some people still like to use `mk objlink` to get `./obj` as a symlink to `.OBJDIR` for convenience. These symlinks are ignored by the build.

Once you have a well defined location for `SRCTOP` and `OBJTOP`, many things become simpler.

For example, one can simply assert:

```
CRYPTOBJDIR= ${OBJTOP}/secure/lib/libcrypt
```

rather than (from FreeBSD's `secure/Makefile.inc`):

```
.if exists(${.CURDIR}/../../lib/libcrypt/obj)
CRYPTOBJDIR=    ${.CURDIR}/../../lib/libcrypt/obj
.else
CRYPTOBJDIR=    ${.CURDIR}/../../lib/libcrypt
.endif
```

which is just wrong. I've seen more elaborate examples which invariably guess the wrong answer, but it was long ago. The above is still around.

As noted in the introduction, if your objdirs are not within the src tree, then cleaning becomes a simple matter of:

```
destroy:
.if ${.OBJDIR} != ${.CURDIR}
        (cd ${.CURDIR} && rm -rf ${.OBJDIR})
.endif
```

Actually, it is quicker to run an `rm -rf` per `${MACHINE}` in parallel but you get the idea.

Keeping the src tree clean, also helps when trying to use tools to do pre-commit checks on a tree. For example, that all the srcs have been added to the SCM.

That is, look at the diff to be committed, build all the directories referenced, then visit them and their pre-requisits and check that none of the src files are unknown to the SCM.

Automated dependency collection

For over a decade, the Junos build has used `autodep.mk` to avoid the need for a separate `make depend`. This has generally worked very well.

The basic idea is to have `gcc -M*` write dependency info to `${.PREFIX}.d` and after a successful compilation, munge those together to form `.depend`.

Originally, `${.PREFIX}.d` was selected so that it could play nice with `.SUFFIX` rules, but presents a problem in that compiling `foo.o` and `foo.po` will both produce the same `foo.d` file. Using `.ORDER` to prevent `foo.o` and `foo.po` from being compiled at the same time avoids problems.

A different model is used by `auto.dep.mk`, using `.d.${TARGET}` as in `.d.foo.o` and `.d.foo.po` avoids any contention, at the cost of duplicate data in `.depend` and not working with `.SUFFIX` rules (which is probably an advantage).

In *meta* mode, which I'll get to in a while, we use `.depend` completely differently.

Directory based dependencies

Not only are tree walks (using `SUBDIR`) expensive, especially when building on NFS, it may be impossible to adequately order the build steps without resorting to phases like `make includes` and `make libraries`.

In a BSD source tree, most Makefiles use `DPADD` to indicate their pre-requisites. As I mentioned earlier, the Junos build uses the libraries from where they are built. Further, leveraging the relatively simple relationship between object and src directories discussed

earlier, it becomes a simple matter to derive the the src directory for a library, from its object dir.

The magic is all handled by a makefile called `dpadd.mk`. It handles the following macros for each library (eg `libfoo.a`) given that `LIBFOO` is set to the absolute path of the library:

`OBJ_libfoo`

> Trivial - `${LIBFOO:H}` but completes the set and simplifies the descriptions below.

`SRC_libfoo`

> The src directory for `libfoo.a` defaults to `${OBJ_libfoo:S,${OBJTOP},${SRCTOP},}`

`INCLUDES_libfoo`

> The `-I` values needed for the public api. If the directory `${SRC_libfoo}/h` exists, then this defaults to `-I${SRC_libfoo}/h`, otherwise all bets are off and the default is `-I${SRC_libfoo} -I${OBJ_libfoo}`

The above are computed for any `${LIB*}` which appears in one of:

`SRC_LIBS`

> Names libs which we just need includes from, so:
>
> ```
> SRC_LIBS += ${LIBFOO}
> ```
>
> results in:
>
> ```
> CFLAGS += ${INCLUDES_libfoo}
> ```

`DPADD`

> As for `SRC_LIBS` and of course, we require that the specified libs exist.

`DPLIBS`

> Directory based dependencies rely on accurate information.
>
> Having observed that many people add `-lgoo` to `LDADD` without adding `${LIBGOO}` to `DPADD`, we use `DPLIBS` to replace both. That is:
>
> ```
> DPLIBS += ${LIBGOO}
> ```
>
> is exactly equivalent to:
>
> ```
> DPADD += ${LIBGOO}
> LDADD += ${LIBGOO:T:R:S,lib,-l,}
> ```

So given something like this in `bsd.libnames.mk`:

```
LIBC ?= ${OBJTOP}/bsd/lib/libc/libc.a
LIBCRYPT ?= ${OBJTOP}/bsd/lib/libcrypt/libcrypt.a
LIBPAM ?=  ${OBJTOP}/bsd/lib/libpam/libpam/libpam.a
...
INCLUDES_libpam += -I${CONTRIB}/openpam/include
```

and a makefile such as in `bsd/usr.bin/login`, which has:

```
DPLIBS += ${LIBCRYPT} ${LIBMD} ${LIBPAM} ...
```

The `dpadd.mk` in mk-files says:

```
$ mk dpadd
bsd/usr.bin/login: bsd/lib/libcrypt/libcrypt.a \
                   bsd/lib/libmd/libmd.a \
                   bsd/lib/libpam/libpam/libpam.so \
                   ... \
                   bsd/lib/libc/libc.a
```

which is interesting, but we can do more with the data. The version in Junos provides a lot more information:

```
$ mk dpadd
# start-of-clues

DPDEPS_bsd/lib/libc += bsd/usr.bin/login
DPDEPS_bsd/lib/libcrypt += bsd/usr.bin/login
DPDEPS_bsd/lib/libmd += bsd/usr.bin/login
DPDEPS_bsd/lib/libpam/libpam += bsd/usr.bin/login
...

# end-of-clues

DPADD_bsd/usr.bin/login = \
        bsd/lib/libc \
        bsd/lib/libcrypt \
        bsd/lib/libmd \
        bsd/lib/libpam/libpam \
        ...


${P}bsd/usr.bin/login: ${DPADD_bsd/usr.bin/login:S,^,${P},}

DPCVS_bsd/usr.bin/login = \
        bsd/sys \
        bsd/contrib/openpam \

cvs-bsd/usr.bin/login: ${DPCVS_bsd/usr.bin/login:S,^,cvs-,}
```

which can be captured, and used by the top-level makefiles to know that to build `bsd/usr.bin/login` one must first visit the list of libraries. It also lets us know what needs to be checked out of the SCM to be able to build `login` (in which case `P=cvs-`) and also build a picture of the clients of each library.

A simple script can take the above information, and visit each of the pre-requisite dirs, and run `bmake dpadd` there too and thus gather the hierarchy of dependencies needed.

It then becomes possible for the top-level makefile to visit the leaves of the tree directly in optimal order to build a given target.

As mentioned earlier, we originally captured this information in the SCM, but eventually removed it (due to too many conflicts), and generated it dynamically on a per `${MACHINE}` basis.

In *meta* mode, we take this idea a step further.

Building in parallel

There's no such thing as building too fast. Doing a number of things in parallel helps soak up every available CPU second that would otherwise be spent waiting for I/O.

Running make in *jobs* mode does that. The optimal max jobs number varies due to factors like the number of CPUs available, how many disks are being striped for the build space, and what else is happening on the box.

Going fast though, doesn't matter if you get incorrect results.

Bmake works in two modes, *jobs* mode and *compat* mode. In compat mode, targets are evaluated depth first in the order listed. In jobs mode, targets are evaluated breadth first and in parallel.

For example:

```
LIB = fool

SRCS = parser.c file1.c file2.c file3.c

parser.c: parser.y
        ${YACC} -d -o ${.TARGET} ${.IMPSRC}
        mv t.tab.y ${.TARGET:T:R}.h

.include <bsd.lib.mk>
```

In compat mode, everything *just works*, because the SRCS are built sequentially - in the order listed. So everything associated with producing `parser.o` will happen before `file1.o` is attempted.

In jobs mode (e.g. `mk -j8`), things can rapidly fall apart. With just the information listed, each of the SRCS will be compiled in parallel. If any of `file*.c` includes `parser.h`, then success will be random depending on whether the `mv` step of the script above completes before any of the compilations of `file*.c` need it. This is called a race condition and more available CPUs typically exacerbates the situation.

To avoid that race condition, more explicit dependencies are needed. This however, is often done incorrectly:

```
# wrong: this can cause YACC to be run twice - at the same time!
parser.c parser.h: parser.y
        ${YACC} -d -o ${.TARGET:T:R}.c ${.IMPSRC}
        mv t.tab.y ${.TARGET:T:R}.h

file1.o:        parser.h
```

The above code introduces a different race condition - both `parser.c` and `parser.h` can trigger running the same script (at about the same time) which usually ends in tears.

The next attempt might be:

```
# wrong: likelihood of circular dependencies
parser.h: parser.c
parser.c: parser.y
        ${YACC} -d -o ${.TARGET:T:R}.c ${.IMPSRC}
        mv t.tab.y ${.TARGET:T:R}.h

file1.o:        parser.h
```

but that can result in cyclic dependencies if `parser.c` includes `parser.h` and this ends

up reflected in `.depend`.

The best bet would be:

```
parser.h: parser.y
        ${YACC} -d -o ${.TARGET:T:R}.c ${.IMPSRC}
        mv t.tab.y ${.TARGET}

parser.c:       parser.h

file1.o:        parser.h
```

This avoids the race conditions as well as the chance of cyclic dependencies.

In the current Junos build, leaf makefiles do not run in *jobs* mode by default. So the parallelism is mainly at the top-level. For small leaf directories this does not matter.

For big libs like `libc` it does matter. There is therefore a `USE_JOBS` knob that can be set in a makefile, to indicate that it can build in *jobs* mode, and this gets picked up by the top-level (via the clues printed by `mk dpadd`), so all is not lost. The speedup for big libs like `libc` is significant.

In *meta* mode, we capture in the `Makefile.depend*` sufficient information to ensure a successful parallel build in a clean tree. Thus the default will be to allow leaf makefiles to build in *meta jobs* mode.

Even better, you can do a simple `mk` in *compat* mode while bootstrapping a new makefile, to capture the local dependencies, and it will then generally *just work* in *jobs* mode.

Captive toolchains

Some things get used many times, but changed rarely. This is especially true for most of the build tools, like the compilers and make. We use captive versions where possible.

Unlike Open Source teams, we have a need to be able to reproduce a build of Junos up to 10 years after it was shipped, so managing captive toolchains is taken seriously.

Our compiler team qualify a new compiler and post the result to a tools collection in a versioned directory. Once posted it is never touched again.

The same goes for tools like `bmake`. I will import a new version, and after a suitable qualification period, someone in the tools team will build and post it.

A `toolchains.mk` can then have things like:

```
BMAKE_VERSION ?= 20101215
BMAKE = ${TOOLS_PREFIX}/bmake/${BMAKE_VERSION}/bin/bmake
```

and developers generally don't waste time building compilers, make and similar tools.

NetBSD's build provides neat hooks for building toolchains and re-using them, even using externally maintained cross-toolchains by setting `EXTERNAL_TOOLCHAIN`.

Some issues

Too many -I's and -L's

As mentioned above, for the last decade we have included headers from where they are edited, and linked libraries from where they are built. While this has been very successful in meeting the original goals, our code base has grown a little, and in some cases the number of `-I`'s being used seems excessive.

Each `-I` results in a directory stat(2) which can be expensive on NFS. More importantly from an architecture point of view, all the `-I`'s and `-L`'s make it too easy to introduce name collisions. When all the headers and libs are being installed into common directories name collisions become immediately obvious.

When you never install the libs or headers, you may not notice name collisions until much later. While this has not been a big issue, the potential remains.

The primary motive for the current model was the difficulty of ensuring that updated headers and libs were installed when they should be (at least when developers are not doing top-level builds), and avoiding the confusion that can result.

With *meta* mode, though, we can eliminate that problem, and thus allow revisiting the current model. This will happen in a phased manner.

Too much complexity

The Junos code base has probably tripled in size, since the nice clean top-level makefiles were introduced in 7.0. While most of the leaf Makefiles remain nice and simple, the complexity of the top-level makefiles has gone up dramatically.

For example the following (simplified version):

```
# Run a sub-make with MACHINE and MACHINE_ARCH set appropriately.
_BUILD_ARCH_USE:        .USE .PHONY .MAKE
        @echo "[Building __${.TARGET} for ${@:E} ...]"
        @(cd ${.CURDIR} && MACHINE=${.TARGET:E} \
        MACHINE_ARCH=${MACHINE_ARCH.${.TARGET:E}} \
        ${.MAKE} __$@)

.for m in ${ALL_MACHINE_LIST}
.if ${MACHINE} == $m
build_arch.$m: __build_arch.$m
# make sure this exists
__build_arch.$m:
.else
build_arch.$m:  _BUILD_ARCH_USE
.endif
.endfor
```

can be used thus:

```
all: build_arch.i386 build_arch.mips

__build_arch.i386:      lots-of-stuff
__build_arch.mips:      lots-of-stuff
```

to build `lots-of-stuff` for both `i386` and `mips`.

Of course, the top-level makefile is used to build many individual package targets too - since developers spend a lot of their time building only for the platform they are currently

working on.

It is quite common for a platform target to need things built for multiple machine types (seems every card needs a unique CPU ;-) It is all too easy to introduce something like:

```
some-thing:   build_arch.abc
__build_arch.abc:       one-thing and-another
and-another:            build_arch.xyz
__build_arch.xyz:       lots-more-stuff
```

which works fine while building just `and-another` or `some-thing`, but if we have elsewhere:

```
every-thing: ${ALL_MACHINE_LIST:%=build_arch.%}
```

then while building `every-thing`, which depends on `build_arch.abc` and `build_arch.xyz`, the dependency for `and-another` above will result in two separate invocations of `bmake __build_arch.xyz` in parallel, which is bound to end in tears.

By the time we complete our migration to *meta* mode, all that complexity will be gone - replaced by dirdeps.mk

Manual maintenance is unreliable

Apparently not all C programmers are build geeks.

My basic rules for writing leaf makefiles are:

1. Do not put anything in your makefile that you don't need
2. Do not put anything in your makefile that you cannot explain the need for. Ie. if you cannot explain it, you don't need it, remove it.
3. Do not cut/paste anything from your friend's makefile (see #1).

Note: #2 does not mean that you should remove everything from an existing makefile that you don't understand the first time you look at it.

Sadly, few people read as far as #3. In fact it seems that some will seek out the most complex possible makefile to cut/paste from.

The result is that makefiles (like C code), can accrete dependencies which in many cases are unnecessary.

The less humans need to maintain, the better.

A top-level build needed

The Junos build leverages a lot of hosttools (those compiled for the build host), to generate code, and other useful things.

As a result, the current build requires some form of top-level build be performed in a clean tree, before developers get into their edit, compile, test cycle. This is considered irksome.

As our earlier example showed, this is not necessary with *meta* mode.

Insufficient parallelism

In the good old days, just being able to run the top-level makefiles in paralllel for two or three different architectures pretty well consumed a build server.

While the build machines have gotten much faster, the build itslef has become more complex. For example, if for packaging requirements machine-A, depends on products from machine-B, then none of machine-A can start until all of machine-B is complete. A couple of such dependencies can put a huge dint in achievable parallelism. The 15min load average is a good clue as to what is being achieved.

Introducing *Meta* Mode

Ok, so just what is *meta* mode? It is quite simple really.

When `bmake` is run in *meta* mode, it creates a `.meta` file for each target.

A `.meta` file simply collects information about the target that `bmake` is building. This includes the expanded commands run, the environment (optional), any command output, and finally a capture of *interesting* system calls performed by the commands. Such information can have many uses, not least of which is automating capture of tree-wide dependencies.

The basic idea of `.meta` files originated in John Birrell's `build` which he started as a project to improve the FreeBSD build before joining Juniper. John also wrote the `filemon` module for FreeBSD. The `build` prototype hard-coded *all* build behavior, and required all new makefiles which made transition (and hence adoption) challenging.

The *meta* mode in `bmake` just provides the basic functionality leveraging the same kernel module (also available in NetBSD), and leaves the usage/policy up to the makefiles, most of which do not need to change at all.

This added flexibility allows for a reasonably smooth transition. The current Junos build has been able to leverage *meta* mode for over a year, and while the data plane build (which used to use `gmake`), has been migrated to build with `bmake` in *meta* mode, further work is needed to complete the transition.

A number of makefiles will be mentioned below. Some have a `meta.` prefix in their name - to distinguish them from other variants, and also to aid in their use along side the other variants since fully leveraging meta mode can require changes to the tree and thus migration may take a while.

- Makefile.depend since Junos is cross-built for multiple target machines, we set `.MAKE.DEPENDFILE=Makefile.depend.${MACHINE}` but we use the unqualified name in most discussions. We can also use the unqualified name for manually edited files which are not machine dependent. dirdeps.mk does the right thing.
- dirdeps.mk included by `Makefile.depend` is where the real magic happens. This may not be one of those makefiles you want to read without plenty of caffeine handy. It is however pretty well commented, and stable (no need to touch it).
- meta.autodep.mk does the same thing as the `autodep.mk` that `bmake` has had for many years, but leverages *meta* mode.
- gendirdeps.mk included by `meta.autodep.mk` when `Makefile.depend` needs to be updated.

- meta.subdir.mk the traditional `bsd.subdir.mk` does not fit into the build style enabled by *meta* mode, but we still want a means of launching a build in a non-leaf directory.

I should point out that most of the above rely heavily on the functionality of `bmake`.

Rationale

Why a new mode for make? To aid the automated capture of dependency information, and thus help optimize build performance, while minimizing the changes needed. That's not to suggest that it is a good idea to support building the same tree multiple ways beyond a transition period.

Optimizing build performance means doing as little as possible, and on modern CPUs doing as much of it in parallel as possible. However, being quick is useless if the results are incorrect. Meta mode helps on all these fronts.

avoid make depend

As noted above, avoiding `make depend` improves build speed.

For many years `autodep.mk` has leveraged the `gcc -M*` flags to gather dependency information as a side effect of compilation. This works fine for most makefiles, but not all when trying to build in parallel. Makefiles which omit important dependencies require `make depend` or equivalent before they can be successfully built in parallel.

By capturing *local* dependencies into `Makefile.depend` (see the example in Makefile.depend), we can successfully build a clean tree in parallel, without having to fix all the makefiles.

Also by leveraging filemon we can capture accurate dependencies for all targets, not just those built by `gcc`.

avoid unnecessary dependencies

There are also advantages to be had when re-building a tree. The fact that in *meta* mode, `bmake` can compare the actual expanded command lines when deciding if a target is out-of-date, means that one can avoid doing things like:

```
# if any of the makefiles have changed we need to regenerate
# this - "just in case"
generated.h:    ${.MAKE.MAKEFILES:N.depend}
${OBJS}:        generated.h
```

which can often lead to lots of unnecessary re-compilation.

detect unnecessary dependencies

In *meta* mode, we can use `DPADD` for bootstrapping `DIRDEPS`, but otherwise it is not necessary.

Subsequent builds follow the actual dependencies recorded in `Makefile.depend`, so when a leaf makefile has a spurious entry in `DPADD` it can cause the build to fail with:

```
bmake-20110306: don't know how to make libbogus.a
```

This is useful during the transition phase, because until you fully cut-over to *meta* mode, cleaning out those spurious entries reduces dependencies and improves the build speed. Later, you can simply ignore `DPADD`.

tree walks don't always cut it

Ideally we want to build the tree in a single pass.

Apart from being inefficient, using `bsd.subdir.mk` to walk a tree, can be a challenge if various leaf directories have dependencies on each other.

As noted, the Junos build has for many years made use of tree-wide dependencies to visit the leaf nodes of the tree directly - in the order required, with a high degree of parallelism.

The model described here allows the same functionality, but with less overhead and in a more generic manner. Just as `autodep.mk` collects the dependencies of a directory as a side effect of building, in *meta* mode we also collect the tree dependencies as a side effect of building, and running `meta2deps`.

Of course one still needs to be able to launch a build in a non-leaf directory, so meta.subdir.mk supports that.

ALL_MACHINES

Another useful trick enabled by having `Makefile.depend.${MACHINE}` is that one can (from any location) do:

```
mk -DALL_MACHINES
```

to tell dirdeps.mk to get a list of all the `Makefile.depend.*` in the current directory, and for each `${machine}` add `${.CURDIR}.${machine}` to the initial `DIRDEPS`. Thus one can easily check if a change, breaks any of the supported architectures.

If there is no `Makefile.depend.*` in the current directory, but `SUBDIR` is defined, meta.subdir.mk finds all the `Makefile.depend.*` below.

sub-set checkouts

As noted below, in addition to capturing `DIRDEPS` (those directories which must be built before the current one), we can also capture `SRC_DIRDEPS` (those directories other than the current one, which must be present), which can be used to drive logic to checkout the minimal source needed to build a given directory.

Building in meta mode

Meta mode is enabled by the keyword `meta` appearing in .MAKE.MODE.

Writing `.meta` files

Generally speaking, for each target to be made, a `.meta` file is created. This is normally named `${.TARGET}.meta`. That is if `cat` is made from `cat.o` then, `cat.o.meta` and `cat.meta` will be used.

If the target is flagged `.PHONY`, `.MAKE` or `.SPECIAL` (eg. `.BEGIN`, `.END`, `.ERROR`), then a `.meta` file is not created unless the target is also flagged `.META`.

A `.meta` file is never created if the target is flagged `.NOMETA`.

If for some reason, a file is being generated outside of the current object dir the meta file will be named for the absolute path of the target with all `/` replaced with `_`.

Normally, if `.OBJDIR` is the same as `.CURDIR`, then `.meta` files will not be created. Adding `curdirOk=yes` to `.MAKE.MODE` overrides this, which can be handy when launching a sub-make in `.OBJDIR` with a generated makefile.

In each `.meta` file `bmake` records:

- the expanded command line, prefixed with `CMD`.
- the environment, prefixed with `ENV`. (this is optional, see .MAKE.MODE).
- the current directory prefixed with `CWD`.
- the target, prefixed with `TARGET`.
- the command output preceded by the line `-- command output --` This can be extremely useful in diagnosing build breaks. See Error handling below.
- syscall data collected from filemon preceded by the line `-- filemon acquired metadata --`

Bmake appends the name of each `.meta` file [re]created, to the variable `.MAKE.META.CREATED` as well as `.MAKE.META.FILES`. These variables are not used directly by `bmake` but allow for simple auto dependency extraction.

Also, in *meta verbose* mode, whenever a `.meta` file is [re]created the variable `.MAKE.META.PREFIX` is expanded and printed (if not empty). The default value is `Building ${.TARGET:H:tA}/${.TARGET:T}`.

Reading `.meta` files

When evaluating targets, if `bmake` has not already decided to re-build the target, it consults the relevant `.meta` file, processing stops as soon as an out-of-date decision is made.

You can use the debug flag `-dM` to have `bmake` explain its decisions.

First it compares the `CMD` entries with the ones currently associated with the target. If the number of commands is different, the target is considered out-of-date. If any expanded command is different and .MAKE.MODE did not contain `ignore-cmd` and the target was not flagged `.NOMETA_CMP`, and the `${.OODATE}` macro was not used, the target is considered out-of-date.

Next the sycall data is consulted. For each `E` and `R` entry, the pathname's modification time is checked and if newer than the target, it is considered out-of-date.

If `.MAKE.META.BAILIWICK` is defined, it provides a list of prefixes which describe the scope of control for `bmake`. For 'L' and 'W' entries that create files within said boundary but outside of `.OBJDIR`, we check if the target file still exists. If not, it is added to a list of potentially missing files. If a subsequent 'M' or 'D' entry explains the missing file, it is removed from the missing list. If the list is not empty by the time we reach the end of the `.meta` file, the target is considered out-of-date.

The name of each `.meta` file evaluated, is appended to the variable `.MAKE.META.FILES`.

Performance

Re-processing `.meta` files does add overhead to the build. There are potentially a *lot* more stat(2) calls performed, especially when nothing needs to be done. This is a good argument for avoiding unnecessary `#include` statements in C code.

For example, building `libc` generates nearly 3000 objects. On a given machine, in normal mode if `bmake` takes about 5 seconds to decide there is nothing to do, in *meta* mode, it will take 6 seconds to reach the same conclusion. Nothing to do is the worst case from the overhead point of view.

By contrast, when building a clean tree, generating the `.meta` files appears to be *slightly* faster (not statistically significant) compared to `*.d` files with `gcc -M*`. Considering all the extra information collected, this is good.

The more common case of re-building after some changes have been made, is where the benefits are seen. The ability to ensure everything that needs to be rebuilt (and only those) throughout the tree, whether it improves build times or not, reduces the time spent working out why something didn't work as expected.

When the entire tree can run in *meta* mode, we can generally improve the parallelism of the build. With *meta* mode, we can easily express dependencies in terms of both machines and directories - allowing much better parallelism for the problem cases outlined earlier.

Error handling

In 2001 I introduced a continuous build system (*sisyphus* - a bit like tinderbox) to find and report build breaks. The system has grown from a single node to a large cluster of machines. A critical aspect of this system is its ability to analyze build breaks and identify the offending commit. The error handling provided in *meta* mode makes that task simpler.

When a target fails, `bmake` will set `.ERROR_TARGET` to its name and in *meta* mode, `.ERROR_META_FILE` to the name of the associated `.meta` file.

These can be leveraged either by a `.ERROR` target, or simply included in the `MAKE_PRINT_VAR_ON_ERROR` list which `bmake` will print on error.

The fact that `.ERROR_META_FILE` names a file containing both the command output - presumably including any errors, as well as a record of all the files read, means that automated build break diagnosis can be greatly simplified.

For example (deliberately induced error):

```
# Meta data file /h/obj/NetBSD/5.X/usr.bin/make/make.o.meta
CMD cc -O -DMAKE_NATIVE -c /amd/mnt/swift/host/c/sjg/work/NetBSD/5.X/\
src/usr.bin/make/make.c
CWD /h/obj/NetBSD/5.X/usr.bin/make
TARGET make.o
-- command output --
/amd/mnt/swift/host/c/sjg/work/NetBSD/5.X/src/usr.bin/make/make.c:2:21: \
        error: no-such.h: No such file or directory
*** Error code 1
-- filemon acquired metadata --
# filemon version 2
# Target pid 5089
```

```
V 2
E 5175 /usr/bin/cc
R 5175 /etc/ld.so.conf
R 5175 /usr/lib/libc.so.12
W 5175 /var/tmp//cceNCjUd.s
E 5436 /usr/libexec/cc1
R 5436 /etc/ld.so.conf
R 5436 /usr/lib/libc.so.12
R 5436 /amd/mnt/swift/host/c/sjg/work/NetBSD/5.X/src/usr.bin/make/make.c
W 5436 /var/tmp//cceNCjUd.s
R 5436 /usr/share/nls/nls.alias
R 5436 /usr/share/nls/C/libc.cat
R 5436 /usr/include/sys/cdefs.h
R 5436 /usr/include/machine/cdefs.h
R 5436 /usr/include/sys/cdefs_elf.h
R 5436 /amd/mnt/swift/host/c/sjg/work/NetBSD/5.X/src/usr.bin/make/make.h
R 5436 /usr/include/sys/types.h
...
R 5436 /amd/mnt/swift/host/c/sjg/work/NetBSD/5.X/src/usr.bin/make/job.h
X 5436 1
D 5175 /var/tmp//cceNCjUd.s
X 5175 1
# Bye bye
```

We can have something like:

```
meta_error_log = ${SB}/error/meta-${.MAKE.PID}.log

.ERROR:
        -@[ "${.ERROR_META_FILE}" ] && { \
        grep -q 'failure has been detected in another branch' \
                ${.ERROR_META_FILE} && exit 0; \
        mkdir -p ${meta_error_log:H}; \
        cp ${.ERROR_META_FILE} ${meta_error_log}; \
        echo "ERROR: log ${meta_error_log}" >&2; }; :
```

to gather the `meta-*.log` files in a convenient location for automated analysis. Note that we suppress complaints about the build having failed elsewhere.

.MAKE.MODE

This variable is processed after all makefiles have been read, and can control the behavior of `bmake`. It can contain various words which are covered in the man page, for now:

`compat`
: puts `bmake` into `compat` mode (the `-B` command line option sets `.MAKE.MODE=compat`). Many makefiles are written with implicit dependencies which only work when targets are made in the order specified. These makefiles can be run in `meta compat` mode.

  Some makefiles rely on `stderr` being separated from `stdout`. Such makefiles need to run in `compat` mode.

`meta`
: puts `bmake` into `meta` mode. The following are only relevant in this case (any combination will do):

  `verbose`
  : When generating or updating a `.meta` file, print the value of `.MAKE.META.PREFIX`. The default is `Building ${.TARGET:H:tA}/${.TARGET:T}`.

`nofilemon`
: Do not attempt to use filemon. For a one time clean tree build, there is no benefit in capturing the system call activity. The `.meta` files however are still useful for capturing error output.

`ignore-cmd`
: Some makefiles have commands which are simply not stable. This tells `bmake` to not consider a target out-of-date due to a change of command. A change in the number of commands will still make the target out-of-date. The same effect can be had on a per target basis using the special source `.NOMETA_CMP`.

`curdirOk=yes`
: If running a generated makefile via a sub-make in `.OBJDIR`, `.OBJDIR` and `.CURDIR` can be the same, this knob allows us to still get `.meta` files.

For example:

```
.MAKE.MODE = meta verbose
.MAKE.MODE = compat
```

some makefiles want to ensure they run in compat mode regardless of meta mode:

```
.MAKE.MODE += compat
```

can take care of that. In some cases, a makefile does not want to run in meta mode, but does not want to run in compat mode either:

```
.MAKE.MODE = normal
```

the actual value in that case is unimportant (so long as it contains neither `meta` nor `compat`).

filemon

This is a kernel module which wraps system calls that are of interest to make. It is a clone device, so each time it is opened a new instance is created. When `bmake` forks a child, the child associates itself with the relevant filemon device.

Then any of the wrapped system calls performed by the child and its descendants will be recorded.

Filemon only records *successful* syscalls. This limits the data collected to only that which we are interested in.

Similar information could be obtained by ktrace(8), but ktrace captures *all* syscalls including those which fail, which makes it harder to gather the relevant data.

The prototype `build` (later `jbuild`) used DTrace for tracking the system calls of interest to the build. On most systems however, this required special (eg. `root`) privilege - which is not desirable. Also DTrace's probes in `exec(2)` can report `argv[0]` but that isn't always useful - when found via the path. So we asked John to implement a kernel module (`filemon`) instead.

The original `filemon` is available in FreeBSD, and a version has been contributed to NetBSD as well.

For each syscall, an entry of the form:

```
tag pid data
```

is added, where `data` is usually a pathname and `tag` is one of:

```
C       chdir
D       unlink
E       exec
F       [v]fork
L       [sym]link
M       rename
R       open for read
S       stat
W       open for write
X       exit
```

as noted below, the `C` `E` and `R` entries are of particular interest to `bmake`.

Extracting dependencies

While `bmake` itself simply uses `.meta` files to help evaluate the out-of-date status of targets, and to capture command output for diagnosis purposes, there is lots of useful data collected from filemon which can be easily leveraged.

For example, we can extract a list of all the files opened for reading. We can split these into two sets:

generated files

Any file that appears in the object tree of our sandbox, is a generated file, and by definition needs to be up to date before the current target is made.

If that generated file is not in the current object directory, we have detected a directory which needs to be visited before the current one.

Ensuring that the layout with the object trees mirror that within the src tree is a trivial means of being able to map generated files back to their src directories.

src files

Any file read from the src tree is one that must exist for the current target to be built. If that file is outside of the current directory, then it represents a directory which must be present in the tree.

These src dependencies can be leveraged to drive minimal subset checkout logic.

Makefiles

The following sections provide some detail about some example makefiles that leverage *meta* mode to improve build performance and functionality.

Makefile.depend

The build and src dependencies can be collected as relative paths (from the top of the tree), into a generated file that can be checked into the SCM.

This is the most visible change, since every leaf directory gets one or more of these.

Note that the name `Makefile.depend` is just an example (though not a bad one ;-) it is just what I set as the value for `.MAKE.DEPENDFILE`.

To support support building for multiple target machines at the same time a better value for `.MAKE.DEPENDFILE` is `Makefile.depend.${MACHINE}`. I use the unqualified name in discussions when it makes no difference.

The per `${MACHINE}` depend file, avoids the need for a mutex when updating, and avoids the need for any cleverness in representing machine specific paths in canonical forms (for example; replacing `sys/i386/include` with `sys/${MACHINE_ARCH}/include`.) The simplicity hopefully trumps the overhead of many almost identical small files.

In addition, we can extract local dependencies needed for a parallel build in a clean tree. That is; any file read from the current object directory is a local dependency for the target being made. If this information is being recorded in `Makefile.depend`, then it is wise to (if necessary) fake entries for profiled objects (`.po`) to avoid needless churn when some builds are done with profiling enabled and some are not.

For example, given the makefile:

```
PROG = ${.CURDIR:T}

SRCS = getdate.c  main.c

YACC ?= yacc
DELAY ?= 1

getdate.h:      getdate.y
        ${YACC} -d ${.ALLSRC:M*.y}
        mv y.tab.c $*.c
        sleep ${DELAY}
        mv y.tab.h $*.h

getdate.c: getdate.h

.include <bsd.prog.mk>
```

there is a missing dependency (assuming `main.c` includes `getdate.h`). Without addressing that, a `bmake -j8` in a clean directory will likely fail. The `sleep` is just there to help exercise the race condition.

However, if we end up with something like the following in `Makefile.depend`:

```
# Autogenerated - do NOT edit!

DEP_RELDIR := ${_PARSEDIR:S,${SRCTOP}/,,}

DEP_MACHINE := ${.PARSEFILE:E}

DIRDEPS = \
        lib/libc

.include <dirdeps.mk>

.if ${DEP_RELDIR} == ${_DEP_RELDIR} && !exists(.depend)
# local dependencies - needed for -jN in clean tree
main.o: getdate.h
main.po: getdate.h
.endif
```

The missing dependency is taken care of.

Obviously, if the programmer was smart enough to get the dependencies for `getdate.c` right, he likely wouldn't have missed that for `main.o` but not all makefiles are this simple, and the dependency on `getdate.h` may have been added later.

One build product per directory

Actually that is an exaggeration. There is no problem with building multiple things (like the various flavors of a library that `bsd.lib.mk` generates), so long as the behavior is consistent.

Since `Makefile.depend*` are intended to be checked into the SCM, they must be stable. For most makefiles this is a non-issue, but for others some re-design may be needed. Stable dependencies are a pre-requisite for declaring a conversion to *meta* mode complete.

To avoid capturing spurious dependencies meta.autodep.mk only considers updating `Makefile.depend*` if we successfully built the default or `all` target. Thus doing `bmake cscope` or `bmake etags` does not perturb the captured dependencies.

Given the above, makefiles which build multiple things, by use of sub-makes running in the same directory for different targets, will not have all their dependencies captured correctly.

In some cases this does not matter. For example `progs.mk` can be used to build lots of apps in the same directory. In *meta* mode it selects just one of the `PROG`'s to be the one for which dependencies will be captured. If all the apps built in that directory have essentially similar dependencies this *just works*.

In other cases, re-organizing to have sub-dirs per build product such that each can simply be built via the `all` target eliminates any problems.

Separate MACHINE independent activity

Many makefiles - for example `bin/sh/Makefile` generate host tools (tools used during their build). Moving the building of these tools into subdirs allows them to be built once, rather than once per target machine.

The same idea can be used for the output of various code generators. Whether this is a significant optimization depends on the cost of running the code generators or building tools, and the number of machines to be built for.

The Junos build makes heavy use of generated code, and builds for many different machine types, so the savings are worth it. Even so, these are optimizations that can be introduced over time.

meta.autodep.mk

Since the extraction of build dependencies from the `.meta` files is controlled by the makefiles (if done at all), it is desirable to avoid running that process unnecessarily.

The fact that `bmake` tracks updated `.meta` files via `.MAKE.META.CREATED` makes it possible to optimize the updating of dependencies, with a `meta.autodep.mk` which is simpler than the old `autodep.mk`. The following is somewhat simplified:

```
.END:           gendirdeps

_DEPENDFILE := ${.CURDIR}/${.MAKE.DEPENDFILE:T}
gendirdeps:     ${_DEPENDFILE}

# the double $$ defers initial evaluation
${_DEPENDFILE}: $${.MAKE.META.CREATED} ${.PARSEDIR}/gendirdeps.mk
        @echo Updating $@: ${.OODATE:T:[1..8]}
        @(cd ${.CURDIR} && \
        SKIP_DIRDEPS='${SKIP_DIRDEPS:O:u}' \
        ${.MAKE} __objdir=${_OBJDIR} -f gendirdeps.mk $@ \
        META_FILES='${.MAKE.META.FILES:T:O:u}' )
```

As noted the double `$$` in the dependency line, prevents `.MAKE.META.CREATED` being expanded immediately, which works to our advantage. Also note: we use `.MAKE.META.CREATED` only to know that an update is needed, `.MAKE.META.FILES` is what we use for the update:

```
$ vi cat.c
$ mk
Checking /c/sjg/work/sb/src/bsd/gnu/lib/csu for i386 ...
Checking /c/sjg/work/sb/src/bsd/lib/csu/i386-elf for i386 ...
Checking /c/sjg/work/sb/src/bsd/include for i386 ...
Checking /c/sjg/work/sb/src/bsd/usr.bin/rpcgen for host ...
Checking /c/sjg/work/sb/src/bsd/include/rpc for i386 ...
Checking /c/sjg/work/sb/src/bsd/include/rpcsvc for i386 ...
Checking /c/sjg/work/sb/src/bsd/lib/libc for i386 ...
Building /c/sjg/work/sb/obj-i386/bsd/bin/cat/cat.o
Building /c/sjg/work/sb/obj-i386/bsd/bin/cat/cat
Updating /c/sjg/work/sb/src/bsd/bin/cat/Makefile.depend.i386: cat.o.meta
$ mk
Checking /c/sjg/work/sb/src/bsd/gnu/lib/csu for i386 ...
Checking /c/sjg/work/sb/src/bsd/lib/csu/i386-elf for i386 ...
Checking /c/sjg/work/sb/src/bsd/include for i386 ...
Checking /c/sjg/work/sb/src/bsd/usr.bin/rpcgen for host ...
Checking /c/sjg/work/sb/src/bsd/include/rpc for i386 ...
Checking /c/sjg/work/sb/src/bsd/include/rpcsvc for i386 ...
Checking /c/sjg/work/sb/src/bsd/lib/libc for i386 ...
$
```

This model also works independently of the tool-chains being used, whereas `gcc -M*` requires use of `-MF` and `-MT` to do a decent job. Not to mention other languages.

If you know that the rest of the tree is up to date, you can tell dirdeps.mk to skip checking:

```
$ vi cat.c
$ mk -DNO_DIRDEPS
Building /c/sjg/work/sb/obj-i386/bsd/bin/cat/cat.o
Building /c/sjg/work/sb/obj-i386/bsd/bin/cat/cat
Updating /c/sjg/work/sb/src/bsd/bin/cat/Makefile.depend.i386: cat.o.meta
$ mk -DNO_DIRDEPS
$
```

Note that `-DNO_DIRDEPS` only supresses `DIRDEPS` outside of `.CURDIR` so you can:

```
$ mk-host -DNO_DIRDEPS -C external/bsd/atf/tests
```

to build and run all the ATF unit tests on the build host. But not waste time checking that things outside of that subtree are up to date.

.depend

In *meta* mode, we still use `.depend` but only for the local dependencies - that we will end up capturing in `Makefile.depend`. This happens independently of updating `Makefile.depend`.

It can be dangerous to look too deep into how a target is made. When temporary files get generated, and post processed, they can result in spurious (even circular) dependencies being recorded. To avoid this, `meta.autodep.mk` has a list of dependency patterns to ignore.

gendirdeps.mk

This is the makefile which extracts `DIRDEPS` and `SRC_DIRDEPS` from a bunch of `.meta` files. The process for doing this can vary. It is almost as complicated as dirdeps.mk

For example, the Junos build leverages the notion of a *sandbox* which is just a tree with a marker at its top, which is used to condition the environment. The location of that marker defines `${SB}`, which has many uses. In the discussion below, assume that:

```
SRCTOP = ${SB}/src
OBJTOP = ${SB}/obj/${MACHINE}
OBJROOT = ${SB}/obj/
```

In our example, `gendirdeps.mk` uses a shell script (`meta2deps.sh`) to look at all the paths read, and executed during the build, converts them to absolute paths, ignores any which are outside of `${SB}`, and then those which match `${SRCTOP}/*` are put into the `SRC_DIRDEPS` list, and the rest into `DIRDEPS`. Also, if a object directory referenced is outside `${OBJTOP}` (ie. directory was built for another value of `MACHINE`), the dirdep entry is left qualified with that machine value.

In general one can use `:S,${OBJTOP},${SRCTOP},` to map an object directory to the corresponding src directory, and `:S,${SRCTOP}/,,` to convert that to a tree relative path - which is what we want in `DIRDEPS` etc.

As mentioned earlier, using headers from their src location may not be ideal. If generated files are collected together in a common directory however, (eg. `${SB}/obj/common/include/`) the mapping described above will fail. By putting beside each such `file` a `file.dirdep` which contains the correct relative path to the src directory, this problem is solved.

meta.stage.mk

This makefile links or copies files into their *staging* locations, and ensures that a `.dirdep` file is associated with each one, so that `meta2deps.sh` can do its job.

You can think of it as doing `make install` as you go.

It handles multiple `STAGE_SETS` each with their own target directory, and even `STAGE_AS_SETS` where the names of the staged files do not match the src. These are handled in much the same was as symlinks.

dirdeps.mk

All of the logic for dealing with `DIRDEPS` is encapsulated in `dirdeps.mk` which is included by `Makefile.depend.${MACHINE}`, and is only relevant for the initial instance of `bmake` (`${.MAKE.LEVEL} == 0`).

Conceptually, the process is quite simple (even if the implementation is not):

When the initial `bmake` reads `${.CURDIR}/Makefile.depend.${MACHINE}`, it gets an initial set of `DIRDEPS`.

`dirdeps.mk` transforms `DIRDEPS` into a set of absolute paths with a `.${MACHINE}` suffix, to deal with building for multiple machine types. The pseudo machine `.host` represents the build host. These are hooked into a dependency for the target `dirdeps`.

`dirdeps.mk` also turns `DIRDEPS` into a list of `*/Makefile.depend*` files which will be read, to get more `DIRDEPS`.

Since each `Makefile.depend*` includes `dirdeps.mk` the process is recursive, at each point adding something like:

```
${SRCTOP}/${DEP_RELDIR}.${MACHINE}: ${DIRDEPS:@d@${SRCTOP}/$d.${MACHINE}@}
```

to the graph.

It is actually more complicated than that, to deal with cases where `${DEP_RELDIR}.${MACHINE}` depends on dirs built for other machines (eg. pseudo machines like `host` for host tools). But the above shows how a tree-wide set of dependencies are built.

You can gather a lot of information from `dirdeps.mk` by setting `DEBUG_DIRDEPS` to a list of directories of interest (`*` for all). If you recall the *teaser* example from earlier:

```
$ mk -n -C bin/sh DEBUG_DIRDEPS='*' |
        sed -n "/275:/ { s,.*275:,,;s,${SB}/src/,,g;p; }"
```

produces the output below (reformatted for readability):

```
bin/sh: \
       bin/sh.i386 \
       include.i386 \
       lib/libc.i386 \
       lib/libedit.i386 \
       lib/ncurses/ncurses.i386 \
       stage.i386 \
       sys/i386/include.i386 \
       sys/sys.i386 \
       sys/x86/include.i386

lib/libc.i386: \
       include.i386 \
       lib/msun.i386 \
       stage.i386 \
       sys/i386/include.i386 \
       sys/sys.i386 \
       sys/x86/include.i386

lib/msun.i386: \
       include.i386 \
       stage.i386 \
       sys/i386/include.i386 \
       sys/sys.i386

lib/libedit.i386: \
       include.i386 \
       lib/ncurses/ncursesw.i386 \
       stage.i386 \
```

```
        sys/i386/include.i386 \
        sys/sys.i386 \
        sys/x86/include.i386

lib/ncurses/ncursesw.i386: \
        include.i386 \
        stage.i386 \
        sys/i386/include.i386 \
        sys/sys.i386 \
        sys/x86/include.i386
```

All the expanded DIRDEPS are associated with a build macro which will cause them to be visited with MACHINE set to the correct value:

```
# we suppress SUBDIR when visiting the leaves
_DIRDEP_USE:    .USE .MAKE
        @for m in ${.MAKE.MAKEFILE_PREFERENCE}; do \
                test -s ${.TARGET:R}/$$m || continue; \
                echo "${TRACER}Checking ${.TARGET:R} for ${.TARGET:E} ..."; \
                MACHINE=${.TARGET:E} MACHINE_ARCH= NO_SUBDIR=1 \
                ${.MAKE} -C ${.TARGET:R} || exit 1; \
                break; \
        done
```

Note: clearing MACHINE_ARCH as above assumes that sys.mk will set it to something appropriate for ${MACHINE} if necessary.

The distributed version of dirdeps.mk as extra checks to handle ${.MAKE.DEPENDFILE} not being qualified with ${MACHINE}. Regardless of whether ${.MAKE.DEPENDFILE} is qualified with ${MACHINE} or not, dirdeps.mk always qualifies the dependencies it constructs.

Also, like many of the makefiles in the mk-files distribution, dirdeps.mk will include local.dirdeps.mk if it exists, so that local customizations can be accommodated without the need to touch the original makefile.

For the build geeks

Here for your reading pleasure, is the full (with the exception of the BSD license) text of dirdeps.mk, there are a couple of extra line breaks. Read carefully - there will be a test!

```
# Copyright (c) 2010, Juniper Networks, Inc.
```

Two clause BSD license elided.

```
# Much of the complexity here is for supporting cross-building.
# If a tree does not support that, simply using plain Makefile.depend
# should provide sufficient clue.
# Otherwise the recommendation is to use Makefile.depend.${MACHINE}
# as expected below.

# Note: this file gets multiply included.
# This is what we do with DIRDEPS

# DIRDEPS:
#       This is a list of directories - relative to SRCTOP, it is only
#       of interest to .MAKE.LEVEL 0.
#       In some cases the entry may be qualified with a .<machine>
#       suffix, for example to force building something for the pseudo
#       machines "host" or "common" regardless of current ${MACHINE}.
#       All unqualified entries end up being qualified with .${MACHINE}
```

```
#       and _DIRDEPS_USE below, uses the suffix to set MACHINE
#       correctly when visiting each entry.
#
#       Each entry is also converted into a set of paths to look for
#       Makefile.depend.<machine> to learn the dependencies of each.
#       Each Makefile.depend.<machine> sets DEP_RELDIR to be the
#       the RELDIR (path relative to SRCTOP) for its directory, and
#       DEP_MACHINE to its suffix (<machine>), further since
#       each Makefile.depend.<machine> includes dirdeps.mk, this
#       processing is recursive and results in .MAKE.LEVEL 0 learning the
#       dependencies of the tree wrt the initial directory (_DEP_RELDIR).
#
# BUILD_AT_LEVEL0
#       Indicates whether .MAKE.LEVEL 0 builds anything:
#       if "no" sub-makes are used to build everything,
#       if "yes" sub-makes are only used to build for other machines.

.if ${.MAKE.LEVEL} == 0
# only the first instance is interested in all this

# pickup customizations
# as below you can use !target(_DIRDEP_USE) to protect things
# which should only be done once.
.-include "local.dirdeps.mk"

# the first time we are included the _DIRDEP_USE target will not be defined
# we can use this as a clue to do initialization and other one time things.
.if !target(_DIRDEP_USE)
# make sure this target exists
dirdeps:

# We normally expect to be included by Makefile.depend.*
# which sets the DEP_* macros below.
DEP_RELDIR ?= ${RELDIR}

# this get's set by Makefile.depend.*
# but that may not have been included yet.
DEP_MACHINE ?= ${TARGET_MACHINE:U${.MAKE.DEPENDFILE:E}}

# if we are using just Makefile.depend DEP_MACHINE will likely be wrong
.if ${DEP_MACHINE} == "depend"
DEP_MACHINE = ${MACHINE}
# nothing else makes sense
ONLY_MACHINE_LIST ?= ${MACHINE}
.endif

# this can cause lots of output!
# set to a set of glob expressions that might match RELDIR
DEBUG_DIRDEPS ?= no

# remember the initial value of DEP_RELDIR - we test for it below.
_DEP_RELDIR := ${DEP_RELDIR}

# things we skip for host tools
SKIP_HOSTDIR ?=

NSkipHostDir = ${SKIP_HOSTDIR:N*.host:S,$,.host,:N.host:${M_ListToSkip}}
NSkipHostDep = ${SKIP_HOSTDIR:R:@d@*/$d*.host@:${M_ListToSkip}}

# things we always skip
# SKIP_DIRDEPS allows for adding entries on command line.
SKIP_DIR += .host *.WAIT ${SKIP_DIRDEPS}

.ifdef HOSTPROG
SKIP_DIR += ${SKIP_HOSTDIR}
.endif
```

```
NSkipDir = ${SKIP_DIR:${M_ListToSkip}}

.if defined(NO_DIRDEPS) || defined(NODIRDEPS)
# confine ourselves to the original dir
DIRDEPS_FILTER += M${_DEP_RELDIR}*
.endif

# we supress SUBDIR when visiting the leaves
_DIRDEP_USE:    .USE .MAKE
        @for m in ${.MAKE.MAKEFILE_PREFERENCE}; do \
                test -s ${.TARGET:R}/$$m || continue; \
                echo "${TRACER}Checking ${.TARGET:R} for ${.TARGET:E} ..."; \
                MACHINE=${.TARGET:E} MACHINE_ARCH= NO_SUBDIR=1 \
                ${.MAKE} -C ${.TARGET:R} || exit 1; \
                break; \
        done

.ifdef ALL_MACHINES
# this is how you limit it to only the machines we have been built for
# previously.
.if empty(ONLY_MACHINE_LIST)
.if !empty(ALL_MACHINE_LIST)
# ALL_MACHINE_LIST is the list of all legal machines - ignore anything else
_machine_list != cd ${_CURDIR} && 'ls' -1 \
${ALL_MACHINE_LIST:O:u:@m@${.MAKE.DEPENDFILE:T:R}.$m@} 2> /dev/null; echo
.else
_machine_list != 'ls' -1 \
${_CURDIR}/${.MAKE.DEPENDFILE:T:R}.* 2> /dev/null; echo
.endif
_only_machines := ${_machine_list:${NIgnoreFiles:UN*.bak}:E:O:u}
.else
_only_machines := ${ONLY_MACHINE_LIST}
.endif

.if empty(_only_machines)
# we must be boot-strapping
_only_machines := ${TARGET_MACHINE:U${ALL_MACHINE_LIST:U${DEP_MACHINE}}}
.endif

.else                           # ! ALL_MACHINES
# if ONLY_MACHINE_LIST is set, we are limited to that
# if TARGET_MACHINE is set - it is really the same as ONLY_MACHINE_LIST
# otherwise DEP_MACHINE is it - so DEP_MACHINE will match.
_only_machines := \
${ONLY_MACHINE_LIST:U${TARGET_MACHINE:U${DEP_MACHINE}}:M${DEP_MACHINE}}
.endif

.if !empty(NOT_MACHINE_LIST)
_only_machines := ${_only_machines:${NOT_MACHINE_LIST:${M_ListToSkip}}}
.endif

# make sure we have a starting place?
DIRDEPS ?= ${RELDIR}
.endif                          # target

# the rest is done repeatedly for every Makefile.depend we read.
# if we are anything but the original dir we care only about the
# machine type we were included for..

# if we are using just Makefile.depend DEP_MACHINE will likely be wrong
.if ${DEP_MACHINE} == "depend"
DEP_MACHINE = ${MACHINE}
.endif

.if ${DEP_RELDIR} == "."
```

```
_this_dir := ${SRCTOP}
.else
_this_dir := ${SRCTOP}/${DEP_RELDIR}
.endif

# on rare occasions, there can be a need for extra help
.if ${.MAKE.DEPENDFILE:E} == "depend"
_dep_hack := ${_this_dir}/${.MAKE.DEPENDFILE:T}.inc
.else
_dep_hack := ${_this_dir}/${.MAKE.DEPENDFILE:T:R}.inc
.endif
.if ${.MAKE.MAKEFILES:M${_dep_hack}} == ""
.-include "${_dep_hack}"
.endif

.if ${DEP_RELDIR} != ${_DEP_RELDIR} || ${DEP_MACHINE} != ${MACHINE}
# this should be all
_machines = ${DEP_MACHINE}
.else
# this is the machine list we actually use below
_machines := ${_only_machines}

.if defined(HOSTPROG) || ${DEP_MACHINE} == "host"
# we need to build this guy's dependencies for host as well.
_machines += host
.endif

_machines := ${_machines:O:u}
.endif

_build_dirs =
_depdir_files =

.if ${DEP_RELDIR} == ${_DEP_RELDIR}
# pickup other machines for this dir if necessary
.if ${BUILD_AT_LEVEL0:Uyes} == "no"
_build_dirs += ${_machines:@m@${_CURDIR}.$m@}
.else
_build_dirs += ${_machines:N${DEP_MACHINE}:@m@${_CURDIR}.$m@}
.if ${DEP_MACHINE} == ${MACHINE}
# pickup local dependencies now
.-include <.depend>
.endif
.endif
.endif

.if ${DEBUG_DIRDEPS:@x@${DEP_RELDIR:M$x}${${DEP_RELDIR}.\
${DEP_MACHINE}:L:M$x}@} != ""
.info ${DEP_RELDIR}.${DEP_MACHINE}: DIRDEPS='${DIRDEPS}'
.info ${DEP_RELDIR}.${DEP_MACHINE}: _machines='${_machines}'
.endif

.if !empty(DIRDEPS)

# this is what we start with
__depdirs := ${DIRDEPS:${NSkipDir}:${DIRDEPS_FILTER:ts:}:O:u:@d@${SRCTOP}/$d@}

# some entries may be qualified with .<machine>
# the :M*/*/*.* just tries to limit the dirs we check to likely ones.
# the ${d:E:M*/*} ensures we don't consider junos/usr.sbin/mgd
__qual_depdirs := ${__depdirs:M*/*/*.*:@d@${exists($d):?:\
${"${d:E:M*/*}":?:${exists(${d:R}):?$d:}}}@}
__unqual_depdirs := ${__depdirs:${__qual_depdirs:Uno:${M_ListToSkip}}}

.if ${DEP_RELDIR} == ${_DEP_RELDIR}
# if it was called out - we likely need it.
```

```
__hostdpadd := ${DPADD:U.:M${HOST_OBJTOP}/*:S,${HOST_OBJTOP}/,,:H:\
${NSkipDir}:${DIRDEPS_FILTER:ts:}:S,$,.host,:N.*:@d@${SRCTOP}/$d@}
__qual_depdirs += ${__hostdpadd}
.endif

.if ${DEBUG_DIRDEPS:@x@${DEP_RELDIR:M$x}${${DEP_RELDIR}.\
${DEP_MACHINE}:L:M$x}@} != ""
.info depdirs=${__depdirs}
.info qualified=${__qual_depdirs}
.info unqualified=${__unqual_depdirs}
.endif

# _build_dirs is what we will feed to _DIRDEP_USE
_build_dirs += \
        ${__qual_depdirs:M*.host:${NSkipHostDir}:N.host} \
        ${__qual_depdirs:N*.host} \
        ${_machines:@m@${__unqual_depdirs:@d@$d.$m@}@}

_build_dirs := ${_build_dirs:O:u}

# this is where we will pick up more dependencies from
.if ${.MAKE.DEPENDFILE:E} == "depend"
_depdir_files += \
        ${_build_dirs:@d@${d:R}/${.MAKE.DEPENDFILE:T}@}
.else
_depdir_files += \
        ${_build_dirs:@d@${d:R}/${.MAKE.DEPENDFILE:T:R}.${d:E}@}
.endif

_depdir_files := ${_depdir_files:O:u}

.endif                          # empty DIRDEPS

.if ${.MAKEFLAGS:M-V} == ""
.if !empty(_build_dirs)
# this makes it all happen
dirdeps: ${_build_dirs}
${_build_dirs}: _DIRDEP_USE

.if ${DEBUG_DIRDEPS:@x@${DEP_RELDIR:M$x}${${DEP_RELDIR}.\
${DEP_MACHINE}:L:M$x}@} != ""
.info ${DEP_RELDIR}.${DEP_MACHINE}: ${_build_dirs}
.endif

.for m in ${_machines}
# it would be nice to do :N${.TARGET}
.if !empty(__qual_depdirs)
.for q in ${__qual_depdirs:E:O:u:N$m}
.if ${DEBUG_DIRDEPS:@x@${DEP_RELDIR:M$x}${${DEP_RELDIR}.$m:L:M$x}\
${${DEP_RELDIR}.$q:L:M$x}@} != ""
.info ${DEP_RELDIR}.$m: ${_build_dirs:M*.$q}
.endif
${_this_dir}.$m: ${_build_dirs:M*.$q}
.endfor
.endif
.if ${DEBUG_DIRDEPS:@x@${DEP_RELDIR:M$x}${${DEP_RELDIR}.$m:L:M$x}@} != ""
.info ${DEP_RELDIR}.$m: ${_build_dirs:M*.$m:N${_this_dir}.$m}
.endif
${_this_dir}.$m: ${_build_dirs:M*.$m:N${_this_dir}.$m}
.endfor

.endif

.for d in ${_depdir_files}
.if ${.MAKE.MAKEFILES:M${d}} == ""
.if ${DEBUG_DIRDEPS:@x@${DEP_RELDIR:M$x}${${DEP_RELDIR}.depend:L:M$x}@} != ""
```

```
.info Looking for $d
.endif
.-include <$d>
.if ${.MAKE.DEPENDFILE:E} != "depend" && ${.MAKE.MAKEFILES:M${d}} == ""
# see if the unqualified file exists
# it might be manually maintained and shared by all machine types
.-include <${d:R}>
.endif
.endif
.endfor
.endif                          # -V

.elif ${.MAKE.LEVEL} > 42
.error You should have stopped recursing by now.
.else
_DEP_RELDIR := ${DEP_RELDIR}
# pickup local dependencies
.-include <.depend>
.endif
```

meta.subdir.mk

While our goal is to build by visiting the tree's leaf nodes directly, we still need to be able to launch a build in say `src/lib/` to build all the libraries there - perhaps to check we didn't break any.

If there is no `Makefile.depend` in the current directory, `meta.subdir.mk` does something like:

```
_subdirs != find ${SUBDIR} -name 'Makefile.depend*'
```

and assigns the cleaned up result to `DIRDEPS`. Thus the initial `DIRDEPS` includes all the leaf directories below the current one. Of course it is actually more complex than that, but that's the basic idea. Once `DIRDEPS` is set, including `dirdeps.mk` does the rest.

meta.sys.mk

This makefile exists mainly to facilitate having *meta* mode as an option for example during a transition period. For example, in `sys.mk` one can have:

```
.if ${USE_META:Uno} == "yes"
.-include <meta.sys.mk>
.endif
# make sure we have a harmless value
.MAKE.MODE ?= normal
```

and the distributed `meta.sys.mk` contains things like:

```
# include this if you want to enable meta mode
# for maximum benefit, requires DTrace or the filemon(9) driver.

.if ${MAKE_VERSION:U0} > 20100901
.if !target(.ERROR)

META_MODE += meta verbose
.MAKE.MODE ?= ${META_MODE}

# make defaults .MAKE.DEPENDFILE to .depend
# that won't work for us.
.if ${.MAKE.DEPENDFILE} == ".depend"
.undef .MAKE.DEPENDFILE
```

```
.endif

# if you don't cross build for multiple MACHINEs concurrently, then
# .MAKE.DEPENDFILE = Makefile.depend
# probably makes sense - you can set that in local.sys.mk
.MAKE.DEPENDFILE ?= Makefile.depend.${MACHINE}

# we use the pseudo machine "host" for the build host.
# this should be taken care of before we get here
.if ${OBJTOP:Ua} == ${HOST_OBJTOP:Ub}
MACHINE = host
.endif

MAKE_PRINT_VAR_ON_ERROR += \
        .ERROR_TARGET \
        .ERROR_META_FILE \
        .MAKE.LEVEL \
        MAKEFILE \
        .MAKE.MODE

.if !defined(SB) && defined(SRCTOP)
SB = ${SRCTOP:H}
.endif
ERROR_LOGDIR ?= ${SB}/error
meta_error_log = ${ERROR_LOGDIR}/meta-${.MAKE.PID}.log

# we are not interested in make telling us a failure happened elsewhere
.ERROR:
        -@[ "${.ERROR_META_FILE}" ] && { \
        grep -q 'failure has been detected in another branch' \
                ${.ERROR_META_FILE} && exit 0; \
        mkdir -p ${meta_error_log:H}; \
        cp ${.ERROR_META_FILE} ${meta_error_log}; \
        echo "ERROR: log ${meta_error_log}" >&2; }; :

.endif

# Are we, after all, in meta mode?
.if ${.MAKE.MODE:Mmeta*} != ""
MKDEP = meta.autodep

.if ${.MAKE.LEVEL} == 0
# make sure dirdeps target exists and do it first
all: dirdeps .WAIT
dirdeps:

.if ${.MAKE.DEPENDFILE:E} != "depend"
# it works best if we do everything via sub-makes
BUILD_AT_LEVEL0 ?= no
.endif
BUILD_AT_LEVEL0 ?= yes
.if ${.MAKE.MODE:M*nofilemon*} != "" || ${.MAKE.MODE:M*read*} != ""
UPDATE_DEPENDFILE = NO
.export UPDATE_DEPENDFILE
.endif
.endif

# if we think we are updating dependencies,
# then filemon had better be present
.if ${UPDATE_DEPENDFILE:Uyes:tl} != "no" && !exists(/dev/filemon)
.error ${.newline}ERROR: The filemon module (/dev/filemon) is not loaded.
.endif

.endif
.endif
```

That last check works best if `/dev/filemon` automatically disappears when the module is not loaded.

Also, as with `dirdeps.mk` there are some macros you'll need from my `sys.mk` and friends if integrating this into something else.

BUILD_AT_LEVEL0

As noted above, if you are always building for multiple machine types, it can be best to reserve level 0 for just computing the tree wide dependency graph, and do all the actual building at level 1 or greater.

Building kernels

The stock BSD kernel build does not lend itself well to capturing dependencies since there isn't normally a src directory as such.

We've worked around this, by providing src directories for the kernels. From `jnx.kernel.mk`:

```
# for each kernel we have:
# ${KERNEL_NAME}/config/
# ${KERNEL_NAME}/kernel/
# and possibly?
# ${KERNEL_NAME}/modules/*
#
# config/ is where config(8) is run
# both kernel/ and modules that need to link with it
# can depend on config/
# If there are kernel specific modules (which do not link into it)
# they could be built under modules/ (one directory each of course)
#
# For example:
#       bsd/kernels/JUNIPER/config
#       bsd/kernels/JUNIPER/kernel
#
# Because config(8) produces a Makefile which we want to use,
# the makefiles in config/ and kernel/ above should be called 'makefile'.
```

All that `makefile` does is `.include <jnx.kernel.mk>` The `Makefile.depend.i386` in `bsd/kernels/JUNIPER/config` says:

```
# Autogenerated - do NOT edit!

DEP_RELDIR := ${_PARSEDIR:S,${SRCTOP}/,,}

DEP_MACHINE := ${.PARSEFILE:E}

DIRDEPS = \
        bsd/usr.sbin/config.host \


SRC_DIRDEPS = \
        bsd/sys/conf \
        bsd/sys/i386/conf \
        bsd/sys/x86/conf \


.include <dirdeps.mk>

.if ${DEP_RELDIR} == ${_DEP_RELDIR} && !exists(.depend)
```

```
# local dependencies - needed for -jN in clean tree
.endif
```

and in `bsd/kernels/JUNIPER/kernel`:

```
# Autogenerated - do NOT edit!

DEP_RELDIR := ${_PARSEDIR:S,${SRCTOP}/,,}

DEP_MACHINE := ${.PARSEFILE:E}

DIRDEPS = \
        bsd/include \
        bsd/kernels/JUNIPER/config \


SRC_DIRDEPS = \
        bsd/include \
        ...
```

The reason for multiple directories is that any loadable modules (as well as the kernel) depend on config(8) having run. Isolating that step allows the kernel and any modules to then be built in parallel.

You can also see above that we build `bsd/usr.sbin/config` for the pseudo machine `host` before we can configure the kernel.

Top-level makefiles?

All very interesting, but what about those hideously complex top-level makefiles, you ask? They become the simplest of all.

Let's say we want to build a package `pkg-tools` that contained:

```
/usr/sbin/pkg_add
/usr/sbin/pkg_delete
/usr/sbin/pkg_info
/sbin/sha1
/sbin/verify-sig
```

We would need a makefile say `pkgs/pkg-tools/Makefile` something like:

```
# pickup the package building magic
.include "../Makefile.inc"
```

and a `Makefile.depend` something like:

```
# This file is NOT autogenerated - take care!

DEP_RELDIR := ${_PARSEDIR:S,${SRCTOP}/,,}

# we get shared by everyone
DEP_MACHINE := ${MACHINE}

DIRDEPS = \
        usr.sbin/pkg_add \
        usr.sbin/pkg_delete \
        usr.sbin/pkg_info \
        sbin/md5 \
        sbin/verify-sig

.include <dirdeps.mk>
```

and perhaps we want to include that package in a bundle package `base-os` it might have an almost identical makefile, except that the `Makefile.depend` might contain:

```
# This file is NOT autogenerated - take care!

DEP_RELDIR := ${_PARSEDIR:S,${SRCTOP}/,,}

DEP_MACHINE := ${MACHINE}

DIRDEPS = \
        pkgs/pkg-tools \
        pkgs/sys-tools \
        ...

.include <dirdeps.mk>
```

and finally, perhaps we have a bundle package which needs `base-os` for a couple of different machine types - because the target platform needs them:

```
# This file is NOT autogenerated - take care!

DEP_RELDIR := ${_PARSEDIR:S,${SRCTOP}/,,}

DEP_MACHINE := ${.PARSEFILE:E}

DIRDEPS = \
        pkgs/base-os.i386 \
        pkgs/base-os.mips \
        ...

.include <dirdeps.mk>
```

and so on. Each such makefile, is quite straight forward and easy to understand, the build structure can be built layer upon layer, and dirdeps.mk makes it all *just work*.

In fact the top-level makefile itself need be no more than:

```
DIRDEPS = ${.TARGETS:Nall:@d@pkgs/$d@}

.include <dirdeps.mk>

.for t in ${.TARGETS:Nall}
$t: dirdeps
.endfor
```

Building FreeBSD current

The example I showed at the start was building some of FreeBSD current in *meta* mode. I was doing this as an exercise to check that `dirdeps.mk` et al, were viable for use outside of Junos and also to experiment with making it easier for our engineers to build FreeBSD.

It is perhaps worth noting that the Junos tree was able to build with *meta* mode for many months prior to any of the `Makefile.depend*` being committed. This makes it possible to play with the idea with minimal impact.

Setup

I was looking to do this with minimal changes to FreeBSD, so the setup is a bit kludgy. In

addition to the normal `share/mk/*.mk` we have my mk-files distribution unpacked in `mk/` and some `local.*.mk` to provide some glue.

`sys.mk`
: This is the generic `sys.mk` from mk-files, it pulls in (if they exist):

```
$ grep '^.-*include' mk/sys.mk
.-include <sys.env.mk>
.include <host-target.mk>
.include <${SYS_OS_MK}>
.-include <sys/$x.mk>
.-include <$x.sys.mk>
.-include <local.sys.mk>
.-include <meta.sys.mk>
```

`sys.env.mk`
: does one-time things to the environment

`host-target.mk`
: defines `HOST_TARGET` and other macros needed to find a platform specific sys.mk

`sys/*.mk`
: platform specific sys makefile. For FreeBSD I added `sys/FreeBSD.mk` which contains:

```
.include "../../share/mk/sys.mk"
```

: to ensure we pull in the real `sys.mk`, since it defines lots of things that `bsd.*.mk` want.

: If no platform specific sys makefile is found, `Generic.mk` is used.

: The sys makefile used is exported as `SYS_OS_MK` so that sub-makes don't have to repeat the search.

`local.sys.mk`
: A convenient place to add local customizations:

```
# handle freebsd'isms
BINOWN = ${USER}
BINGRP = software
OBJROOT ?= ${SB_OBJROOT}

# do the right thing for mk-host and mk-hostprog
.ifdef HOSTPROG || (defined(OBJDIR) && ${OBJDIR:M*${HOST_TARGET}} != "")
MACHINE = host
MACHINE_ARCH := ${_HOST_ARCH}
OBJTOP = ${HOST_OBJTOP}
.endif

# this is handy
MACHINE_OBJ.host = ${HOST_TARGET}
MACHINE_OBJ.${MACHINE} ?= ${MACHINE}
MACHINE_OBJDIR = ${MACHINE_OBJ.${MACHINE}}

# we use this lot during the build
STAGE_ROOT=${SB_OBJROOT}stage/${MACHINE_OBJDIR}
# what we export to others
SHARED_STAGE_ROOT= ${SB}/shared/stage/${MACHINE_OBJDIR}

.if ${MACHINE} != "host"
CFLAGS += -nostdinc
CFLAGS_LAST += -I${STAGE_ROOT}/usr/include
# hack for now
CFLAGS_LAST += -I/usr/include
```

```
.endif

MAKE=${.MAKE}
ECHO=echo
# freebsd needs this
DEPENDFILE ?= .depend

MKOBJDIRS ?= auto
.if ${MKOBJDIRS} == "auto"
.include <auto.obj.mk>
.endif
.NOMETA: clean destroy obj
.if make(destroy*)
NO_SILENT=1
.endif

MAKE_PRINT_VAR_ON_ERROR ?= \
        HOSTNAME SB_LOCATION _CURDIR .CURDIR \
        _OBJTOP OBJTOP _OBJDIR .OBJDIR \
        .MAKE MAKE_VERSION \
        MACHINE_ARCH MACHINE \
        .TARGETS \
        ${MAKE_PRINT_VAR_ON_ERROR_XTRAS}
```

The `CFLAGS_LAST += -I/usr/include` hack, just allows getting things building for demo/test purposes without having to create/fix all the makefiles that would need to stage headers to `${STAGE_ROOT}/usr/include`.

`meta.sys.mk`
: Setup *meta* mode - see above.

We also have `mk/bsd.obj.mk -> obj.mk` as a means of pulling in:

`auto.obj.mk`
: ensure objdirs are created automagically.

and in addition we have:

`local.dirdeps.mk`
: Included from `dirdeps.mk` contains:

```
.if !empty(DIRDEPS)
DIRDEPS += stage
.endif
```

To ensure `stage/` gets built first (since everything depends on it) - see below.

`local.libnames.mk`
: Included from `bsd.libnames.mk` contains:

```
.ifdef PROG
LDADD+= -L${STAGE_ROOT}${LIBDIR}
.endif
```

And since this is a *sandbox* we have:

```
$ cat $SB/.sandbox-env
. ${SB}/../.sandboxrc
$ cat ${SB}/../.sandboxrc
export SB_SRC=$SB/src
export BMAKE=/homes/sjg/freebsd7-i386/bin/bmake-20110327
export MAKESYSPATH=$SB/src/mk:$SB/src/share/mk
export SB_OBJROOT=$SB/obj/
export OBJTOP='${SB_OBJROOT}${MACHINE}'
```

```
export SRCTOP=${SB}/src
export MAKEOBJDIR='${.CURDIR:S,${SRCTOP},${OBJTOP},}'
SB_PATH=$PATH
export MKOBJDIRS=auto
export USE_META=yes
export HOST_OBJTOP=${SB_OBJROOT}${HOST_TARGET}
export OBJTOP='${SB_OBJROOT}${MACHINE}'
$
```

from which you can see that `$SB/src/mk` will be searched before `$SB/src/share/mk` so that our `sys.mk` and `bsd.obj.mk` will be used by preference.

`bmake` vs FreeBSD `make`

We have been building FreeBSD with `bmake` for a long time, though we have not attempted to keep `bsd.*.mk` in sync. The majority of makefiles though *just work*.

There are a couple of modifiers that FreeBSD make has (from OpenBSD I think), which conflict with the ODE modifiers in `bmake`:

```
Modifier        FreeBSD         NetBSD

:U              to upper        value if undefined
:L              to lower        treat variable name as value
                                (ie. Literal)

:tu                             to upper
:tl                             to lower
```

It turns out though that the only usage of either `:U` or `:L` we could find in FreeBSD current was in `tools/build/make_check/Makefile` and the actual modifier used made no difference to the test case.

There were a couple of tests in that makefile though for which we decided the `bmake` behavior was incorrect - so I fixed it.

There are a couple of remaining differences which are not important. With the exception of accepting strings of `:` in file names, which I'm told doesn't matter in FreeBSD anymore?

Bmake requires you to be more explicit about some things. For example, something like:

```
.NOPATH: ${CLEANFILES}
```

tells it not to attempt to find generated files via `.PATH`.

Similarly, NetBSD's `bsd.own.mk` uses something like:

```
PHONY_NOTMAIN = all clean cleandir depend dependall distclean includes \
                install lint obj regress tags beforedepend afterdepend \
                beforeinstall afterinstall realinstall realdepend realall \
                html subdir-all subdir-install subdir-depend
.PHONY:         ${PHONY_NOTMAIN}
.NOTMAIN:       ${PHONY_NOTMAIN}
```

to ensure there is no confusion about what `make depend` means when an app just happens to have `depend.c` in its `SRCS`.

FreeBSD's make, does not seem to need that, it behaves more like Solaris's make, than

pmake, gmake or bmake. To illustrate:

```
$ cat makefile
foo:    hello

hello:
        echo $@
```

On NetBSD:

```
$ make -n foo
echo hello
cc -O2   -o foo foo.c
$ gmake -n foo
echo hello
cc     foo.c hello   -o foo
$
```

On FreeBSD:

```
$ make -n foo
echo hello
$ pmake -n foo
--- hello ---
echo hello
--- foo ---
cc -O -pipe foo.c  -o foo
```

if we add:

```
.PHONY: foo
```

to the makefile, then on NetBSD:

```
$ make -n foo
echo hello
$
```

The `.PHONY` tells make that the target `foo` does not actually make a file called `foo`.

Staging headers and libs

We might as well aim to *stage* (install) at least headers and libs as we go.

preparing the stage

In `stage/Makefile` we initialize `STAGE_ROOT` which is set to:

```
$ mk -V STAGE_ROOT
${SB_OBJROOT}stage/${MACHINE_OBJDIR}
$
```

by running `mtree`:

```
.if !empty(STAGE_ROOT)
all:    stage_include

MTREE?= mtree
MTREES= ${.CURDIR:H}/etc/mtree

stage0:
```

```
.if !exists(${STAGE_ROOT}/usr/include)
        mkdir -p ${STAGE_ROOT}/usr/include
.endif
        touch $@

stage_root:        stage0
        ${MTREE} -deUw -f ${MTREES}/BSD.root.dist -p ${STAGE_ROOT} > /dev/null
        touch $@

stage_include: stage_root
        ${MTREE} -deUw -f ${MTREES}/BSD.include.dist -p \
        ${STAGE_ROOT}/usr/include > /dev/null
        touch $@

.endif
```

headers

The makefile in `src/include` is *special*. We can opt to use it as is and make it `install` into the stage tree, we tweak it to leverage `meta.stage.mk` or we can massively simplify it to install only the headers that it owns.

In the later case, we would need Makefiles like the following:

```
INCSDIR= /usr/include/sys
INCS= \
        _bus_dma.h \
        _iovec.h \
        _lock.h \
        _lockmgr.h \
        _mutex.h \
        ...
        watchdog.h \

.include <bsd.init.mk>
.include <bsd.incs.mk>
.include <bsd.sys.mk>
```

in `sys/sys`, `sys/*/include` and pretty much anywhere else that has headers which belong in `/usr/include`

I've experimented with the last two approaches, and both have their downsides (if trying to minimize changes to FreeBSD).

The following patch to `bsd.incs.mk` does the work:

```
Index: share/mk/bsd.incs.mk
===================================================================
--- share/mk/bsd.incs.mk        (revision 219256)
+++ share/mk/bsd.incs.mk        (working copy)
@@ -24,6 +24,8 @@
 ${group}GRP?=  ${BINGRP}
 ${group}MODE?= ${NOBINMODE}
 ${group}DIR?=  ${INCLUDEDIR}
+STAGE_SETS += ${group}
+STAGE_DIR.${group} = ${STAGE_ROOT}${${group}DIR}

 _${group}INCS=
 .for header in ${${group}}
@@ -39,6 +41,10 @@
 .else
 ${group}NAME_${header:T}?=     ${header:T}
 .endif
```

```
+STAGE_AS_SETS += ${group}
+STAGE_AS_${header:T} = ${${group}NAME_${header:T}}
+stage_as.${group}: ${header}
+
 installincludes: _${group}INS_${header:T}
 _${group}INS_${header:T}: ${header}
        ${INSTALL} -C -o ${${group}OWN_${.ALLSRC:T}} \
@@ -50,6 +56,8 @@
 .endif
 .endfor
 .if !empty(_${group}INCS)
+stage_files.${group}: ${_${group}INCS}
+
 installincludes: _${group}INS
 _${group}INS: ${_${group}INCS}
 .if defined(${group}NAME)
@@ -81,4 +89,14 @@
 realinstall: installincludes
 .ORDER: beforeinstall installincludes

+.if !empty(STAGE_ROOT)
+.if !empty(STAGE_SETS)
+buildincludes: stage_files
+.if !empty(STAGE_AS_SETS)
+buildincludes: stage_as
+.endif
+.endif
+.endif
+
 .endif # !defined(NO_INCS) && ${MK_TOOLCHAIN} != "no"
```

libraries

For libraries we have:

```
Index: share/mk/bsd.lib.mk
===================================================================
--- share/mk/bsd.lib.mk (revision 219256)
+++ share/mk/bsd.lib.mk (working copy)
@@ -263,6 +263,21 @@
 .endif
 .endif

+# some libs have lots of objects, and scanning all .o, .po and .So meta files
+# is a waste of time, this tells meta.autodep.mk to just pick one
+# (typically .So)
+# yes, 42 is a random number.
+.if ${SRCS:Uno:[\#]} > 42
+OPTIMIZE_OBJECT_META_FILES ?= yes
+.endif
+
+.if !empty(STAGE_ROOT)
+STAGE_LIBDIR ?= ${STAGE_ROOT}${LIBDIR}
+
+all:   stage_libs
+stage_libs: ${_LIBS}
+.endif
+
 .if !target(install)

 .if defined(PRECIOUSLIB)
```

and `meta.stage.mk` (included from `bsd.sys.mk`) does the rest.

auto dependencies

Hooking `meta.autodep.mk` is simple too:

```
Index: share/mk/bsd.dep.mk
===================================================================
--- share/mk/bsd.dep.mk (revision 219256)
+++ share/mk/bsd.dep.mk (working copy)
@@ -119,6 +119,10 @@
 .endfor
 .endif

+.if ${MKDEP:Uno:M*auto*} != ""
+.include <${MKDEP}.mk>
+.else
+
 .if !target(depend)
 .if defined(SRCS)
 depend: beforedepend ${DEPENDFILE} afterdepend
@@ -172,6 +176,8 @@
 .endif
 .endif

+.endif
+
 .if !target(cleandepend)
 cleandepend:
 .if defined(SRCS)
```

other `bsd.*.mk` changes

We also tweak some others:

```
Index: share/mk/bsd.subdir.mk
===================================================================
--- share/mk/bsd.subdir.mk      (revision 219256)
+++ share/mk/bsd.subdir.mk      (working copy)
@@ -31,6 +31,12 @@

 .include <bsd.init.mk>

+.if ${.MAKE.LEVEL} == 0 && ${.MAKE.MODE:Mmeta*} != "" && !empty(SUBDIR)
+.if "${SUBDIR}" == "@auto"
+SUBDIR = ${:!echo ${.CURDIR}/*/Makefile!:H:T:O:N\*}
+.endif
+.include <meta.subdir.mk>
+.else
 DISTRIBUTION?= base
 .if !target(distribute)
 distribute:
@@ -92,3 +98,5 @@
 install: beforeinstall realinstall afterinstall
 .ORDER: beforeinstall realinstall afterinstall
 .endif
+
+.endif
Index: share/mk/bsd.libnames.mk
===================================================================
--- share/mk/bsd.libnames.mk    (revision 219256)
+++ share/mk/bsd.libnames.mk    (working copy)
@@ -7,6 +7,7 @@
 .if !target(__<bsd.init.mk>__)
 .error bsd.libnames.mk cannot be included directly.
```

```
 .endif
+.-include <local.libnames.mk>

 LIBCRT0?=      ${DESTDIR}${LIBDIR}/crt0.o

Index: share/mk/bsd.sys.mk
===================================================================
--- share/mk/bsd.sys.mk (revision 219256)
+++ share/mk/bsd.sys.mk (working copy)
@@ -89,3 +89,34 @@

 # Allow user-specified additional warning flags
 CFLAGS         +=      ${CWARNFLAGS}
+
+.if !empty(CLEANFILES)
+.NOPATH: ${CLEANFILES}
+.endif
+
+CFLAGS += ${CFLAGS_LAST}
+CXXFLAGS += ${CXXFLAGS_LAST}
+
+# from netbsd's bsd.own.mk
+PHONY_NOTMAIN = all clean cleandir depend dependall distclean includes \
+                install lint obj regress tags beforedepend afterdepend \
+                beforeinstall afterinstall realinstall realdepend realall \
+                html subdir-all subdir-install subdir-depend
+.PHONY:         ${PHONY_NOTMAIN}
+.NOTMAIN:       ${PHONY_NOTMAIN}
+
+# handy for debugging
+.SUFFIXES:  .S .c .cc .cpp .cpp-out
+
+
+.S.cpp-out .c.cpp-out:
+       @${CC} -E ${CFLAGS} ${.IMPSRC} | grep -v '^[[:space:]]*$$'
+
+.cc.cpp-out:
+       @${CXX} -E ${CXXFLAGS} ${.IMPSRC} | grep -v '^[[:space:]]*$$'
+
+.if !empty(STAGE_ROOT)
+.if !empty(STAGE_SETS)
+.include <meta.stage.mk>
+.endif
+.endif
```

Debugging

Being able to debug the build - when things appear not to be behaving as expected, is very handy.

We can use `-dM` to see why `bmake` thinks a target is considered out of date.

For example, to see what happens if a build command changes, we can edit a `.meta` file and subsitute `-O` for `-O2`:

```
$ mk -dM -DNO_DIRDEPS
Skipping meta for /c/sjg/work/FreeBSD/current/src/bin/sh.i386: .MAKE
Checking /c/sjg/work/FreeBSD/current/src/bin/sh for i386 ...
Skipping meta for objwarn: no commands
/c/sjg/work/FreeBSD/current/obj/i386/bin/sh/var.o.meta: 2: a build \
command has changed
cc -O -pipe  -nostdinc -DSHELL -I. \
-I/c/sjg/work/FreeBSD/current/src/bin/sh \
```

```
-std=gnu99 -fstack-protector -Wsystem-headers \
-Werror -Wall -Wno-format-y2k -Wno-uninitialized \
-Wno-pointer-sign \
-I/c/sjg/work/FreeBSD/current/obj/stage/i386/usr/include \
-I/usr/include -c /c/sjg/work/FreeBSD/current/src/bin/sh/var.c
vs
cc -O2 -pipe  -nostdinc -DSHELL -I. \
-I/c/sjg/work/FreeBSD/current/src/bin/sh \
-std=gnu99 -fstack-protector -Wsystem-headers \
-Werror -Wall -Wno-format-y2k -Wno-uninitialized \
-Wno-pointer-sign \
-I/c/sjg/work/FreeBSD/current/obj/stage/i386/usr/include \
-I/usr/include -c /c/sjg/work/FreeBSD/current/src/bin/sh/var.c
Building /c/sjg/work/FreeBSD/current/obj/i386/bin/sh/var.o
Building /c/sjg/work/FreeBSD/current/obj/i386/bin/sh/sh
```

There's actually quite a bit more noise, but the above is the interesting bit. Similarly, if we delete a command from the meta file:

```
$ mk -dM -DNO_DIRDEPS
Skipping meta for /c/sjg/work/FreeBSD/current/src/bin/sh.i386: .MAKE
Checking /c/sjg/work/FreeBSD/current/src/bin/sh for i386 ...
Skipping meta for objwarn: no commands
/c/sjg/work/FreeBSD/current/obj/i386/bin/sh/var.o.meta: 3: \
there are extra build commands now that weren't in the meta data file
Building /c/sjg/work/FreeBSD/current/obj/i386/bin/sh/var.o
Building /c/sjg/work/FreeBSD/current/obj/i386/bin/sh/sh
```

You get the idea.

Apart from debug info from `bmake` itself, we can also get it from `dirdeps.mk`:

```
$ mk DEBUG_DIRDEPS='*/lib*'
bmake-20110327: "/c/sjg/work/FreeBSD/current/src/mk/dirdeps.mk" \
line 220: lib/libc.i386: DIRDEPS='lib/msun  sys/i386/include  \
sys/sys  sys/x86/include stage'
bmake-20110327: "/c/sjg/work/FreeBSD/current/src/mk/dirdeps.mk" \
line 221: lib/libc.i386: _machines='i386'
bmake-20110327: "/c/sjg/work/FreeBSD/current/src/mk/dirdeps.mk" \
line 242: depdirs=/c/sjg/work/FreeBSD/current/src/lib/msun \
/c/sjg/work/FreeBSD/current/src/stage \
/c/sjg/work/FreeBSD/current/src/sys/i386/include \
/c/sjg/work/FreeBSD/current/src/sys/sys \
/c/sjg/work/FreeBSD/current/src/sys/x86/include
bmake-20110327: "/c/sjg/work/FreeBSD/current/src/mk/dirdeps.mk" \
line 243: qualified=
bmake-20110327: "/c/sjg/work/FreeBSD/current/src/mk/dirdeps.mk" \
line 244: unqualified=/c/sjg/work/FreeBSD/current/src/lib/msun \
/c/sjg/work/FreeBSD/current/src/stage \
/c/sjg/work/FreeBSD/current/src/sys/i386/include \
/c/sjg/work/FreeBSD/current/src/sys/sys \
/c/sjg/work/FreeBSD/current/src/sys/x86/include
bmake-20110327: "/c/sjg/work/FreeBSD/current/src/mk/dirdeps.mk" \
line 275: lib/libc.i386: /c/sjg/work/FreeBSD/current/src/lib/msun.i386 \
/c/sjg/work/FreeBSD/current/src/stage.i386 \
/c/sjg/work/FreeBSD/current/src/sys/i386/include.i386 \
/c/sjg/work/FreeBSD/current/src/sys/sys.i386 \
/c/sjg/work/FreeBSD/current/src/sys/x86/include.i386
bmake-20110327: "/c/sjg/work/FreeBSD/current/src/mk/dirdeps.mk" \
line 288: lib/libc.i386: /c/sjg/work/FreeBSD/current/src/lib/msun.i386 \
/c/sjg/work/FreeBSD/current/src/stage.i386 \
/c/sjg/work/FreeBSD/current/src/sys/i386/include.i386 \
/c/sjg/work/FreeBSD/current/src/sys/sys.i386 \
/c/sjg/work/FreeBSD/current/src/sys/x86/include.i386
bmake-20110327: "/c/sjg/work/FreeBSD/current/src/mk/dirdeps.mk" \
```

```
line 299: Looking for /c/sjg/work/FreeBSD/current/src/lib/msun/Makefile.depend.i386
bmake-20110327: "/c/sjg/work/FreeBSD/current/src/mk/dirdeps.mk" \
line 220: lib/libedit.i386: DIRDEPS='include  lib/ncurses/ncursesw  \
sys/i386/include  sys/sys  sys/x86/include stage'
bmake-20110327: "/c/sjg/work/FreeBSD/current/src/mk/dirdeps.mk" \
line 221: lib/libedit.i386: _machines='i386'
bmake-20110327: "/c/sjg/work/FreeBSD/current/src/mk/dirdeps.mk" \
line 242: depdirs=/c/sjg/work/FreeBSD/current/src/include \
/c/sjg/work/FreeBSD/current/src/lib/ncurses/ncursesw \
/c/sjg/work/FreeBSD/current/src/stage \
/c/sjg/work/FreeBSD/current/src/sys/i386/include \
/c/sjg/work/FreeBSD/current/src/sys/sys \
/c/sjg/work/FreeBSD/current/src/sys/x86/include
...
Checking /c/sjg/work/FreeBSD/current/src/bin/sh for i386 ...
Building /c/sjg/work/FreeBSD/current/obj/i386/bin/sh/sh
Updating .depend: sh.meta
Checking /c/sjg/work/FreeBSD/current/src/bin/sh/Makefile.depend.i386: .depend
```

The `*/lib*` is matched against the `DEP_RELDIR` of the `Makefile.depend*` as they are processed, so `*` matches all. You can also add the keyword `depend` and `dirdeps.mk` will output:

```
bmake-20110327: "/c/sjg/work/FreeBSD/current/src/mk/dirdeps.mk" \
line 299: Looking for \
/c/sjg/work/FreeBSD/current/src/lib/msun/Makefile.depend.i386
```

for each `Makefile.depend*` it looks for.

And if all that isn't enough you can add extra magic to `local.dirdeps.mk`.

Finally, if you use my `sys.mk` you can do things like:

```
DEBUG_MAKE_FLAGS=-dM DEBUG_MAKE_DIRS='*/libc' mk
```

to have `-dM` turned on only in `libc` when `.MAKE.LEVEL > 0` and only after `sys.mk` has finished.

```
DEBUG_MAKE_FLAGS=-dM DEBUG_MAKE_SYS_DIRS='*/libc' mk
```

The same, except that `-dM` is turned on at the start of `sys.mk`.

```
DEBUG_MAKE_FLAGS0=-dc DEBUG_MAKE_SYS_DIRS0=$PWD \
DEBUG_MAKE_FLAGS=-dM DEBUG_MAKE_DIRS='*/libc' mk
```

turns on `-dc` at the start of `sys.mk` for `.MAKE.LEVEL 0` and turns on `-dM` after `sys.mk` in `libc`.

You could put similar magic in `local.sys.mk` of course.

Sparse tree

It is worth pointing out, that like the current Junos build, `dirdeps.mk` does not get upset if a directory it thinks is a pre-requisite is missing. It doesn't even comment on the fact.

This makes it easy to work with a sparse tree and what we call a backing sandbox. The idea is you may have a daily, weekly, whatever build of your branch, which you can leverage. For example I could have:

```
.ifdef PROG
LDADD+= -L${STAGE_ROOT}${LIBDIR}
```

```
.if !empty(SHARED_STAGE_ROOT)
LDADD+= -L${SHARED_STAGE_ROOT}${LIBDIR}
.endif
.endif
CFLAGS_LAST+= -I${SHARED_STAGE_ROOT}/usr/include
```

in an appropriate `*.mk`, and any headers/libs not in my `$SB` would be found in the shared stage tree.

Further, because the headers and libs in that shared stage tree have their `.dirdep` files containing the `RELDIR` to visit, I still harvest all the correct dependency information.

Conclusion

In many ways *meta* mode simply builds on the aspects of our build which have worked well.

At the same time, it provides us with a simple solution to some rather complex problems.

We expect others can benefit in the same way.

URLs:

```
http://www.crufty.net/help/sjg/bmake.htm
ftp://ftp.netbsd.org/pub/NetBSD/misc/sjg/bmake-20110505.tar.gz
ftp://ftp.netbsd.org/pub/NetBSD/misc/sjg/mk-20110505.tar.gz
```

Author: sjg@juniper.net
Revision: $Id: building-bsd.txt,v 1.20 2011/05/14 18:27:22 sjg Exp sjg $