Building BSD

in meta mode

Simon J. Gerraty

Juniper Networks, Inc.

BSDCan 2011

Imagine something very witty here

Agenda

Introduction

History

Desirable build features

Some issues

Meta mode

Building FreeBSD current

Introduction

- building BSD for multiple architectures, in a reliable and efficient manner.
- some lessons learned from evolution of Junos build.
    - produces 3 times the Gb/hour of FreeBSD `universe` build.
    - still plenty of room for improvement
- building with `bmake` in *meta* mode.
    - uses `.meta` file idea from John Birrell's `build` project for FreeBSD

Teaser

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 ...
[Creating objdir /c/sjg/work/FreeBSD/current/obj/i386/stage...]
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 ...
Checking /c/sjg/work/FreeBSD/current/src/include/rpcsvc for i386 ...
[Creating objdir /c/sjg/work/FreeBSD/current/obj/freebsd9-i386/usr.bin/rpcgen...]
Building /c/sjg/work/FreeBSD/current/obj/i386/include/.dirdep
...
```

it's hard to make a build log interesting.

Teaser cont...

```
Building /c/sjg/work/FreeBSD/current/obj/i386/lib/libc/stage_libs
Checking /c/sjg/work/FreeBSD/current/src/lib/libc/Makefile.depend.i386: .depend
Building /c/sjg/work/FreeBSD/current/obj/i386/bin/sh/parser.o
Building /c/sjg/work/FreeBSD/current/obj/i386/bin/sh/sh
Updating .depend: builtins.c.meta mkinit.o.meta mknodes.o.meta
Checking /c/sjg/work/FreeBSD/current/src/bin/sh/Makefile.depend.i386: .depend
       67.03 real       196.12 user       170.12 sys
```

Things to note:

- objdirs were created automatically
  - keeping objdirs in a separate tree facilitates cleaning
- no `make depend`
- everything ran in parallel, but in the correct order
- leaf dirs visited directly
- `Makefile.depend*`

A quick look at Makefile.depend

```
# 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
...
.endif
```

Some definitions

`.CURDIR`: the value returned by `getcwd(3)` when `make` first starts

`.OBJDIR`: the directory `make` is in when it starts building things

`MACHINE`: the specific machine or cpu that we are building for

`MACHINE_ARCH`: the architecture that matches `${MACHINE}`
    `mips` for any of `xlr`, `octeon`, ...

.OBJDIR

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
```

Automated .OBJDIR

With `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
```

more definitions

`SB`: names the directory where `mk` found `.sandbox-env`

`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}`.

`RELDIR`: relative path from `SRCTOP` to `.CURDIR`.

bmake modifiers

NetBSD's make has a plethora of variable modifiers, several come from OSF Development Environment (ODE):

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

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*

History

The traditional BSD build looked something like:

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

in some cases `make dependall` coalesced the `depend` and `all` steps.

Multiple tree walks, using `SUBDIR` to visit next layer. Originally necessary to kept memory footprint reasonable.

Top-level Makefile can be *big*.

The Junos build

Juniper routers typically have separate CPU's for control and data planes.

- control plane is basically BSD
- data plane is proprietary
  - where the cool ASICs are
  - until very recently had its own build (`gmake`)

Junos 4.x (2000)

Originally Junos was built as a few packages added to a stock FreeBSD 2.x install. An experimental build introduced:

- 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*
- use of `bmake`
- single src tree for entire system
  - ability to checkout and build subsets
  - no tree walks - visit leaf dirs directly based on dependencies

Junos 5.0 (2001)

Migration to FreeBSD 4.x and to ELF, build overhaul:

- headers included from their src dir (no `make includes`)
  - use `${SRC_lib*}/h` as location for public headers
  - `dpadd.mk` automatically adds correct `-I`'s
- libraries linked from their objdir (no `make install`)
  - `dpadd.mk` automatically adds correct `-L`'s
- software packaged as ISO images
- new simpler top-level makefiles
  - visit leaf dirs directly based on dependencies
- use `autodep.mk` (no `make depend`)

Junos 7.0 (2004)

Maintainability improvements

- new set of top-level makefiles
  - broken out into function units `depend`, `cvs`, `build` etc.
  - each component easier to understand
- objdir creation automated with `auto.obj.mk`
- automatically derrive the tree dependencies from manifest files
  - stored in CVS
  - leveraged for subset checkout
- backing sandbox support
- digitally sign packages

Junos 9.3 (2008)

The tree had grown considerably, as had the number of architectures supported.

- removed dependency information from CVS

  - generate dynamically per `${MACHINE}`
- each subtree can come from different repository/SCM

Today

Completed first stage of migration to *meta* mode.

- data plane `gmake` build converted to `bmake` in *meta* mode
  - better debugging
  - better parallelism
  - more accurate dependencies
- rest of build can also run in *meta* mode
  - old build used to bootstrap `Makefile.depend*`
  - bulk of the tree *just works*
  - some major makefiles need re-work to avoid circular dependencies

Desirable build features

Some features have proven beneficial over long term

- separating sources and objects
- automated objdir creation
- automated dependency collection
- directory based dependencies
- building in parallel
- captive toolchains

Separating sources and objects

- while some people may *like* their objects and src in the same directory we don't give them that option (any more ;-)

- default `${.CURDIR}/obj/` or `${.CURDIR}/obj.${MACHINE}/` insufficient

- `MAKEOBJDIRPREFIX` easy - but ugly

- `bmake` allows applying modifiers to `MAKEOBJDIR`

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

- `mk objlink` still handy but `./obj/` ignored by build

Separating src and objects cont.

Well defined `SRCTOP` and `OBJTOP` simplify things.

One can simply assert:

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

rather than guess (*wrongly*):

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

Automated dependency collection

- `autodep.mk` leverages `gcc -M*` to collect dependency information as a side effect of building.
  - uses `${.PREFIX}.d` so `.SUFFIX` rules work

    - newer `auto.dep.mk` uses `.d.${.TARGET}` to avoid contention
    - requires compiler support (eg. `gcc`)
- `bmake` automatically ignores stale dependencies read from `.depend`
    - [re]moved headers cause target out-of-date not failure

Directory based dependencies

Allow the top-level build to visit leaf dirs directly

- tree walks are expensive (especially on NFS)
    - may be impossible to adequately order the build steps without resorting to phases like `make includes` and `make libraries`.
- leverage `DPADD` information in makefiles.
    - with `SRCTOP` and `OBJTOP` it is easy to derive src dir from objdir, `dpadd.mk` does the work.

dpadd.mk

Given:

```
LIBFOO ?= ${OBJTOP}/lib/libfoo/libfoo.a
```

If `${LIBFOO}` is referenced in `DPADD`, `dpadd.mk` computes:

```
OBJ_libfoo = ${LIBFOO:H}
SRC_libfoo ?= ${OBJ_libfoo:S,${OBJTOP},${SRCTOP},}
.if exists(${SRC_libfoo}/h)
INCLUDES_libfoo ?= -I${SRC_libfoo}/h
.else
# all bets are off
INCLUDES_libfoo ?= -I${OBJ_libfoo} -I${SRC_libfoo}
.endif
```

dpadd.mk cont.

Since accurate dependencies in makefiles are key, we use `DPLIBS`:

```
DPLIBS += ${LIBFOO}
```

is equivalent to:

```
DPADD += ${LIBFOO}
LDADD += -lfoo -L${OBJ_libfoo}
```

If `${LIBFOO}` in any of `SRC_LIBS`, `DPADD` or `DPLIBS`:

```
CFLAGS += ${INCLUDES_libfoo}
```

dpadd.mk cont.

Gather tree dependencies by recursively visiting dirs doing:

```
$ mk -C bsd/usr.bin/login dpadd
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},}
```

Obviously this is expensive (a tree walk), but typically only done after updating the tree, editing makefiles or manifests.

Building in parallel

- there's no such thing as building too fast
- building in parallel soaks up otherwise wasted CPU
- going fast doesn't matter if the results are incorrect:

  ```
  # this works fine in compat mode
  # but likely not in jobs mode
  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>
  ```

Building in parallel cont.

First fix attempt might be:

```
# 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
```

take two:

```
# 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
```

Building in parallel cont.

This is more like it:

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

# specified dependencies consistent with those
# captured from gcc -M
parser.c:       parser.h

file1.o:        parser.h
```

- by default we do not run leaf makefiles in jobs mode
- can set `USE_JOBS=yes` in makefiles known to work
- will flip that with *meta* mode

Captive toolchains

- compilers and similar toolchains, used a lot, changed rarely
- we need to be able to reproduce a build many years later
- tools team qualify new compiler, post it, done
- build checks for toolchain changes
- NetBSD's build provides support for externally maintained cross-toolchains via `EXTERNAL_TOOLCHAIN`

Some issues

- Some ideas scale better than others
- Some hacks live too long
- Periodic overhauls needed
- Be prepared to revisit decisions

Too many -I's and -L's

- including headers from their src dir and linking libs from objdir solved a problem but
  - too much of a good thing can be bad
  - makes it more difficult to spot name conflicts
  - most `#include ""` usage is wrong
- using *meta* mode we can address the original problem differently
  - automated staging

Too much complexity

- Junos build has more than trippled in size since 7.0 when the current top-level makefiles introduced
- hybrid architectures add more inter-machine dependencies
- leaf makefiles still simple, but top-level complexity has increased

Too much complexity cont.

```
# 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

all: build_arch.i386 build_arch.mips

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

Too much complexity cont.

- `build_arch.*` can be easily missused:

  ```
  __build_arch.${MACHINE}:        lots-of-stuff

  # this works ok for building just some-thing or and-another
  ```

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

# the above causes problems for this
every-thing: ${ALL_MACHINE_LIST:%=build_arch.%}
```

- `dirdeps.mk` allows easy (and safe) mixing of directory and machine dependencies

Manual maintenance is unreliable

- not all C programmers are build geeks
- basic rules for writing leaf makefiles:

```
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.
```

- 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

- Junos build uses lots of hosttools; code generators etc.
- some built for host and target
- top-level is where `MACHINE` gets changed
- thus, some form of top-level build is almost always needed
- top-level build requires tree dependencies to be collected
- none of this is necessary with `dirdeps.mk`

Insufficient parallelism

- 8 years ago just running top-level makefiles in parallel consumed build servers
  - too easy to write leaf makefiles which don't work in parallel
  - default leaf makefiles to compat mode
- build machines have gotten much faster, but build is more complex
  - for packaging reasons, one machine can depend on products of another
- 15min load average is a useful clue

Introducing *Meta* Mode

- create a `.meta` file for each target
- `.meta` file collects information about the target
  - the expanded command line
  - command output
  - *interesting* system calls
- `.meta` files first introduced in John Birrell's `build` for FreeBSD.
  - leveraged `DTrace` to collect syscall data
  - we asked John to write a simple kernel module `filemon`
  - `build` required all new makefiles

Meta mode cont.

- `bmake` + `.meta` files allows easy transition

- Junos build has been *meta* mode capable for over a year with almost no changes
- converting data plane (`gmake`) build was first priority

Rationale

- aid automated capture of dependency information
  - help optimize build performace
  - improve build reliability
- optimizing build means
  - do as little as possible
  - do it in parallel
  - but do it correctly!
- *meta* mode helps all the above

avoid make depend

- saves a lot of time
- requires better makefiles for parallel building
  - capture *local* dependencies to `Makefile.depend` for clean tree build
- `filemon` works for all targets not just `gcc`
  - automatically catches toolchain changes

avoid unnecessary dependencies

- In *meta* mode, `bmake` can compare expanded commands to *know* if there is a change. Thus dependencies 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
  ```

  can be skipped.

- use `DPADD` to bootstrap `DIRDEPS`

- entries in `DPADD` but not `DIRDEPS` were unnecessary.

tree walks don't always cut it

- ideally, build tree in a single pass
- `bsd.subdir.mk` and walking tree is inefficient
  - may not be possible to express dependencies between leaf directories
  - need *phases* like `make includes`, `make depend`
- Junos build visits leaf dirs directly based on tree dependencies
- *meta* mode supports that, more efficiently and generically

Building in meta mode

- enabled by the word `meta` in `.MAKE.MODE` which can be set by makefile

- `meta.sys.mk` included by `sys.mk`, does:

  ```
  .if ${.MAKE.LEVEL} == 0
  # make sure dirdeps exists and do it first
  all: dirdeps .WAIT
  dirdeps:
  .endif
  META_MODE += meta verbose
  .MAKE.MODE ?= ${META_MODE}
  ```

Writing .meta files

- for each target, a `.meta` file called `${.TARGET}.meta` is created
- if target is `.PHONY`, `.MAKE` or `.SPECIAL` (eg. `.BEGIN`, `.END`, `.ERROR`), then a `.meta` file is not created unless the target is also flagged `.META`
- never created if target flagged `.NOMETA`
- skip `.meta` if `.OBJDIR` == `.CURDIR` and `curdirOk=yes` not in `.MAKE.MODE`
- if target not in `${.OBJDIR}`, replace all `/` with `_` in meta file name

Meta file content

- expanded command line(s), prefixed with `CMD`
- current directory prefixed with `CWD`
- target, prefixed with `TARGET`
- command output preceded by line `-- command output --`
  - this is useful for error handling
- syscall data collected from `filemon` preceded by line `-- filemon acquired metadata --`
- append the name of the `.meta` file to variables `.MAKE.META.CREATED` and `.MAKE.META.FILES`
- if *meta verbose* mode expand and print `.MAKE.META.PREFIX` which defaults to the full path of the target.

filemon

- kernel module replaces use of `DTrace`

- available in FreeBSD and NetBSD

- for each syscall, an entry of the form:

  ```
  tag pid data
  data is usually a pathname, 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
  ```

- `bmake` mainly interested in `C` `E` and `R` entries

Reading .meta files

- skipped if target already considered out-of-date
- use `-dM` to see why `bmake` thinks target out-of-date
- compare expanded commands
  - unless told not to (`.NOMETA_CMP`)
  - or commands use `${.OODATE}`
- compare mtime of files Read or Executed against target
- if generated file within `${.MAKE.META.BAILIWICK}` but outside `${.OBJDIR}` is missing, target is out-of-date

Performance

- lots of extra `stat(2)` calls
- nothing to be done is worst case

    - adds about 1 second to `libc`
    - incentive to avoid unnecessary `#include`
- otherwise comparable to using `autodep.mk`
- `meta2deps` currently a shell script
- when entire build runs in *meta* mode, expect significantly better parallelism

Error handling

- since 2001 *sisyphus* (a tindebox-like system) builds Junos, analyzes breaks, identifies cause and fingers the guilty
- currently uses wrapper scripts to re-run compiler to capture errors and dependencies
- *meta* mode makes this simpler
    - on failure `.ERROR_META_FILE` is set to path of failed `.meta` file
    - `.ERROR` target copies failed `.meta` file to `$SB/error/`
    - `.meta` file contains everything *sisyphus* needs for failure analysis

Error example

```
# 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 /usr/lib/libc.so.12
W 5175 /var/tmp//cceNCjUd.s
E 5436 /usr/libexec/cc1
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
R 5436 /usr/include/sys/cdefs.h
R 5436 /usr/include/machine/cdefs.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
```

Extracting dependencies

- `bmake` simply uses `.meta` files to better know when a target is out-of-date
- `bmake` tracks `.meta` files via `.MAKE.META.FILES` and `.MAKE.META.CREATED`
- allows makefiles such as `meta.autodep.mk` to post-process `.MAKE.META.FILES` to gather tree wide dependencies.
- this process is greatly simplified by keeping objdirs out of the src tree

post-processing meta files

```
# Meta data file /c/sjg/work/FreeBSD/current/obj/i386/bin/sh/var.o.meta
...
-- filemon acquired metadata --
...
E 16111 /bin/sh
```

```
...
R 16112 /c/sjg/work/FreeBSD/current/src/bin/sh/var.c
W 16113 var.o
R 16112 /c/sjg/work/FreeBSD/current/obj/stage/i386/usr/include/sys/cdefs.h
R 16112 /c/sjg/work/FreeBSD/current/obj/stage/i386/usr/include/unistd.h
...
R 16112 /c/sjg/work/FreeBSD/current/obj/stage/i386/usr/include/stddef.h
R 16112 /c/sjg/work/FreeBSD/current/src/bin/sh/expand.h
R 16112 ./nodes.h
```

- any file read or executed from an objdir other than `.OBJDIR` idendifies a directory which must be built before `.CURDIR`, (`DIRDEPS`)
- any file read from the the src tree outside of `.CURDIR` identifies a directory which must exist, (`SRC_DIRDEPS`)

mapping objdir to src dir

- when linking libraries from their objdir, the mapping to src dir is trivial:

  ```
  SRC_libfoo = ${OBJ_libfoo:S,${OBJTOP},${SRCTOP},}
  ```

- when using headers and libraries which have been *staged*, help is needed:

  ```
  $ cd /c/sjg/work/FreeBSD/current/obj/stage/i386/usr/include
  $ ls -l unistd.h*
  -rw-r--r--   2 sjg  wheel  18731 Mar  2 18:37 unistd.h
  -rw-r--r--  92 sjg  wheel     13 Apr  3 14:53 unistd.h.dirdep
  $ cat unistd.h.dirdep
  include.i386
  ```

  the `.dirdep` file contains the `DIRDEPS` entry needed.

Makefiles

- majority of leaf makefiles *just work*
- some minor changes to `bsd.*.mk`
- new makefiles `meta.*.mk`, `dirdeps.mk` and `gendirdeps.mk`
- top level makefiles can be very simple
- `Makefile.depend*` is most visible change

Makefile.depend

- collects `DIRDEPS`, `SRC_DIRDEPS` and local dependencies for each directory
- can be maintained in SCM
- use `Makefile.depend.${MACHINE}` if cross-building supported.

One build product per directory

- building multiple things is ok but
- each directory/makefile should do the same thing every time
- only collect dependencies when doing default target

Separate MACHINE independent activity

- this is an optimization (ie. optional)
- when cross building for lots of architectures
- doing code generation and building host tools once helps

meta.autodep.mk

- post-processing `.meta` files can be expensive, skip if possible
- if `.MAKE.META.CREATED` is not empty, we have work to do
- process `.MAKE.META.FILES`:

```
.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}' )
```

gendirdeps.mk

- runs `meta2deps.sh` to extract *interesting* directories
- things in `${SRCTOP}/*` are `SRC_DIRDEPS`
- things in `${OBJTOP}/*` are `DIRDEPS`
- things in objdirs other than `${OBJTOP}` (ie. build for other `${MACHINE}`) are qualified `DIRDEPS`.

meta.stage.mk

- links or copies files into *staging* locations
- puts `.dirdep` file next to each *staged* file, so mapping to src directory not lost
- multiple `STAGE_SETS` with own `STAGE_DIR`
- `STAGE_AS_SETS` for renaming while staging
- provides various simple targets `stage_incs`, `stage_libs`, `stage_symlinks` and generic `stage_files` and `stage_as_files`

dirdeps.mk

- deals with `DIRDEPS`
- only interesting to initial instance of `bmake` (`${.MAKE.LEVEL} == 0`)
- conceptually simple
    - initial `bmake` reads `${.CURDIR}/Makefile.depend.${MACHINE}` gets `DIRDEPS`
    - generate dependencies on each `${DIRDEP}` for `${DEP_RELDIR}`
    - process `Makefile.depend*` from each `${DIRDEP}`
    - repeat

dirdeps.mk cont.

Given:

```
DIRDEPS = lib/libc include ...
```

then (ignoring the complication of other machines):

```
# always qualified
_build_dirs := ${DIRDEPS:@d@${SRCTOP}/$d.${MACHINE}@}

${SRCTOP}/${DEP_RELDIR}.${MACHINE}: ${_build_dirs}

.for f in ${_build_dirs:@d@${d:R}/${.MAKE.DEPENDFILE:T}@}
.if ${.MAKE.MAKEFILES:M${f}} == ""
.-include <$f>
.endif
.endfor
```

Supressing DIRDEPS

Use `-DNO_DIRDEPS` to supress `DIRDEPS` outside of `.CURDIR`:

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

builds and runs all unit tests in that subtree without checking anything else.

meta.subdir.mk

- we do not tree walk
- may still want to launch a build in `src/usr.bin/`
- set initial `DIRDEPS` based on result of `find ${SUBDIR}` if no `Makefile.depend*` exists in `.CURDIR`

BUILD_AT_LEVEL0

- our data plane developers expect `mk` to DTRT regardless of target machine(s) appropriate to `.CURDIR`
- this can be simplified by never building anything in the 0th instance of `bmake`, so we set `BUILD_AT_LEVEL0 = no`
- `no` means sub-makes used to build `.CURDIR`
- `yes` means sub-makes only used to build `.CURDIR` for other machines

Building kernels

- BSD kernel build does not provide a src dir per kernel to capture dependencies

- `jnx.kernel.mk` lets us build kernels anywhere:

  ```
  # 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'.
  ```

Top-level makefiles?

Given a collection of directories `pkgs/*/` that contain little more than `Makefile.depend*`, the top-level makefile need be no more complex than:

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

.include <dirdeps.mk>

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

```
.endfor
```

Building FreeBSD current

- test case for generic `meta.*.mk` and `dirdeps.mk`
- want to be able to easily cross-build stock FreeBSD
- minimize changes to FreeBSD

Setup

Install mk-files in `$SB/src/mk/` and set `MAKESYSPATH=$SB/src/mk:$SB/src/share/mk` we use:

```
sys.mk
auto.obj.mk                generate objdirs automatically
obj.mk                     linked as bsd.obj.mk
meta.*.mk
dirdeps.mk
gendirdeps.mk
```

and some local additions:

```
sys/FreeBSD.mk             includes ../../share/mk/sys.mk
local.sys.mk               tweaks to blend everything
local.dirdeps.mk           enable staging
local.libnames.mk          link libs from stage tree
```

bmake vs FreeBSD make

- FreeBSD make has `:U` and `:L` modfiers that conflict but not used

- `bmake` requires explicit `.NOPATH` in some cases. Generally:

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

- also NetBSD's `bsd.own.mk` flags all standard targets as `.PHONY`

- `dirdeps.mk` requires lots of `bmake` features

Staging headers and libs

- like `make install` as you go
- no need to be `root`
- minor changes to `bsd.lib.mk`, `bsd.incs.mk` to leverage `meta.stage.mk`

Debugging

- `bmake -dM` will say why *meta* mode decides out-of-date

- `sys.mk` supports enabling make flags in certain dirs:

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

Sparse tree

- `dirdeps.mk` does not mind if a directory is missing
- makes it easy to re-use pre-built tree as *backing sb*
- Junos SDK leverages this

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
```

Questions

Q&A

Author: sjg@juniper.net

Revision: $Id: building-bsd-slides.txt,v 1.7 2011/05/05 18:08:43 sjg Exp sjg $