SEE TEXT ONLY

GETTING THE JOB DONE

The init System Debate RE LAUNCHED

By Mark Heily

Over time, Unix operating systems have evolved a variety of mechanisms to launch programs and influence their execution. To put this into perspective, let's take a quick tour of the different kinds of programs that run on a typical FreeBSD system.

When the computer first boots up, the `rc(8)` mechanism launches programs called "services" that can perform one-time system initialization tasks, or execute daemons to run in the background. These services can also be stopped and started while the computer is running.

Every minute the computer is running, the `cron(8)` daemon wakes up and launches scheduled tasks. Cron also supports the ability to run `at(1)` jobs after a certain time interval has passed. It can also run `batch(1)` jobs when the system load average falls below a certain threshold.

There are many other mechanisms for launching programs. `Inetd(8)` launches programs when an incoming socket is created. `nice(1)` runs programs with a modified priority level. `chroot(8)` launches programs with an alternate root directory. `service(8)` launches programs with a "supervisor" process that can restart the program if it crashes. Jail managers like `jail(8)` launch programs within a jailed environment. Virtual machine managers like `iohyve(8)` launch programs that execute a

bhyve virtual machine. The `cloudabi-run(1)` command launches programs with a specific set of file descriptors pre-allocated.

Each of the mechanisms described above has its own configuration file format, its own particular style of usage, and essentially lives in its own isolated world. What if I told you there was a common underlying theme behind all of these different launchers? What if it were possible to have a single "launch all the things" mechanism that could become a common foundation for many of the operating system tools we enjoy today, as well as tools of the future?

I've started to explore these questions in greater detail, and have some interesting results to share. I've produced a working implementation of a new system called "the `jobd` job framework," or just "`jobd`" for short.

Overview of the Job Framework

`jobd` is a mechanism for launching and monitoring "jobs" that run within a single operating system instance. Jobs can be thought of as 1) some amount of work to be done, 2) the dependencies and preconditions before work can be started, 3) various ways to observe and interact with the program doing the work, and 4) a set of cleanup actions to perform after the work is complete.

That's quite a mouthful, so let's break it down into smaller chunks. We should start by going over the important concepts and common terms used in the job framework.

A job is defined by a JSON configuration file called a "manifest." The manifest contains a combination of a program, an execution context, properties, dependencies, and resources. These terms have specific meanings that are discussed below.

Within a job, the "program" is literally the path to the executable and the ARGV argument vector passed to the `execl(3)` system call. This begins the execution of a new program image, right after `jobd` calls `fork(2)` to create a new process.

The job's "execution context" represents all the changes to the child process that occur after the `fork()` call, but prior to the `exec()` call. Examples include: setting the user-id and group-id, calling `chroot(2)`, setting resource limits, setting the umask and nice value, and so forth.

System administrators need the flexibility to customize various aspects of how the job runs. For example, they might want to change the port number that a network daemon listens on. Job "properties" are the mechanism that allows for this control.

Job properties are simple key/value strings that can be substituted inside the manifest. Properties can also be queried directly, using library calls or the `jobcfg(1)` command. Taking it one step further, properties can be used to generate application-specific configuration files using a template; for example, the main `httpd.conf file` used by Apache.

Jobs can have "dependencies" that determine when the job should be stopped and started. Jobs can be started on-demand, on a fixed timer interval, at certain calendar dates/times, or when manually enabled by a system administrator. Jobs can also depend on the status of other jobs, so you can start and stop multiple jobs in a certain order.

Jobs can have "resources" which represent external things that are automatically created before the job starts, and are automatically destroyed when the job terminates. The job assumes ownership of the resources. This is most useful in combination with jails, as it allows you to create a jail on the fly to run a job.

We've covered the main aspects of what constitutes a "job" as far as the job framework is concerned. If you think back to the list of launchers in the first part of this article, there isn't a one-to-one mapping between what `jobd` offers and any of the existing mechanism. `Jobd` is very similar to `rc(8)` in that it supports all of the things you would need to manage services, but it isn't gravitationally bound to `init(1)` and the boot/shutdown process. It is more of an automaton, constantly running in the background and

Concept	Meaning	Example
Program	What to run	/usr/local/sbin/httpd
Context	How to run the program	setuid(2) to 'httpd'
Property	Information to tell the program	Listen on port 80
Method	A command the user can invoke to interact with the program	Run 'jobctl httpd status' to display server status information
Resource	Something the program needs before it runs	Install the 'httpd' package from the ports tree
Dependency	When the program should run	Whenever a client connects to port 80

reacting to events and conditions by starting, stopping, or restarting programs.

Now that the scope of the job framework is clear, it is worth mentioning a few things that are explicitly not in scope for the project. Unlike most of the init system replacements that have come out in the last few decades, `jobd(8)` is not trying to replace `init(8)` or usurp the role of `pid #1`. It will not handle early-boot tasks like setting up the console and `pttys`, or mounting filesystems listed in `/etc/fstab`. If you boot into single-user mode, I expect that this will still be handled by `init(8)` and `rc(8)` in much the same way as it is done now.

Despite the flexibility of the job framework to cover a wide range of use cases, I still would like to see it used first and foremost as a replacement for `rc(8)` and the current FreeBSD `init` system. This is a position that is likely to reawaken a lively debate among the community, as there are passionate defenders of the status quo. Nonetheless, it is a debate worth having again, in light of the potential benefits of adopting the job framework.

Some will ask, "Why change? Shell scripts have served us well since the beginning of BSD." I can't argue with the historical truth of that statement, but I will point out that the landscape of modern computing is far different from what it was during the early days of Unix. New problems demand new solutions.

Here are some reasons why I think that relying on a collection of shell scripts to power the init system is holding us back:

- The `rc(8)` system was not designed to run in the background and react to events and conditions that call for taking action. It runs once at boot time, and once at shutdown.
- Shell variables are simple pairs of strings that share a single global namespace. You can't build a complex hierarchical data structure out of simple key/value pairs. Without support for information-rich data structures, the `rc(8)` system will be limited to simple problem domains.
- Shell scripts mix data and code in the same file. This makes it impossible to programmatically edit `rc(8)` scripts, except for very trivial edits like changing the name of a variable.
- Shell scripts are imprecise, and unless they are written with a high level of paranoia, it is possible for their environment to be polluted by the person running the script.
- Managing large numbers of servers at scale requires the use of a configuration management system, such as Puppet or Chef. Teaching these systems how to configure individual services is very difficult, because each service has its own configuration file format and location, and these vary across platforms.
- The `rc(8)` system is not portable. Each variant of Unix has its own spin on the classic design, be it System V or BSD. Independent software vendors do not have the resources to support all of these different init systems, so they will target the most popular platforms such as commercial Linux distributions.

The job framework attempts to solve all of the problems listed above, or at least move us in the right direction.

History and Motivation

`jobd` grew out of an experiment to write a clone of the `launchd(8)` system found in MacOS X. Its command line syntax and fault-handling capabilities are heavily inspired by the Service Management Facility (SMF) system in Solaris. It is not merely a clone or mashup of these two Unix utilities; rather, it borrows liberally from the best features of each, while avoiding some of the less desirable aspects.

I was motivated to write `jobd` partly in response to the `systemd` debacle that caused a major schism in the Linux community. The adoption of `systemd` was a major contributor to my decision to switch my personal computer from Linux to PC-BSD. Having made the switch, I started looking around for little side projects that I could do to give back to the FreeBSD Project.

The desire to create a new init system was there, but it needed a spark to get things going. Coincidentally, around the time I switched to PC-BSD, the NextBSD project was announced, and after listening to a few of Jordan Hubbard's talks about the benefits of `launchd`, I was ready to help out.

Unfortunately, upon closer inspection, I did not agree with the technical approach that NextBSD used to port `launchd`; namely, the decision to write a partial implementation of the Mach microkernel as a compatibility layer, to essentially make FreeBSD pretend to be a half-baked variant of MacOS X.

I had previously spent years porting the `kevent(2)` API from FreeBSD to other operating systems, and knew firsthand the pain involved in

creating compatibility layers to make one kernel pretend to be a different kernel. This experience made me highly skeptical of the decision to create a Mach compatibility layer for `launchd`. Even if the Mach compatibility layer worked perfectly when added to the FreeBSD kernel, it would add a lot of technical debt that might not be acceptable to the FreeBSD Project. It would be difficult and time-consuming to port to other Unix-like operating systems, and there would not be much of an appetite for it.

I wanted something that worked "right now" on existing operating systems, using standard POSIX APIs where possible, so I started a new implementation of `launchd` from scratch, and called it "`relaunchd`."

Work Thus Far: Implementation

A relatively stable and feature-complete version of `relaunchd` was created and released as version 0.6. This contained most of the features found in the original `launchd`.

At an iXsystems hackathon, I worked closely with Kris Moore to try using `relaunchd` to manage a new PC-BSD service called SysAdm. We ran into several issues that pointed at the inadequacy of the original `launchd` design with respect to how software is packaged for FreeBSD. MacOS X does not have the concept of a ports tree, and software is usually installed via a graphical installer program that takes care of interacting with `launchd`. It turned out that human interaction with `launchd` leaves a good bit to be desired.

`Launchd` also does not have any support for user-defined properties, or exposing properties and methods to the user to manipulate how a service is launched. People coming from a background of using `rc(8)` would be understandably unhappy about losing the ability to define a `$foo_flags` variable in `rc.conf` to control the `foo` service, for example.

Another weakness of the `launchd` design is the lack of a fault management facility. If a daemon crashes, `launchd` will sit in an infinite loop trying to restart it. There is no built-in mechanism for determining that a service is faulty, and doing something about it. I have spent several years administering Solaris 10 servers, and like the idea that misbehaving services can be transitioned to a faulted state. In `jobd`, it will be possible to define a fault handler script that is executed whenever a

STATUS	LABEL
running	com.example.proprietary-agent
offline	org.freebsd.ports.apache24
disabled	org.freebsd.ports.mysql
running	org.freebsd.ports.postgresql
running	system.jail.my-jail-name
waiting	system.at.job-23
done	system.cron.job-4175
disabled	system.service.sshd
running	system.service.xorg
running	user.1001.kde-session

job fault is detected. This fault handler could do pretty much anything: send an email to an administrator, pop up a window on a desktop, send an alert to a monitoring system, and so on.

I started to be concerned at the mismatch between what I wanted **relaunchd** to do, and the original **launchd** design. This led me to take a step back and ask, "What are the problems that we are trying to solve, and is the classic **launchd** design an adequate solution?" As is often the case, the real world throws more problems at you than you originally anticipated.

This growing frustration with **launchd** led me to think about the concept of a job framework as a low-level operating system construct. This framework would provide a library and supporting command line tools to allow for building a variety of other operating system facilities, such as a service manager and a **cron(8)** replacement. I started to visualize splitting **relaunchd** in half; the lower layer dealing with starting and stopping jobs, and the upper layer presenting the face of a "service manager" to the world. Over time, other domain-specific front ends could be added, all of them sharing the same back-end job manager.

Coincidentally, other developers at the hackathon were working on the **iocage** jail manager tool, and the **iohyve** virtual machine manager. I started to look at the nascent "lower layer of **relaunchd**" as something that could be reused by **iocage** and **iohyve**, since fundamentally a jail and a virtual machine are just processes.

This idea of sharing code across disparate tools may sound crazy, but if you look closely, what do **rc**, **iocage**, and **iohyve** have in common?

* They start daemon processes, either automatically when the system boots, or manually when requested by an administrator.
* They allow these daemons to be stopped, started, enabled, and disabled.
* They allow system administrators to customize certain aspects of how the daemons are started, similar to the concept of "job properties."
* In the case of **iocage** and **iohyve**, they create virtual network interfaces for each process.
* They may require firewall modifications, such as adding NAT rules or filtering rules.

In addition to supporting a wider variety of use cases, I also wanted to avoid "feature creep" that would see **relaunchd** slowly assume new roles and responsibilities that would cause it to become a giant mess. In a layered design, it's important to have clear separation of concerns between the layers, and it was not clear where the line should be drawn between **relaunchd** and tools that are built on top of **relaunchd**. Even the name, "**relaunchd**," creates the impression that it is purely an init system in the style of **launchd(8)**. In order to be useful in a wider variety of situations, it needed to stop being an init system altogether.

To avoid the feature creep problem, the job framework provides a core set of features to be consumed by higher layers in the stack. These higher layers are expected to handle the details of job specialization and customization to fit their specific problem domain. For example, if **iocage** wanted to offer a "Jail Marketplace" where people could download pre-configured jails, all of this functionality could be implemented within **iocage** itself, and not require any modifications to the underlying job framework.

Once the overall picture of what a job framework should look like was in place, the decision was made to rename the project to "**jobd**" and make a clean break with the past. The existing **launchd(8)** daemon would be renamed to **jobd(8)**, and the problematic **launchctl(8)** utility would be retired and replaced with a new set of command-line tools that would provide a better user experience for developers, packagers,

and system administrators.

At this point, I didn't totally want to reinvent the wheel when designing the new `command-line` interfaces, so I borrowed some of the concepts from the Solaris(TM) SMF framework, and created three new CLI tools: `jobadm(1)`, which operates against the job database; `jobctl(1)`, which controls a single job, handing operational requests like starting and stopping the job; and `jobcfg(1)`, which would handle everything related to getting and setting job properties.

Owing to its simple design, `relaunchd` was a few thousand lines of C code. With the new mission of `jobd`, I started to be concerned about the potential challenges of implementing the new job framework functionality in standard C. The original `launchd` API was simple enough to implement in C, but it would be hard to implement all of the powerful new functionality in C. This led to the decision to rewrite the project in C++ to take advantage of the richer set of data structures and move to a more object-oriented design.

Moving to C++ posed some initial challenges, not least of which was my total lack of practical experience with the language. After some initial growing pains, the benefits of C++ started to come to light, and it became easier to implement powerful functionality that would have been difficult to do in standard C. Learning a new language is also fun and interesting, and raises the challenge level of whatever you are trying to accomplish.

Goals

The overall goals of the `jobd` project are:

- Explore the idea of job management at the operating-system level. This includes educating people about job management concepts, and getting developers to think about how to solve problems using jobs.
- Be the best job management framework on the market; provide a general purpose framework for launching jobs within a single operating system instance; handle all aspects of the job lifecycle: setup, startup, supervise, kill, teardown. Be the reference implementation of the job management API.
- Make the easy things easy, and the hard things possible; make it trivial to start processes in a consistent manner. The hard thing is getting firewall rules, jails, datasets, network interfaces, configuration files, package dependencies, etc., all playing nice; jobd will make it possible to tie all of these things together with a single configura-