Pid Eins

レナート
لينارت

POSTED ON FR 30 APRIL 2010

Rethinking PID 1

If you are well connected or good at reading between the lines you might already know what this blog post is about. But even then you may find this story interesting. So grab a cup of coffee, sit down, and read what's coming.

This blog story is long, so even though I can only recommend reading the long story, here's the one sentence summary: we are experimenting with a new init system and it is fun.

Here's the code. And here's the story:

Process Identifier 1

On every Unix system there is one process with the special process identifier 1. It is started by the kernel before all other processes and is the parent process for all those other processes that have nobody else to be child of. Due to that it can do a lot of stuff that other processes cannot do. And it is also responsible for some things that other processes are not responsible for, such as bringing up and maintaining userspace during boot.

Historically on Linux the software acting as PID 1 was the venerable sysvinit package, though it had been showing its age for quite a while. Many replacements have been suggested, only one of them really took off: Upstart, which has by now found its way into all major distributions.

As mentioned, the central responsibility of an init system is to bring up userspace. And a good init system does that fast. Unfortunately, the traditional SysV init system was not particularly fast.

For a fast and efficient boot-up two things are crucial:

- To start **less**.
- And to start **more** in *parallel*.

What does that mean? Starting less means starting fewer services or deferring the starting of services until they are actually needed. There are some services where we know that they will be required sooner or later (syslog, D-Bus system bus, etc.), but for many others this isn't the case. For example, bluetoothd does not need to be running unless a bluetooth dongle is actually plugged in or an application wants to talk to its D-Bus interfaces. Same for a printing system: unless the machine physically is connected to a printer, or an application wants to print

something, there is no need to run a printing daemon such as CUPS. Avahi: if the machine is not connected to a network, there is no need to run Avahi, unless some application wants to use its APIs. And even SSH: as long as nobody wants to contact your machine there is no need to run it, as long as it is then started on the first connection. (And admit it, on most machines where sshd might be listening somebody connects to it only every other month or so.)

Starting more in parallel means that if we have to run something, we should not serialize its start-up (as sysvinit does), but run it all at the same time, so that the available CPU and disk IO bandwidth is maxed out, and hence the overall start-up time minimized.

Hardware and Software Change Dynamically

Modern systems (especially general purpose OS) are highly dynamic in their configuration and use: they are mobile, different applications are started and stopped, different hardware added and removed again. An init system that is responsible for maintaining services needs to listen to hardware and software changes. It needs to dynamically start (and sometimes stop) services as they are needed to run a program or enable some hardware.

Most current systems that try to parallelize boot-up still synchronize the start-up of the various daemons involved: since Avahi needs D-Bus, D-Bus is started first, and only when D-Bus signals that it is ready, Avahi is started too. Similar for other services: livirtd and X11 need HAL (well, I am considering the Fedora 13 services here, ignore that HAL is obsolete), hence HAL is started first, before livirtd and X11 are started. And libvirtd also needs Avahi, so it waits for Avahi too. And all of them require syslog, so they all wait until Syslog is fully started up and initialized. And so on.

Parallelizing Socket Services

This kind of start-up synchronization results in the serialization of a significant part of the boot process. Wouldn't it be great if we could get rid of the synchronization and serialization cost? Well, we can, actually. For that, we need to understand what exactly the daemons require from each other, and why their start-up is delayed. For traditional Unix daemons, there's one answer to it: they wait until the socket the other daemon offers its services on is ready for connections. Usually that is an AF_UNIX socket in the file-system, but it could be AF_INET[6], too. For example, clients of D-Bus wait that `/var/run/dbus/system_bus_socket` can be connected to, clients of syslog wait for `/dev/log`, clients of CUPS wait for `/var/run/cups/cups.sock` and NFS mounts wait for `/var/run/rpcbind.sock` and the portmapper IP port, and so on. And think about it, this is actually the only thing they wait for!

Now, if that's all they are waiting for, if we manage to make those sockets available for connection earlier and only actually wait for that instead of the full daemon start-up, then we can speed up the entire boot and start more processes in parallel. So, how can we do that? Actually quite easily in Unix-like systems: we can create the listening sockets **before** we actually start the daemon, and then just pass the socket during `exec()` to it. That way, we can create **all** sockets for **all** daemons in one step in the init system, and then in a second step run all daemons at once. If a service needs another, and it is not fully started up, that's completely OK: what will happen is that the connection is queued in the providing service and the client will potentially block on that single request. But only that one client will block and only on that one request. Also, dependencies between services will no longer necessarily have to be configured to allow proper parallelized start-up: if we start all sockets at once and a service needs another it can be sure that it can connect to its socket.

Because this is at the core of what is following, let me say this again, with different words and by example: if you start syslog and and various syslog clients at the same time, what will happen in the scheme pointed out above is that the messages of the clients will be added to the `/dev/log` socket buffer. As long as that buffer doesn't run full, the clients will not have to wait in any way and can immediately proceed with their start-up. As soon as syslog itself finished start-up,

it will dequeue all messages and process them. Another example: we start D-Bus and several clients at the same time. If a synchronous bus request is sent and hence a reply expected, what will happen is that the client will have to block, however only that one client and only until D-Bus managed to catch up and process it.

Basically, the kernel socket buffers help us to maximize parallelization, and the ordering and synchronization is done by the kernel, without any further management from userspace! And if all the sockets are available before the daemons actually start-up, dependency management also becomes redundant (or at least secondary): if a daemon needs another daemon, it will just connect to it. If the other daemon is already started, this will immediately succeed. If it isn't started but in the process of being started, the first daemon will not even have to wait for it, unless it issues a synchronous request. And even if the other daemon is not running at all, it can be auto-spawned. From the first daemon's perspective there is no difference, hence dependency management becomes mostly unnecessary or at least secondary, and all of this in optimal parallelization and optionally with on-demand loading. On top of this, this is also more robust, because the sockets stay available regardless whether the actual daemons might temporarily become unavailable (maybe due to crashing). In fact, you can easily write a daemon with this that can run, and exit (or crash), and run again and exit again (and so on), and all of that without the clients noticing or loosing any request.

It's a good time for a pause, go and refill your coffee mug, and be assured, there is more interesting stuff following.

But first, let's clear a few things up: is this kind of logic new? No, it certainly is not. The most prominent system that works like this is Apple's launchd system: on MacOS the listening of the sockets is pulled out of all daemons and done by launchd. The services themselves hence can all start up in parallel and dependencies need not to be configured for them. And that is actually a really ingenious design, and the primary reason why MacOS manages to provide the fantastic boot-up times it provides. I can highly recommend this video where the launchd folks explain what they are doing. Unfortunately this idea never really took on outside of the Apple camp.

The idea is actually even older than launchd. Prior to launchd the venerable `inetd` worked much like this: sockets were centrally created in a daemon that would start the actual service daemons passing the socket file descriptors during `exec()`. However the focus of `inetd` certainly wasn't local services, but Internet services (although later reimplementations supported AF_UNIX sockets, too). It also wasn't a tool to parallelize boot-up or even useful for getting implicit dependencies right.

For TCP sockets `inetd` was primarily used in a way that for every incoming connection a new daemon instance was spawned. That meant that for each connection a new process was spawned and initialized, which is not a recipe for high-performance servers. However, right from the beginning `inetd` also supported another mode, where a single daemon was spawned on the first connection, and that single instance would then go on and also accept the follow-up connections (that's what the `wait/nowait` option in `inetd.conf` was for, a particularly badly documented option, unfortunately.) Per-connection daemon starts probably gave inetd its bad reputation for being slow. But that's not entirely fair.

Parallelizing Bus Services

Modern daemons on Linux tend to provide services via D-Bus instead of plain AF_UNIX sockets. Now, the question is, for those services, can we apply the same parallelizing boot logic as for traditional socket services? Yes, we can, D-Bus already has all the right hooks for it: using bus activation a service can be started the first time it is accessed. Bus activation also gives us the minimal per-request synchronisation we need for starting up the providers and the consumers of D-Bus services at the same time: if we want to start Avahi at the same time as CUPS (side note: CUPS uses Avahi to browse for mDNS/DNS-SD printers), then we can simply

run them at the same time, and if CUPS is quicker than Avahi via the bus activation logic we can get D-Bus to queue the request until Avahi manages to establish its service name.

So, in summary: the socket-based service activation and the bus-based service activation together enable us to start **all** daemons in parallel, without any further synchronization. Activation also allows us to do lazy-loading of services: if a service is rarely used, we can just load it the first time somebody accesses the socket or bus name, instead of starting it during boot.

And if that's not great, then I don't **know** what is great!

Parallelizing File System Jobs

If you look at the serialization graphs of the boot process of current distributions, there are more synchronisation points than just daemon start-ups: most prominently there are file-system related jobs: mounting, fscking, quota. Right now, on boot-up a lot of time is spent idling to wait until all devices that are listed in `/etc/fstab` show up in the device tree and are then fsck'ed, mounted, quota checked (if enabled). Only after that is fully finished we go on and boot the actual services.

Can we improve this? It turns out we can. Harald Hoyer came up with the idea of using the venerable autofs system for this:

Just like a `connect()` call shows that a service is interested in another service, an `open()` (or a similar call) shows that a service is interested in a specific file or file-system. So, in order to improve how much we can parallelize we can make those apps wait only if a file-system they are looking for is not yet mounted and readily available: we set up an autofs mount point, and then when our file-system finished fsck and quota due to normal boot-up we replace it by the real mount. While the file-system is not ready yet, the access will be queued by the kernel and the accessing process will block, but only that one daemon and only that one access. And this way we can begin starting our daemons even before all file systems have been fully made available -- without them missing any files, and maximizing parallelization.

Parallelizing file system jobs and service jobs does not make sense for `/`, after all that's where the service binaries are usually stored. However, for file-systems such as `/home`, that usually are bigger, even encrypted, possibly remote and seldom accessed by the usual boot-up daemons, this can improve boot time considerably. It is probably not necessary to mention this, but virtual file systems, such as procfs or sysfs should never be mounted via autofs.

I wouldn't be surprised if some readers might find integrating autofs in an init system a bit fragile and even weird, and maybe more on the "crackish" side of things. However, having played around with this extensively I can tell you that this actually feels quite right. Using autofs here simply means that we can create a mount point without having to provide the backing file system right-away. In effect it hence only delays accesses. If an application tries to access an autofs file-system and we take very long to replace it with the real file-system, it will hang in an interruptible sleep, meaning that you can safely cancel it, for example via C-c. Also note that at any point, if the mount point should not be mountable in the end (maybe because fsck failed), we can just tell autofs to return a clean error code (like ENOENT). So, I guess what I want to say is that even though integrating autofs into an init system might appear adventurous at first, our experimental code has shown that this idea works surprisingly well in practice -- if it is done for the right reasons and the right way.

Also note that these should be *direct* autofs mounts, meaning that from an application perspective there's little effective difference between a classic mount point and one based on autofs.

Keeping the First User PID Small

Another thing we can learn from the MacOS boot-up logic is that shell scripts are evil. Shell is fast and shell is slow. It is fast to hack, but slow in execution. The

classic sysvinit boot logic is modelled around shell scripts. Whether it is `/bin/bash` or any other shell (that was written to make shell scripts faster), in the end the approach is doomed to be slow. On my system the scripts in `/etc/init.d` call `grep` at least 77 times. `awk` is called 92 times, `cut` 23 and `sed` 74. Every time those commands (and others) are called, a process is spawned, the libraries searched, some start-up stuff like i18n and so on set up and more. And then after seldom doing more than a trivial string operation the process is terminated again. Of course, that has to be incredibly slow. No other language but shell would do something like that. On top of that, shell scripts are also very fragile, and change their behaviour drastically based on environment variables and suchlike, stuff that is hard to oversee and control.

So, let's get rid of shell scripts in the boot process! Before we can do that we need to figure out what they are currently actually used for: well, the big picture is that most of the time, what they do is actually quite boring. Most of the scripting is spent on trivial setup and tear-down of services, and should be rewritten in C, either in separate executables, or moved into the daemons themselves, or simply be done in the init system.

It is not likely that we can get rid of shell scripts during system boot-up entirely anytime soon. Rewriting them in C takes time, in a few case does not really make sense, and sometimes shell scripts are just too handy to do without. But we can certainly make them less prominent.

A good metric for measuring shell script infestation of the boot process is the PID number of the first process you can start after the system is fully booted up. Boot up, log in, open a terminal, and type `echo $$`. Try that on your Linux system, and then compare the result with MacOS! (Hint, it's something like this: Linux PID 1823; MacOS PID 154, measured on test systems we own.)

Keeping Track of Processes

A central part of a system that starts up and maintains services should be process babysitting: it should watch services. Restart them if they shut down. If they crash it should collect information about them, and keep it around for the administrator, and cross-link that information with what is available from crash dump systems such as abrt, and in logging systems like syslog or the audit system.

It should also be capable of shutting down a service completely. That might sound easy, but is harder than you think. Traditionally on Unix a process that does double-forking can escape the supervision of its parent, and the old parent will not learn about the relation of the new process to the one it actually started. An example: currently, a misbehaving CGI script that has double-forked is not terminated when you shut down Apache. Furthermore, you will not even be able to figure out its relation to Apache, unless you know it by name and purpose.

So, how can we keep track of processes, so that they cannot escape the babysitter, and that we can control them as one unit even if they fork a gazillion times?

Different people came up with different solutions for this. I am not going into much detail here, but let's at least say that approaches based on ptrace or the netlink connector (a kernel interface which allows you to get a netlink message each time any process on the system fork()s or exit()s) that some people have investigated and implemented, have been criticised as ugly and not very scalable.

So what can we do about this? Well, since quite a while the kernel knows Control Groups (aka "cgroups"). Basically they allow the creation of a hierarchy of groups of processes. The hierarchy is directly exposed in a virtual file-system, and hence easily accessible. The group names are basically directory names in that file-system. If a process belonging to a specific cgroup fork()s, its child will become a member of the same group. Unless it is privileged and has access to the cgroup file system it cannot escape its group. Originally, cgroups have been introduced into the kernel for the purpose of containers: certain kernel subsystems can enforce limits on resources of certain groups, such as limiting CPU or memory usage. Traditional resource limits (as implemented by `setrlimit()`) are (mostly) per-process. cgroups on the other hand let you enforce limits on entire groups of processes. cgroups are also useful to enforce limits outside of the immediate

container use case. You can use it for example to limit the total amount of memory or CPU Apache and all its children may use. Then, a misbehaving CGI script can no longer escape your `setrlimit()` resource control by simply forking away.

In addition to container and resource limit enforcement cgroups are very useful to keep track of daemons: cgroup membership is securely inherited by child processes, they cannot escape. There's a notification system available so that a supervisor process can be notified when a cgroup runs empty. You can find the cgroups of a process by reading `/proc/$PID/cgroup`. cgroups hence make a very good choice to keep track of processes for babysitting purposes.

Controlling the Process Execution Environment

A good babysitter should not only oversee and control when a daemon starts, ends or crashes, but also set up a good, minimal, and secure working environment for it.

That means setting obvious process parameters such as the `setrlimit()` resource limits, user/group IDs or the environment block, but does not end there. The Linux kernel gives users and administrators a lot of control over processes (some of it is rarely used, currently). For each process you can set CPU and IO scheduler controls, the capability bounding set, CPU affinity or of course cgroup environments with additional limits, and more.

As an example, `ioprio_set()` with `IOPRIO_CLASS_IDLE` is a great away to minimize the effect of `locate`'s `updatedb` on system interactivity.

On top of that certain high-level controls can be very useful, such as setting up read-only file system overlays based on read-only bind mounts. That way one can run certain daemons so that all (or some) file systems appear read-only to them, so that EROFS is returned on every write request. As such this can be used to lock down what daemons can do similar in fashion to a poor man's SELinux policy system (but this certainly doesn't replace SELinux, don't get any bad ideas, please).

Finally logging is an important part of executing services: ideally every bit of output a service generates should be logged away. An init system should hence provide logging to daemons it spawns right from the beginning, and connect stdout and stderr to syslog or in some cases even `/dev/kmsg` which in many cases makes a very useful replacement for syslog (embedded folks, listen up!), especially in times where the kernel log buffer is configured ridiculously large out-of-the-box.

On Upstart

To begin with, let me emphasize that I actually like the code of Upstart, it is very well commented and easy to follow. It's certainly something other projects should learn from (including my own).

That being said, I can't say I agree with the general approach of Upstart. But first, a bit more about the project:

Upstart does not share code with sysvinit, and its functionality is a super-set of it, and provides compatibility to some degree with the well known SysV init scripts. It's main feature is its event-based approach: starting and stopping of processes is bound to "events" happening in the system, where an "event" can be a lot of different things, such as: a network interfaces becomes available or some other software has been started.

Upstart does service serialization via these events: if the `syslog-started` event is triggered this is used as an indication to start D-Bus since it can now make use of Syslog. And then, when `dbus-started` is triggered, `NetworkManager` is started, since it may now use `D-Bus`, and so on.

One could say that this way the actual logical dependency tree that exists and is understood by the admin or developer is translated and encoded into event and action rules: every logical "a needs b" rule that the administrator/developer is aware of becomes a "start a when b is started" plus "stop a when b is stopped". In some way this certainly is a simplification: especially for the code in Upstart itself.

However I would argue that this simplification is actually detrimental. First of all, the logical dependency system does not go away, the person who is writing Upstart files must now translate the dependencies manually into these event/action rules (actually, two rules for each dependency). So, instead of letting the computer figure out what to do based on the dependencies, the user has to manually translate the dependencies into simple event/action rules. Also, because the dependency information has never been encoded it is not available at runtime, effectively meaning that an administrator who tries to figure our *why* something happened, i.e. why a is started when b is started, has no chance of finding that out.

Furthermore, the event logic turns around all dependencies, from the feet onto their head. Instead of *minimizing* the amount of work (which is something that a good init system should focus on, as pointed out in the beginning of this blog story), it actually *maximizes* the amount of work to do during operations. Or in other words, instead of having a clear goal and only doing the things it really needs to do to reach the goal, it does one step, and then after finishing it, it does **all** steps that possibly could follow it.

Or to put it simpler: the fact that the user just started D-Bus is in no way an indication that NetworkManager should be started too (but this is what Upstart would do). It's right the other way round: when the user asks for NetworkManager, that is definitely an indication that D-Bus should be started too (which is certainly what most users would expect, right?).

A good init system should start only what is needed, and that on-demand. Either lazily or parallelized and in advance. However it should not start more than necessary, particularly not everything installed that could use that service.

Finally, I fail to see the actual usefulness of the event logic. It appears to me that most events that are exposed in Upstart actually are not punctual in nature, but have duration: a service starts, is running, and stops. A device is plugged in, is available, and is plugged out again. A mount point is in the process of being mounted, is fully mounted, or is being unmounted. A power plug is plugged in, the system runs on AC, and the power plug is pulled. Only a minority of the events an init system or process supervisor should handle are actually punctual, most of them are tuples of start, condition, and stop. This information is again not available in Upstart, because it focuses in singular events, and ignores durable dependencies.

Now, I am aware that some of the issues I pointed out above are in some way mitigated by certain more recent changes in Upstart, particularly condition based syntaxes such as `start on (local-filesystems and net-device-up IFACE=lo)` in Upstart rule files. However, to me this appears mostly as an attempt to fix a system whose core design is flawed.

Besides that Upstart does OK for babysitting daemons, even though some choices might be questionable (see above), and there are certainly a lot of missed opportunities (see above, too).

There are other init systems besides sysvinit, Upstart and launchd. Most of them offer little substantial more than Upstart or sysvinit. The most interesting other contender is Solaris SMF, which supports proper dependencies between services. However, in many ways it is overly complex and, let's say, a bit *academic* with its excessive use of XML and new terminology for known things. It is also closely bound to Solaris specific features such as the *contract* system.

Putting it All Together: systemd

Well, this is another good time for a little pause, because after I have hopefully explained above what I think a good PID 1 should be doing and what the current most used system does, we'll now come to where the beef is. So, go and refill you coffee mug again. It's going to be worth it.

You probably guessed it: what I suggested above as requirements and features for an ideal init system is actually available now, in a (still experimental) init system called `systemd`, and which I hereby want to announce. Again, here's the code. And here's a quick rundown of its features, and the rationale behind them:

systemd starts up and supervises the entire system (hence the name...). It implements all of the features pointed out above and a few more. It is based around the notion of *units*. Units have a name and a type. Since their configuration is usually loaded directly from the file system, these unit names are actually file names. Example: a unit `avahi.service` is read from a configuration file by the same name, and of course could be a unit encapsulating the Avahi daemon. There are several kinds of units:

1. `service`: these are the most obvious kind of unit: daemons that can be started, stopped, restarted, reloaded. For compatibility with SysV we not only support our own configuration files for services, but also are able to read classic SysV init scripts, in particular we parse the LSB header, if it exists. `/etc/init.d` is hence not much more than just another source of configuration.
2. `socket`: this unit encapsulates a socket in the file-system or on the Internet. We currently support AF_INET, AF_INET6, AF_UNIX sockets of the types stream, datagram, and sequential packet. We also support classic FIFOs as transport. Each `socket` unit has a matching `service` unit, that is started if the first connection comes in on the socket or FIFO. Example: `nscd.socket` starts `nscd.service` on an incoming connection.
3. `device`: this unit encapsulates a device in the Linux device tree. If a device is marked for this via udev rules, it will be exposed as a `device` unit in systemd. Properties set with `udev` can be used as configuration source to set dependencies for device units.
4. `mount`: this unit encapsulates a mount point in the file system hierarchy. systemd monitors all mount points how they come and go, and can also be used to mount or unmount mount-points. `/etc/fstab` is used here as an additional configuration source for these mount points, similar to how SysV init scripts can be used as additional configuration source for `service` units.
5. `automount`: this unit type encapsulates an automount point in the file system hierarchy. Each `automount` unit has a matching `mount` unit, which is started (i.e. mounted) as soon as the automount directory is accessed.
6. `target`: this unit type is used for logical grouping of units: instead of actually doing anything by itself it simply references other units, which thereby can be controlled together. Examples for this are: `multi-user.target`, which is a target that basically plays the role of run-level 5 on classic SysV system, or `bluetooth.target` which is requested as soon as a bluetooth dongle becomes available and which simply pulls in bluetooth related services that otherwise would not need to be started: `bluetoothd` and `obexd` and suchlike.
7. `snapshot`: similar to `target` units snapshots do not actually do anything themselves and their only purpose is to reference other units. Snapshots can be used to save/rollback the state of all services and units of the init system. Primarily it has two intended use cases: to allow the user to temporarily enter a specific state such as "Emergency Shell", terminating current services, and provide an easy way to return to the state before, pulling up all services again that got temporarily pulled down. And to ease support for system suspending: still many services cannot correctly deal with system suspend, and it is often a better idea to shut them down before suspend, and restore them afterwards.

All these units can have dependencies between each other (both positive and negative, i.e. 'Requires' and 'Conflicts'): a device can have a dependency on a service, meaning that as soon as a device becomes available a certain service is started. Mounts get an implicit dependency on the device they are mounted from. Mounts also gets implicit dependencies to mounts that are their prefixes (i.e. a mount `/home/lennart` implicitly gets a dependency added to the mount for `/home`) and so on.

A short list of other features:

1. For each process that is spawned, you may control: the environment, resource limits, working and root directory, umask, OOM killer adjustment, nice level, IO class and priority, CPU policy and priority, CPU affinity, timer slack, user id, group id, supplementary group ids, readable/writable/inaccessible directories, shared/private/slave mount flags, capabilities/bounding set, secure bits, CPU

scheduler reset of fork, private `/tmp` name-space, cgroup control for various subsystems. Also, you can easily connect stdin/stdout/stderr of services to syslog, `/dev/kmsg`, arbitrary TTYs. If connected to a TTY for input systemd will make sure a process gets exclusive access, optionally waiting or enforcing it.

2. Every executed process gets its own cgroup (currently by default in the debug subsystem, since that subsystem is not otherwise used and does not much more than the most basic process grouping), and it is very easy to configure systemd to place services in cgroups that have been configured externally, for example via the libcgroups utilities.
3. The native configuration files use a syntax that closely follows the well-known `.desktop` files. It is a simple syntax for which parsers exist already in many software frameworks. Also, this allows us to rely on existing tools for i18n for service descriptions, and similar. Administrators and developers don't need to learn a new syntax.
4. As mentioned, we provide compatibility with SysV init scripts. We take advantages of LSB and Red Hat chkconfig headers if they are available. If they aren't we try to make the best of the otherwise available information, such as the start priorities in `/etc/rc.d`. These init scripts are simply considered a different source of configuration, hence an easy upgrade path to proper systemd services is available. Optionally we can read classic PID files for services to identify the main pid of a daemon. Note that we make use of the dependency information from the LSB init script headers, and translate those into native systemd dependencies. Side note: Upstart is unable to harvest and make use of that information. Boot-up on a plain Upstart system with mostly LSB SysV init scripts will hence not be parallelized, a similar system running systemd however will. In fact, for Upstart all SysV scripts together make one job that is executed, they are not treated individually, again in contrast to systemd where SysV init scripts are just another source of configuration and are all treated and controlled individually, much like any other native systemd service.
5. Similarly, we read the existing `/etc/fstab` configuration file, and consider it just another source of configuration. Using the `comment=` fstab option you can even mark `/etc/fstab` entries to become `systemd` controlled automount points.
6. If the same unit is configured in multiple configuration sources (e.g. `/etc/systemd/system/avahi.service` exists, and `/etc/init.d/avahi` too), then the native configuration will always take precedence, the legacy format is ignored, allowing an easy upgrade path and packages to carry both a SysV init script and a systemd service file for a while.
7. We support a simple templating/instance mechanism. Example: instead of having six configuration files for six gettys, we only have one `getty@.service` file which gets instantiated to `getty@tty2.service` and suchlike. The interface part can even be inherited by dependency expressions, i.e. it is easy to encode that a service `dhcpcd@eth0.service` pulls in `avahi-autoipd@eth0.service`, while leaving the `eth0` string wild-carded.
8. For socket activation we support full compatibility with the traditional inetd modes, as well as a very simple mode that tries to mimic launchd socket activation and is recommended for new services. The inetd mode only allows passing one socket to the started daemon, while the native mode supports passing arbitrary numbers of file descriptors. We also support one instance per connection, as well as one instance for all connections modes. In the former mode we name the cgroup the daemon will be started in after the connection parameters, and utilize the templating logic mentioned above for this. Example: `sshd.socket` might spawn services `sshd@192.168.0.1-4711-192.168.0.2-22.service` with a cgroup of `sshd@.service/192.168.0.1-4711-192.168.0.2-22` (i.e. the IP address and port numbers are used in the instance names. For AF_UNIX sockets we use PID and user id of the connecting client). This provides a nice way for the administrator to identify the various instances of a daemon and control their runtime individually. The native socket passing mode is very easily implementable in applications: if `$LISTEN_FDS` is set it contains the number of sockets passed and the daemon will find them sorted as listed in

the `.service` file, starting from file descriptor 3 (a nicely written daemon could also use `fstat()` and `getsockname()` to identify the sockets in case it receives more than one). In addition we set `$LISTEN_PID` to the PID of the daemon that shall receive the fds, because environment variables are normally inherited by sub-processes and hence could confuse processes further down the chain. Even though this socket passing logic is very simple to implement in daemons, we will provide a BSD-licensed reference implementation that shows how to do this. We have ported a couple of existing daemons to this new scheme.

9. We provide compatibility with `/dev/initctl` to a certain extent. This compatibility is in fact implemented with a FIFO-activated service, which simply translates these legacy requests to D-Bus requests. Effectively this means the old `shutdown`, `poweroff` and similar commands from Upstart and `sysvinit` continue to work with systemd.
10. We also provide compatibility with `utmp` and `wtmp`. Possibly even to an extent that is far more than healthy, given how crufty `utmp` and `wtmp` are.
11. systemd supports several kinds of dependencies between units. `After`/`Before` can be used to fix the ordering how units are activated. It is completely orthogonal to `Requires` and `Wants`, which express a positive requirement dependency, either mandatory, or optional. Then, there is `Conflicts` which expresses a negative requirement dependency. Finally, there are three further, less used dependency types.
12. systemd has a minimal transaction system. Meaning: if a unit is requested to start up or shut down we will add it and all its dependencies to a temporary *transaction*. Then, we will verify if the transaction is consistent (i.e. whether the ordering via `After`/`Before` of all units is cycle-free). If it is not, systemd will try to fix it up, and removes non-essential jobs from the transaction that might remove the loop. Also, systemd tries to suppress non-essential jobs in the transaction that would stop a running service. Non-essential jobs are those which the original request did not directly include but which where pulled in by `Wants` type of dependencies. Finally we check whether the jobs of the transaction contradict jobs that have already been queued, and optionally the transaction is aborted then. If all worked out and the transaction is consistent and minimized in its impact it is merged with all already outstanding jobs and added to the run queue. Effectively this means that before executing a requested operation, we will verify that it makes sense, fixing it if possible, and only failing if it really cannot work.
13. We record start/exit time as well as the PID and exit status of every process we spawn and supervise. This data can be used to cross-link daemons with their data in abrtd, auditd and syslog. Think of an UI that will highlight crashed daemons for you, and allows you to easily navigate to the respective UIs for syslog, abrt, and auditd that will show the data generated from and for this daemon on a specific run.
14. We support reexecution of the init process itself at any time. The daemon state is serialized before the reexecution and deserialized afterwards. That way we provide a simple way to facilitate init system upgrades as well as handover from an initrd daemon to the final daemon. Open sockets and autofs mounts are properly serialized away, so that they stay connectible all the time, in a way that clients will not even notice that the init system reexecuted itself. Also, the fact that a big part of the service state is encoded anyway in the cgroup virtual file system would even allow us to resume execution without access to the serialization data. The reexecution code paths are actually mostly the same as the init system configuration reloading code paths, which guarantees that reexecution (which is probably more seldom triggered) gets similar testing as reloading (which is probably more common).
15. Starting the work of removing shell scripts from the boot process we have recoded part of the basic system setup in C and moved it directly into systemd. Among that is mounting of the API file systems (i.e. virtual file systems such as `/proc`, `/sys` and `/dev`.) and setting of the host-name.
16. Server state is introspectable and controllable via D-Bus. This is not complete yet but quite extensive.
17. While we want to emphasize socket-based and bus-name-based activation, and we hence support dependencies between sockets and services, we also

support traditional inter-service dependencies. We support multiple ways how such a service can signal its readiness: by forking and having the start process exit (i.e. traditional `daemonize()` behaviour), as well as by watching the bus until a configured service name appears.

18. There's an interactive mode which asks for confirmation each time a process is spawned by systemd. You may enable it by passing `systemd.confirm_spawn=1` on the kernel command line.
19. With the `systemd.default=` kernel command line parameter you can specify which unit systemd should start on boot-up. Normally you'd specify something like `multi-user.target` here, but another choice could even be a single service instead of a target, for example out-of-the-box we ship a service `emergency.service` that is similar in its usefulness as `init=/bin/bash`, however has the advantage of actually running the init system, hence offering the option to boot up the full system from the emergency shell.
20. There's a minimal UI that allows you to start/stop/introspect services. It's far from complete but useful as a debugging tool. It's written in Vala (yay!) and goes by the name of `systemadm`.

It should be noted that systemd uses many Linux-specific features, and does not limit itself to POSIX. That unlocks a lot of functionality a system that is designed for portability to other operating systems cannot provide.

Status

All the features listed above are already implemented. Right now systemd can already be used as a drop-in replacement for Upstart and sysvinit (at least as long as there aren't too many native upstart services yet. Thankfully most distributions don't carry too many native Upstart services yet.)

However, testing has been minimal, our version number is currently at an impressive 0. Expect breakage if you run this in its current state. That said, overall it should be quite stable and some of us already boot their normal development systems with systemd (in contrast to VMs only). YMMV, especially if you try this on distributions we developers don't use.

Where is This Going?

The feature set described above is certainly already comprehensive. However, we have a few more things on our plate. I don't really like speaking too much about big plans but here's a short overview in which direction we will be pushing this:

We want to add at least two more unit types: `swap` shall be used to control swap devices the same way we already control mounts, i.e. with automatic dependencies on the device tree devices they are activated from, and suchlike. `timer` shall provide functionality similar to `cron`, i.e. starts services based on time events, the focus being both monotonic clock and wall-clock/calendar events. (i.e. "start this 5h after it last ran" as well as "start this every monday 5 am")

More importantly however, it is also our plan to experiment with systemd not only for optimizing boot times, but also to make it the ideal session manager, to replace (or possibly just augment) `gnome-session`, `kdeinit` and similar daemons. The problem set of a session manager and an init system are very similar: quick start-up is essential and babysitting processes the focus. Using the same code for both uses hence suggests itself. Apple recognized that and does just that with launchd. And so should we: socket and bus based activation and parallelization is something session services and system services can benefit from equally.

I should probably note that all three of these features are already partially available in the current code base, but not complete yet. For example, already, you can run systemd just fine as a normal user, and it will detect that is run that way and support for this mode has been available since the very beginning, and is in the very core. (It is also exceptionally useful for debugging! This works fine even without having the system otherwise converted to systemd for booting.)

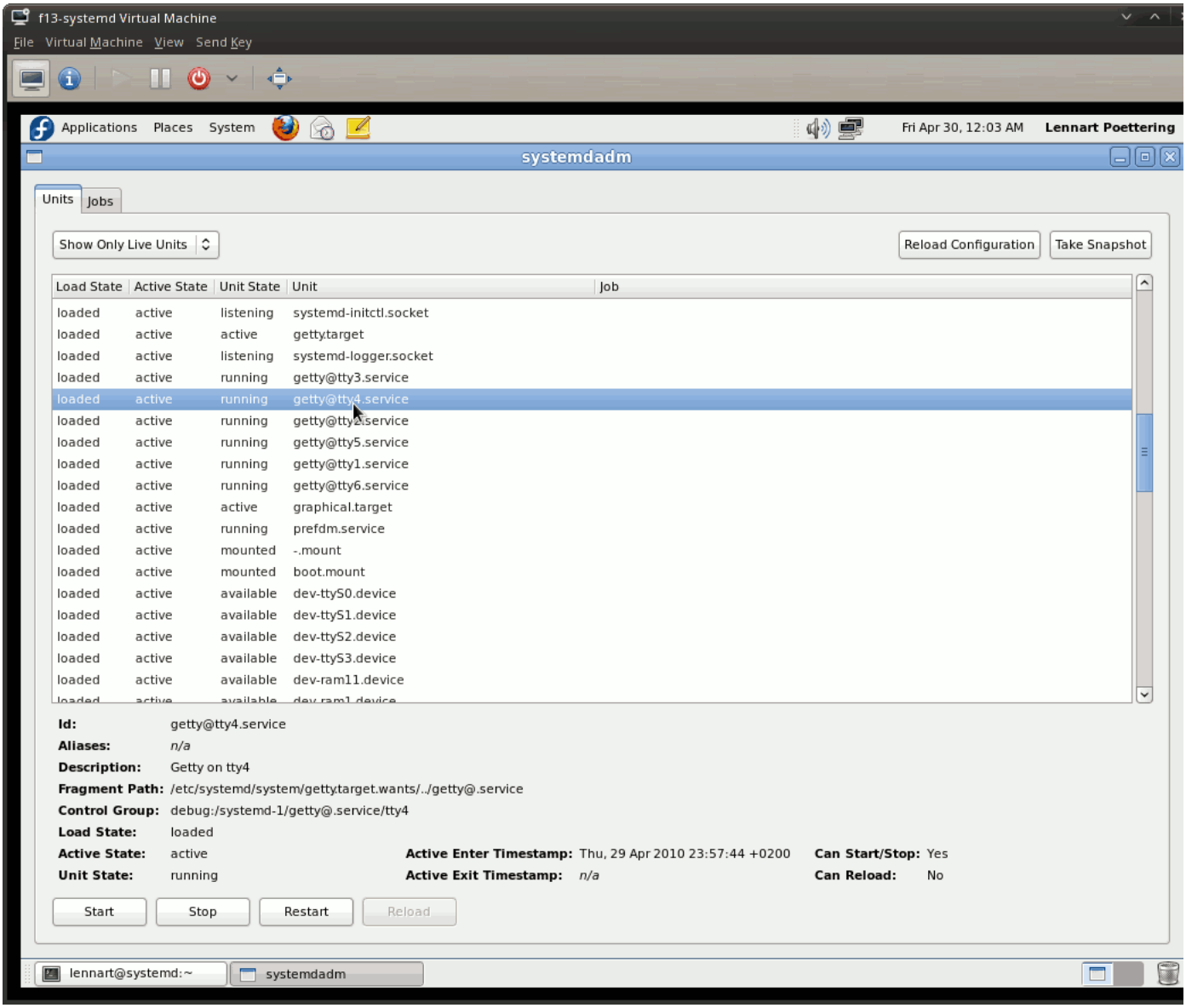
However, there are some things we probably should fix in the kernel and elsewhere before finishing work on this: we need swap status change notifications from the kernel similar to how we can already subscribe to mount changes; we want a notification when CLOCK_REALTIME jumps relative to CLOCK_MONOTONIC; we want to allow normal processes to get some init-like powers; we need a well-defined place where we can put user sockets. None of these issues are really essential for systemd, but they'd certainly improve things.

You Want to See This in Action?

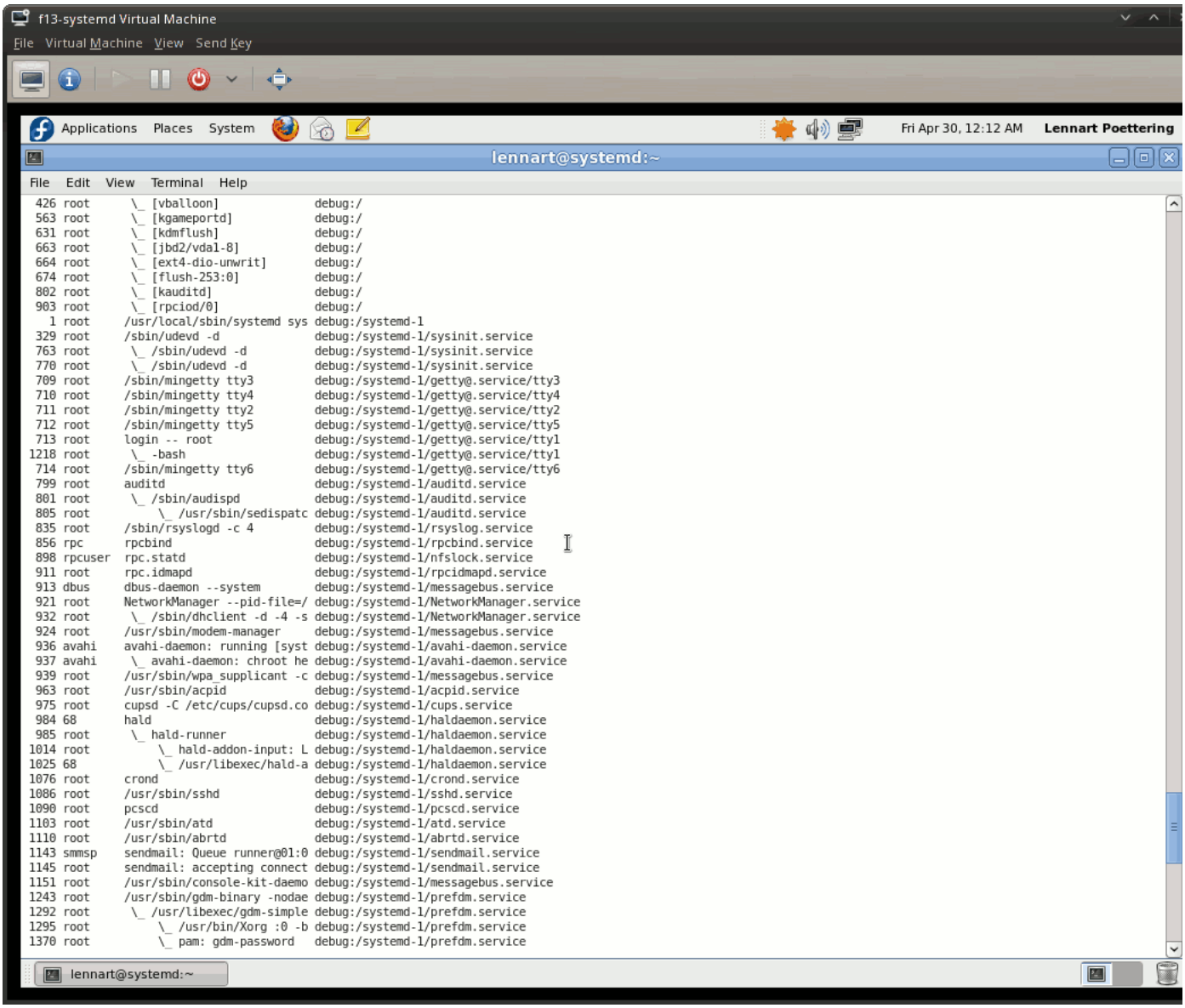
Currently, there are no tarball releases, but it should be straightforward to check out the code from our repository. In addition, to have something to start with, here's a tarball with unit configuration files that allows an otherwise unmodified Fedora 13 system to work with systemd. We have no RPMs to offer you for now.

An easier way is to download this Fedora 13 qemu image, which has been prepared for systemd. In the grub menu you can select whether you want to boot the system with Upstart or systemd. Note that this system is minimally modified only. Service information is read exclusively from the existing SysV init scripts. Hence it will not take advantage of the full socket and bus-based parallelization pointed out above, however it will interpret the parallelization hints from the LSB headers, and hence boots faster than the Upstart system, which in Fedora does not employ any parallelization at the moment. The image is configured to output debug information on the serial console, as well as writing it to the kernel log buffer (which you may access with `dmesg`). You might want to run `qemu` configured with a virtual serial terminal. All passwords are set to `systemd`.

Even simpler than downloading and booting the qemu image is looking at pretty screen-shots. Since an init system usually is well hidden beneath the user interface, some shots of `systemadm` and `ps` must do:



That's systemadm showing all loaded units, with more detailed information on one of the getty instances.



That's an excerpt of the output of `ps xaf -eo pid,user,args,cgroup` showing how neatly the processes are sorted into the cgroup of their service. (The fourth column is the cgroup, the `debug:` prefix is shown because we use the debug cgroup controller for systemd, as mentioned earlier. This is only temporary.)

Note that both of these screenshots show an only minimally modified Fedora 13 Live CD installation, where services are exclusively loaded from the existing SysV init scripts. Hence, this does not use socket or bus activation for any existing service.

Sorry, no bootcharts or hard data on start-up times for the moment. We'll publish that as soon as we have fully parallelized all services from the default Fedora install. Then, we'll welcome you to benchmark the systemd approach, and provide our own benchmark data as well.

Well, presumably everybody will keep bugging me about this, so here are two numbers I'll tell you. However, they are completely unscientific as they are measured for a VM (single CPU) and by using the stop timer in my watch. Fedora 13 booting up with Upstart takes 27s, with systemd we reach 24s (from grub to gdm, same system, same settings, shorter value of two bootups, one immediately following the other). Note however that this shows nothing more than the speedup effect reached by using the LSB dependency information parsed from the init script headers for parallelization. Socket or bus based activation was not utilized for this, and hence these numbers are unsuitable to assess the ideas pointed out above. Also, systemd was set to debug verbosity levels on a serial console. So again, this benchmark data has barely any value.

Writing Daemons

An ideal daemon for use with systemd does a few things differently then things were traditionally done. Later on, we will publish a longer guide explaining and suggesting how to write a daemon for use with this systemd. Basically, things get simpler for daemon developers:

- We ask daemon writers not to fork or even double fork in their processes, but run their event loop from the initial process systemd starts for you. Also, don't call `setsid()`.
- Don't drop user privileges in the daemon itself, leave this to systemd and configure it in systemd service configuration files. (There are exceptions here. For example, for some daemons there are good reasons to drop privileges inside the daemon code, after an initialization phase that requires elevated privileges.)
- Don't write PID files
- Grab a name on the bus
- You may rely on systemd for logging, you are welcome to log whatever you need to log to stderr.
- Let systemd create and watch sockets for you, so that socket activation works. Hence, interpret `$LISTEN_FDS` and `$LISTEN_PID` as described above.
- Use SIGTERM for requesting shut downs from your daemon.

The list above is very similar to what Apple recommends for daemons compatible with launchd. It should be easy to extend daemons that already support launchd activation to support systemd activation as well.

Note that systemd supports daemons not written in this style perfectly as well, already for compatibility reasons (launchd has only limited support for that). As mentioned, this even extends to existing inetd capable daemons which can be used unmodified for socket activation by systemd.

So, yes, should systemd prove itself in our experiments and get adopted by the distributions it would make sense to port at least those services that are started by default to use socket or bus-based activation. We have written proof-of-concept patches, and the porting turned out to be very easy. Also, we can leverage the work that has already been done for launchd, to a certain extent. Moreover, adding support for socket-based activation does not make the service incompatible with non-systemd systems.

FAQs

Who's behind this?

> Well, the current code-base is mostly my work, Lennart Poettering (Red Hat). However the design in all its details is result of close cooperation between Kay Sievers (Novell) and me. Other people involved are Harald Hoyer (Red Hat), Dhaval Giani (Formerly IBM), and a few others from various companies such as Intel, SUSE and Nokia.

Is this a Red Hat project?

> No, this is my personal side project. Also, let me emphasize this: *the opinions reflected here are my own. They are not the views of my employer, or Ronald McDonald, or anyone else.*

Will this come to Fedora?

> If our experiments prove that this approach works out, and discussions in the Fedora community show support for this, then yes, we'll certainly try to get this into Fedora.

Will this come to OpenSUSE?

> Kay's pursuing that, so something similar as for Fedora applies here, too.

Will this come to Debian/Gentoo/Mandriva/MeeGo/Ubuntu/[insert your favourite distro here]?

> That's up to them. We'd certainly welcome their interest, and help with the integration.

Why didn't you just add this to Upstart, why did you invent something new?

> Well, the point of the part about Upstart above was to show that the core design of Upstart is flawed, in our opinion. Starting completely from scratch suggests itself if the existing solution appears flawed in its core. However, note that we took a lot of inspiration from Upstart's code-base otherwise.

If you love Apple launchd so much, why not adopt that?

launchd is a great invention, but I am not convinced that it would fit well into Linux, nor that it is suitable for a system like Linux with its immense scalability and flexibility to numerous purposes and uses.

Is this an NIH project?

Well, I hope that I managed to explain in the text above why we came up with something new, instead of building on Upstart or launchd. We came up with systemd due to technical reasons, not political reasons.

Don't forget that it is Upstart that includes a library called NIH (which is kind of a reimplementation of glib) -- not systemd!

Will this run on [insert non-Linux OS here]?

Unlikely. As pointed out, systemd uses many Linux specific APIs (such as epoll, signalfd, libudev, cgroups, and numerous more), a port to other operating systems appears to us as not making a lot of sense. Also, we, the people involved are unlikely to be interested in merging possible ports to other platforms and work with the constraints this introduces. That said, git supports branches and rebasing quite well, in case people really want to do a port.

Actually portability is even more limited than just to other OSes: we require a very recent Linux kernel, glibc, libcgroup and libudev. No support for less-than-current Linux systems, sorry.

If folks want to implement something similar for other operating systems, the preferred mode of cooperation is probably that we help you identify which interfaces can be shared with your system, to make life easier for daemon writers to support both systemd and your systemd counterpart. Probably, the focus should be to share interfaces, not code.

I hear [fill one in here: the Gentoo boot system, initng, Solaris SMF, runit, uxlaunch, ...] is an awesome init system and also does parallel boot-up, so why not adopt that?

Well, before we started this we actually had a very close look at the various systems, and none of them did what we had in mind for systemd (with the exception of launchd, of course). If you cannot see that, then please read again what I wrote above.

Contributions

We are very interested in patches and help. It should be common sense that every Free Software project can only benefit from the widest possible external contributions. That is particularly true for a core part of the OS, such as an init system. We value your contributions and hence do not require copyright assignment (Very much unlike Canonical/Upstart!). And also, we use git, everybody's favourite VCS, yay!

We are particularly interested in help getting systemd to work on other distributions, besides Fedora and OpenSUSE. (Hey, anybody from Debian, Gentoo, Mandriva, MeeGo looking for something to do?) But even beyond that we are keen to attract contributors on every level: we welcome C hackers, packagers, as well as folks who are interested to write documentation, or contribute a logo.

Community

At this time we only have source code repository and an IRC channel (`#systemd` on Freenode). There's no mailing list, web site or bug tracking system. We'll probably set something up on freedesktop.org soon. If you have any questions or want to contact us otherwise we invite you to join us on IRC!

Update: our GIT repository has moved.

Category: projects

← BACK TO INDEX