JBUILD – NEXT GENERATION BUILD TOOL FOR FREEBSD

Craig Rodrigues
rodrigc@juniper.net / rodrigc@FreeBSD.org
BSDCan, May 2010

OVERVIEW

- Brief introduction to build system in FreeBSD
- List some pain points in the current build system
- Describe the design and implementation of "jbuild" tool at Juniper to attempt to solve some of these issues

WHY DO PEOPLE LIKE TO BASE PRODUCTS ON BSD?

- BSD license works well for products
- Solid operating system
- Drivers available for lots of off the shelf hardware (storage, network, CPU, etc.)
- Availability of tools (toolchains, scripting languages, etc.)
- Easy to extend or customize due to source code availability, and also *build system makes it easy to add new extensions to codebase and integrate them*

FREEBSD BUILD SYSTEM

- **/usr/bin/make** is a utility in FreeBSD that is used to build software
- **make** checks dependencies, and executes actions if dependencies are out of date
- in FreeBSD, **make** comes with a set of “makefile infrastructure” files in /usr/share/mk, which contain common rules for building programs (bsd.prog.mk), libraries (bsd.lib.mk). Other makefile infrastructure exists for compiling kernels
- FreeBSD ports follows similar conventions, and has its “makefile infrastructure” in /usr/ports

EXAMPLE: MAKEFILE FOR LIBRT

```
LIB=rt
SHLIB_MAJOR= 1
CFLAGS+=-I${.CURDIR}/../libc/include -I${.CURDIR}
CFLAGS+=-Winline -Wall -g
SRCS+= aio.c mq.c sigev_thread.c timer.c


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

- BSD convention is to have average Makefiles used by users be very simple, only defining variables
- bsd.lib.mk contains the logic for what rules to execute when dependencies are out of data for compiling **librt**
- use of “makefile infrastucture” makes it easy to add new libraries

COMMON USES OF FREEBSD BUILD SYSTEM

- **make buildworld, make installworld**
  - Rebuild all of the userland from sources in /usr/src, and install them
- **make buildkernel, make installkernel**
  - Rebuild kernel from sources in /usr/src, and install it
- This is commonly used by more advanced users who are comfortable with upgrading a system by rebuilding it from source

OTHER INTERESTING BUILD TARGETS

- **make universe**
  - Builds world and kernel on all possible architectures (amd64 arm i386 ia64 pc98 powerpc sparc64 sun4v)
  - FreeBSD Developers are supposed to use this to make sure they don't break other architectures
- **make release**
  - Used to rebuild everything from sources, and then end up with installation media (CD/DVD)
  - FreeBSD release engineering team uses this to create installation media available from FreeBSD.org ftp and web sites

WHAT ARE SOME PAIN POINTS WITH THE CURRENT SYSTEM?

- FreeBSD build system works quite well, but there are some problems:
  1. **make universe** takes a long time, so few developers use it. Less commonly used architectures break, and are only noticed by tinderbox systems
  2. Parallel builds (make –j [n]) of buildworld/buildkernel, are not so reliable, and are not the default. This prevents optimal use of multi-cpu/multi-core systems and distributed systems for building.
  3. Expressing more complex build dependencies requires a lot of knowledge of build systems. For example, an IDL compiler can generate .c and .h files, which can then be built into a library. This library and all things which link against this library now depend on the IDL compiler.
- Problems manifest as build breaks, or very long build times.

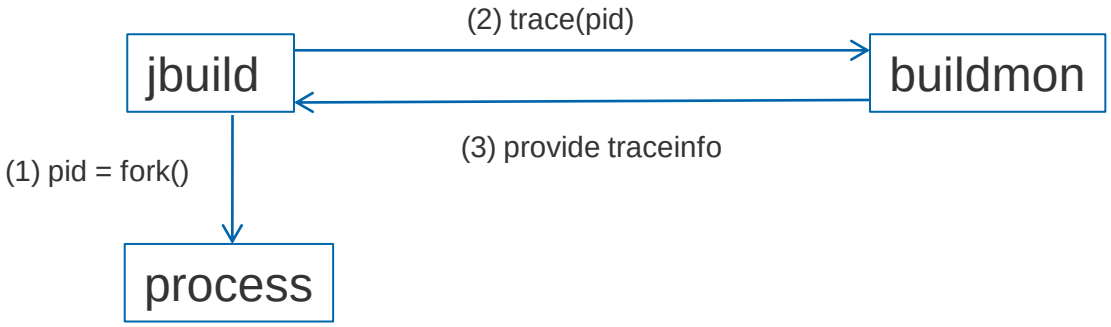
IDEAS FOR INCREMENTAL IMPROVEMENTS TO MAKE

- Pain points in build are annoying for FreeBSD but tolerable.
- At Juniper, due to the size of the codebase, these problems are not so tolerable.
- John Birrell had worked on build systems before, and had some ideas to improve things at Juniper by making incremental improvements in make:
  - As **make** forks off processes during the build, the file accesses of these processes could be traced. Makefile needs to specify **explicit dependencies** (these .c files make up this library), but more complex **implicit dependencies** can be captured by the tool and used during the build.
  - “makefile infrastructure” for better handling dependencies between directories could be written, and help with improved parallel builds

WHAT'S IN A NAME: JBUILD?

- Proof of concept project started in 2008
- Idea was to make modifications to make, but compile two binaries from the same codebase, but use #ifdef preprocessor macros to enable different logic in the binaries
  1. **make** – binary would behave like existing make and not contain new behaviors
  2. **build** – binary would contain new behaviors, and read Buildfile, and ignore Makefile. Buildfile syntax is identical to Makefile, but would allow two build systems to coexist. This would allow for more flexibility of prototyping and testing
- **build** conflicted with another internal Juniper tool, so **jbuild** was chosen

PROTOTYPE 1: JBUILD + DTRACE



- As jbuild forked processes, the pid of these processes was passed to a buildmon server over IPC
- Buildmon then ran a DTrace pid provider to trace all file accesses done by the process. Separate server required because DTrace pid provider needed elevated privileges.
- Tracing information was provided by buildmon to jbuild
- As forked processes exited, information would be written out

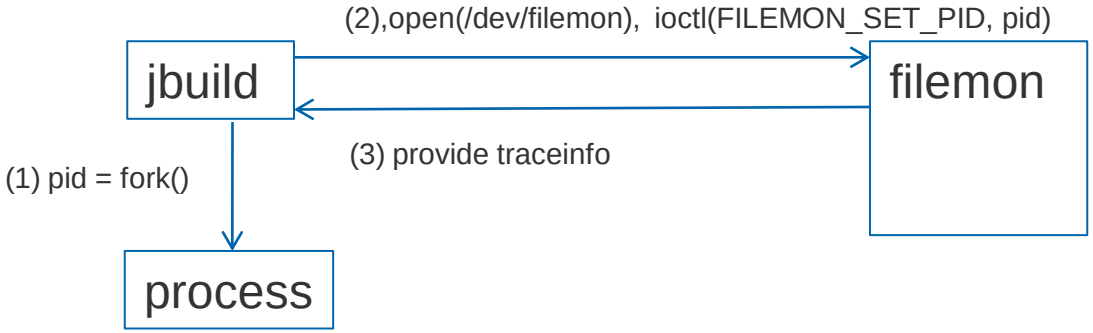
PROTOTYPE 1: ANALYSIS

- Tracing information was useful, and captured implicit dependencies quite well.
- Update builds were very reliable, and would only build things that changed.
- Under load, on FreeBSD 7, i386, the extensive use of DTrace pid providers seemed to use lots of kernel memory. This led to serious performance problems. At the time, we saw on mailing lists similar reports of high kernel memory usage with ZFS on i386 FreeBSD. John suspected that since DTrace and ZFS both came from Sun, the way the code was written to use kernel memory was an issue
- Using a buildmon server posed problems (single point of failure, possible security issues, potential problems with multiple users)

PROTOTYPE 2: JBUILD + FILEMON KERNEL MODULE

- Tracing information from first prototype was useful, but overhead of DTrace pid provider under load, plus problems with separate buildmon server didn't seem worth it
- filemon kernel module was written which implemented a clone device which wrapped certain file system related syscalls (open, read, write)
- Wrapper would log file syscall, then delegate to the real implementation of the syscall
- This logging would replace the DTrace pid provider

PROTOTYPE 2: FILEMON KERNEL MODULE



- As jbuild forks processes, it opens a descriptor on the /dev/filemon device, and passes the pid to the device
- Syscalls of interest are logged, and tracing information is provided back to jbuild
- jbuild uses this tracing information as part of dependency checking
- “jbuild –Q” turns off accessing /dev/filemon….we lose the tracing of implicit dependencies, and jbuild behaves more like regular make which depends on specification of explicit dependencies

PROTOTYPE 2: ANALYSIS

- filemon kernel module was very stable, and is what we currently use
- Code is simple, and only traces a few syscalls of interest (file system related ones, like open, read, write)
- Format of log file is the same as what DTrace pid provider gave, allowing us to reuse the same code in jbuild for parsing the log file

FILEMON OUTPUT

- For each target, built under obj, a log file is created, with a .meta extension
- For example, src/lib/liby/main.c is compiled to obj/i386/lib/liby/main.o, and obj/i386/lib/liby/main.o.meta is created

DIRECTORY DEPENDENCIES AND PARALLEL BUILDS

- to allow a build to be more parallel (higher –j flags), we follow the convention where one directory builds one final product, and then we track dependencies between the directories
- For example:
  - **src/lib/libfoo/Buildfile** builds l**ibfoo.a** library
  - **src/usr.bin/foo/Buildfile** builds **foo** binary, and links against libfoo.a
- src/usr.bin/foo has a “directory dependency” on src/lib/libfoo, since the library needs to be built before the binary
- Getting these directory dependencies right gets difficult in larger build systems, but are very important for making a build more parallel

DIRECTORY DEPENDENCIES: JDIRDEP

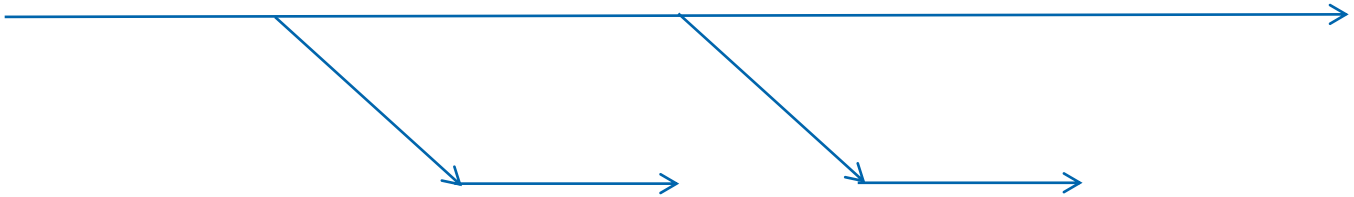
- As targets are being built, log files from filemon will be written
- As the jbuild process exits, the log files which were just created are parsed by jdirdep (a utility which is part of jbuild)
- jdirdep looks for files that were accessed during the build
- If files accessed are in the obj tree, then a variable called DIRDEP is updated. The DIRDEP variable contains a list of directories that can be built *before* this directory. The directories are relative to the top-level src directory
- Buildfile.dep is a file which is written out which contains the DIRDEP variable
- A target in the “makefile infrastructure” called updatedirdep is then called by jbuild, and another file called Buildfile.dirdep is written out which has logic for determining what Buildfiles can be built before the current Buildfile

DIRECTORY DEPENDENCIES: 2-LEVEL SUB-MAKE PROCESSES

- In our existing build system, we try to restrict the level of make processes to 2
- The level 1 jbuild process reads in sys.mk and the default “makefile infrastructure” . If a Buildfile.dirdep does not exist in the current directory, a default one is created. The Buildfile.dirdep is then read, which then possibly includes many other Buildfile.dirdep files
- The level 1 jbuild process gets a quick view of all the directory dependencies
- At the end of the Buildfile.dirdep, “jbuild all” is invoked. This is the second-level jbuild process, which actually does the build.
- Having more than two levels of sub-make processes can cause race conditions under high levels of parallel builds

DIRECTORY DEPENDENCIES: 2-LEVEL SUB-MAKE PROCESSES

Level 1 jbuild process: reads Buildfile.dirdep files, then calls “jbuild all” for for each Buildfile corresponding to a Buildfile.dirdep



Level 2 jbuild process: builds the all target. Does the actual building of the target, and writing out of .meta files.
When level 2 process exits, the atexit() handler parses the .meta files created, and does jdirdep processing to update directory dependencies

TESTING THAT A DIRECTORY BUILDS ON ALL TARGETS

- cd src/lib/libfoo
  jbuild –DALLMACHINES
- This is much faster than “make universe”, because we capture the directory dependencies at a finer level.

TRYING IT OUT

1) Check out jbuild project branch from FreeBSD svn:

   svn co svn://svn.freebsd.org/base/projects/jbuild src

2) Build and load filemon.ko :

   cd src/usr.bin/jbuild/filemon
   make
   kldload filemon.ko

3) Build jbuild and jdirdep:

   cd src/usr.bin/jbuild ; make
   cd src/usr.bin/jdirdep; make

4) Copy jbuild and jdirdep to somewhere in your path.

5) cd src/lib/liby
   jbuild

FUTURE DIRECTIONS

- filemon functionality (“meta-mode”) is being rolled into NetBSD’s bmake by Simon Gerraty <sjg@juniper.net> who is the NetBSD bmake maintainer.
- Functionality being rolled into NetBSD bmake is cleaner implementation than what was done for jbuild, but principles are the same.
- Being able to capture tracing information of actual files touched during build is extremely useful for determining accurate dependencies.

CONCLUSIONS

- The FreeBSD build system is well established and extensible. Many people make products derived from FreeBSD, and hook into the existing build system.
- Some of the pain points in the build system can be alleviated by incremental modifications to the make tool, and by writing new “makefile infrastructure” to handle directory dependencies, and a 2-level sub-make model.

IN MEMORIAL

- John Birrell, Nov. 20, 2009
- jb@FreeBSD.org
  jbirrell@juniper.net

http://lists.freebsd.org/pipermail/freebsd-announce/2009-November/001284.html