Package manager

Whilst building a mkduring searching for such a tool a new thread in harvey-os popped up titled Why not make?. TL;DR: harvey team ditched [[http://plan9.bell-labs.com/sys/doc/mk.html) -- which is what you had to use to build plan9 originally -- for a custom build script written in go. Mostly because go and plan9 share ancestry and most harvey contributors are well versed in go.

Why not make?

I don't think I can elobarate this further then it already has been in that thread, but I'll try anyways, something Aki said really struct a chord

Today, Go is the build system for Harvey, with some stuff pushed to json for convenience. For all I care, build.go should be seen as malleable as the json it reads is. Or maybe the json should be viewed as struct literals split out from build.go. Either way, I prefer to think of go as the single fully general underlying system, with build.go being the equivalent of a Makefile.

and with that, and what ron had said

Build is fast and can be faster also, build can let us do some interesting things not yet done. For example, we could gather all the .c files for the kernel into one file, compile that, and see if we get benefit from whole program optimization. Easy to write this in Go, hard to do in shell scripts (I've tried) since we want to correctly order the inclusion of files.

(Indeed, on my lenovo x201, when I time a build I get

```
106.39s user 19.74s system 92cpu 2:15.88 total
```

so there is still room for improvement, infact that was my main motivation for doing this.)

{Java, Java} - Choose two

So I started looking at build systems, state of the art seems to be Buck, Bazel which seem to be built by ex or current googlers. Installing buck and bazel are a nightmare since you have to install java -- you have to go to oracle's website and download a version of jdk like its 1995 -- ant and a bunch of other dependencies, so there is no brewinstall for buck or bazel there is a **AUR** package for bazel but that seems to be it from my perspective since I only use OSX and archlinux. (I don't know what the story is on ubuntu or debian and I couldn't care less because all I care about as a dependency is go and all harvey devs will have go installed.)

As Álvaro said,

Before build.go we thought in other methods. One of them recent opensourced by Google, but it requieres java. Java is very expensive for now if we want build harvey from harvey before Christmas. I also wanted to be able to make this extensible

as the section Adding Custom Rules to Buckstates

As of the writing of this document, the only official way to add rules to Buck is to fork the project and modify the source. We will, at some point, construct a beautiful and elegant extensions API. Until then....

meanwhile bazel atleast has an extension mechanism called skylark but the main code is still java so who cares. Ok I shouldn't probably give so much grief to buck for the extensibility story, because thats forking is how you would extend build as well. Although the extension writing process is immensely easier as it shall be apparent later in this post.

Enter build

So I started hacking on this thing called build, incidentally the same name given to the harvey build script.

Installing it is as simple as go getting sevki.org/build other reason, go getable bazel or buck is plenty good to be doing this.

But there are other motivations behind it, for starters I wanted the build tool to have a web interface, not that it isn't a CLI or requires to be used via a browser, but especially build-bots would benefit alot from being able to know what the dependency graph looks like.

for instance, building a harvey is as simple as

```
build //:harvey
```

to build harvey. One thing build assumes is that you are always in a git repo, so **//** is where your **.git** lives, and everything is relative to that. If you are in a submodule, we assume that you are doing stuff related to that module and assume that that submodule also has **BUILD** files with targets, and source file mappings relative to that folders **//**

To run as a server you add **server** to the command like so

```
build server
```

(I haven't nailed down the server functionality yet, how the build bot server should be configured, how timeouts should work on a target compliation level, so on and so forth, which is why this isn't public yet) and build runs in server mode which is when things start to become interesting. There are two view modes as of writing this,

depGraph mode and **buildGraph** mode; **depGrpah** hashes the target name and does some cutesy coloring for differentiating of targets, **buildGraph** is what is produced after a build, and thats when things start to get interesting, for instance take harvey travis build log, infact I'll actually spare you the navigating from

here

Image

I know this is a cheap shot but you can't really tell what which of the targets are acting up and for what reason from looking at that image even if it were dockerand [[why-not-make/docker-buildLog.png) aren't doing much better either. On the other hand if you very clumsily stare at the graphbelow, you'll see that **syscallheader** caused a chain reaction for all the targets that depend on it to fail.

Page

Extending

Let's take mksys for instance, the entire application hasn't really changed that much but I'll quickly go trough the changed bits, previous build file looked like

```
{
        "Library": "klibc.a",
        "Name": "KernelLibc",
        "Pre": [
                "mksys -o sys.h -mode=sys.h $HARVEY/sys/src/sysconf.json",
                "mksys -o . -mode=syscallfiles $HARVEY/sys/src/sysconf.json"
        ]
}
```

we moved the mksys flags to the build definition by declaring a struct like so

```
type MkSys struct {
        Name         string   `mk_sys:"name"`
        Mode         string   `mk_sys:"mode"`
        ARCH         string   `mk_sys:"arch"`
        OutPath      string   `mk_sys:"out" build:"path"`
        SysConf      string   `mk_sys:"sysconf" build:"path"`
        Dependencies []string `mk_sys:"deps"`
        Out          []byte
        Source       string
}
```

simple enough. The end [[why-not-make/mksys.go][result]] is, not extremely different from its json couterpart, but its looks bazely.

```
mk_sys(
        name = "syscallheader",
        mode = "sys.h",
        arch = "amd64",
        sysconf = "//sys/src/sysconf.json",
        out = "//sys/src/libc/9syscall/sys.h"
)

mk_sys(
        name = "syscallfiles",
        mode = "syscallfiles",
        arch = "amd64",
        sysconf = "//sys/src/sysconf.json",
        out = "//sys/src/libc/9syscall",
        deps = [
             "syscallheader"
             ]
)
```

In order to be a target type MkSys needs to implement the build target interface, which means that it should have a couple of functions,

```
// Target is for implementing different build targets.
type Target interface {
        Build() error
        GetName() string
        GetDependencies() []string
        GetSource() string
        Reader() io.Reader
        Hash() []byte
}
```

Build() , obviously builds the target, everything else that starts with get are convinience methods, **Reader()** returns the output log reader, it could be a log file, it could be a byte buffer, what ever you like, and **Hash()** which is for caching.

Caching

Harvey, builds really fast, no question about that, and will harvey benefit from caching, probably not to the extent that it will become a problem for a very long while, but if something is worth porting, it's worth overdoing. And while Ronhas his reservations about how gnu make handles it, which as far as I can gather is by file modtimes, I think this is not a particularly hard problem to solve, build tries to fix it by hashing everything underthe sun, files, arguments for targets hashes of dependencies, and even hashes the CC version, while it isn't as cheap as stat() ing the file, it is the most effective way I could think of to assure correctness of builds, whilst increasing the speed still dramatically.

```
build //:harvey > /dev/null  0.10s user 0.07s system 111cpu 0.149 total
```

This is the time it takes to hash the entire file tree, variables and even the **CC**--verison

Concurrency

All target builds are executed in their own go routines. So one should not assume serial, execution of dependencies, they almost always will be in randomium ("mium" is latin for not really) order, if a target has to be executed before another target then by definition it's a dependency hence it goes in its dependency pile.

Clearly the mechanism for concurrency should be bound by the ammount of cpu power you have, there is also no reason that the workers should only be distributed to your machines CPUs they should also be distrubted to a cluster machines in a data center, so that is something I'm looking to implement in build.

Beyond harvey

Caching and paralelism is thrown in to this project not because there is a real need for it in harvey but because I think everyone can benefit from build, there is probably a case to be made for using build to build docker images, vendoring go packages and so on and so forth. Of bazel, buck and pants, only pants has support of go packages, and I feel uneasy about that, building go python doesn't feel right. Meanwhile it would be trivial integrate the already great tools like godepinto build, I think any go developer could do it in less than a day.