Rocky Enterprise Linux 9.2 Manual Pages on command 'daemon.7'

C:\>man daemon.7

DAEMON(7) daemon DAEMON(7)

NAME

daemon - Writing and packaging system daemons

DESCRIPTION

A daemon is a service process that runs in the background and supervises the system or provides functionality to other processes. Traditionally, daemons are implemented following a scheme originating in SysV Unix. Modern daemons should follow a simpler yet more powerful scheme (here called "new-style" daemons), as implemented by systemd(1). This manual page covers both schemes, and in particular includes recommendations for daemons that shall be included in the systemd init system.

SysV Daemons

When a traditional SysV daemon starts, it should execute the following steps as part of the initialization. Note that these steps are unnecessary for new-style daemons (see below), and should only be implemented if compatibility with SysV is essential.

1. Close all open file descriptors except standard input, output, and error (i.e. the first three file descriptors 0, 1, 2). This ensures that no accidentally passed file descriptor stays around in the daemon process. On Linux, this is best implemented by iterating through /proc/self/fd, with a fallback of iterating from file descriptor 3 to the value returned by getrlimit() for RLIMIT_NOFILE.

2. Reset all signal handlers to their default. This is best done by iterating through the available signals up to the limit of _NSIG and resetting them to SIG_DFL.
3. Reset the signal mask using sigprocmask().
4. Sanitize the environment block, removing or resetting environment variables that might negatively impact daemon runtime.
5. Call fork(), to create a background process.
6. In the child, call setsid() to detach from any terminal and create an independent session.
7. In the child, call fork() again, to ensure that the daemon can never re-acquire a terminal again. (This relevant if the program ? and all its dependencies ? does not carefully specify `O_NOCTTY` on each and every single `open()` call that might potentially open a TTY device node.)
8. Call exit() in the first child, so that only the second child (the actual daemon process) stays around. This ensures that the daemon process is re-parented to init/PID 1, as all daemons should be.
9. In the daemon process, connect /dev/null to standard input, output, and error.
10. In the daemon process, reset the umask to 0, so that the file modes passed to open(), mkdir() and suchlike directly control the access mode of the created files and directories.
11. In the daemon process, change the current directory to the root directory (/), in order to avoid that the daemon involuntarily blocks mount points from being unmounted.
12. In the daemon process, write the daemon PID (as returned by getpid()) to a PID file, for example /run/foobar.pid (for a hypothetical daemon "foobar") to ensure that the daemon cannot be started more than once. This must be implemented in race-free fashion so that the PID file is only updated when it is verified at the same time that the PID previously stored in the PID file no longer exists or belongs to a foreign process.
13. In the daemon process, drop privileges, if possible and applicable.
14. From the daemon process, notify the original process started that initialization is complete. This can be implemented via an unnamed pipe or similar communication channel that is created before the first fork() and hence

available in both the original and the daemon process.

15. Call exit() in the original process. The process that invoked the daemon must be able to rely on that this exit() happens after initialization is complete and all external communication channels are established and accessible.

The BSD daemon() function should not be used, as it implements only a subset of these steps.

A daemon that needs to provide compatibility with SysV systems should implement the scheme pointed out above. However, it is recommended to make this behavior optional and configurable via a command line argument to ease debugging as well as to simplify integration into systems using systemd.

New-Style Daemons

Modern services for Linux should be implemented as new-style daemons. This makes it easier to supervise and control them at runtime and simplifies their implementation.

For developing a new-style daemon, none of the initialization steps recommended for SysV daemons need to be implemented. New-style init systems such as systemd make all of them redundant. Moreover, since some of these steps interfere with process monitoring, file descriptor passing and other functionality of the init system, it is recommended not to execute them when run as new-style service.

Note that new-style init systems guarantee execution of daemon processes in a clean process context: it is guaranteed that the environment block is sanitized, that the signal handlers and mask is reset and that no left-over file descriptors are passed. Daemons will be executed in their own session, with standard input connected to /dev/null and standard output/error connected to the systemd-journald.service(8) logging service, unless otherwise configured. The umask is reset.

It is recommended for new-style daemons to implement the following:

1. If SIGTERM is received, shut down the daemon and exit cleanly.
2. If SIGHUP is received, reload the configuration files, if this applies.
3. Provide a correct exit code from the main daemon process, as this is used by the init system to detect service errors and problems. It is recommended to follow the exit code scheme as defined in the LSB recommendations for SysV init scripts[1].

4. If possible and applicable, expose the daemon's control interface via the D-Bus IPC system and grab a bus name as last step of initialization.
5. For integration in systemd, provide a .service unit file that carries information about starting, stopping and otherwise maintaining the daemon. See systemd.service(5) for details.
6. As much as possible, rely on the init system's functionality to limit the access of the daemon to files, services and other resources, i.e. in the case of systemd, rely on systemd's resource limit control instead of implementing your own, rely on systemd's privilege dropping code instead of implementing it in the daemon, and similar. See systemd.exec(5) for the available controls.
7. If D-Bus is used, make your daemon bus-activatable by supplying a D-Bus service activation configuration file. This has multiple advantages: your daemon may be started lazily on-demand; it may be started in parallel to other daemons requiring it ? which maximizes parallelization and boot-up speed; your daemon can be restarted on failure without losing any bus requests, as the bus queues requests for activatable services. See below for details.
8. If your daemon provides services to other local processes or remote clients via a socket, it should be made socket-activatable following the scheme pointed out below. Like D-Bus activation, this enables on-demand starting of services as well as it allows improved parallelization of service start-up. Also, for state-less protocols (such as syslog, DNS), a daemon implementing socket-based activation can be restarted without losing a single request. See below for details.
9. If applicable, a daemon should notify the init system about startup completion or status updates via the sd_notify(3) interface.
10. Instead of using the syslog() call to log directly to the system syslog service, a new-style daemon may choose to simply log to standard error via fprintf(), which is then forwarded to syslog by the init system. If log levels are necessary, these can be encoded by prefixing individual log lines with strings like "<4>" (for log level 4 "WARNING" in the syslog priority scheme), following a similar style as the Linux kernel's printk() level system. For details, see sd-daemon(3) and systemd.exec(5).
11. As new-style daemons are invoked without a controlling TTY (but as their own

session leaders) care should be taken to always specify `O_NOCTTY` on `open()` calls that possibly reference a TTY device node, so that no controlling TTY is accidentally acquired.

These recommendations are similar but not identical to the Apple MacOS X Daemon Requirements[2].

ACTIVATION

New-style init systems provide multiple additional mechanisms to activate services, as detailed below. It is common that services are configured to be activated via more than one mechanism at the same time. An example for systemd: bluetoothd.service might get activated either when Bluetooth hardware is plugged in, or when an application accesses its programming interfaces via D-Bus. Or, a print server daemon might get activated when traffic arrives at an IPP port, or when a printer is plugged in, or when a file is queued in the printer spool directory. Even for services that are intended to be started on system bootup unconditionally, it is a good idea to implement some of the various activation schemes outlined below, in order to maximize parallelization. If a daemon implements a D-Bus service or listening socket, implementing the full bus and socket activation scheme allows starting of the daemon with its clients in parallel (which speeds up boot-up), since all its communication channels are established already, and no request is lost because client requests will be queued by the bus system (in case of D-Bus) or the kernel (in case of sockets) until the activation is completed.

Activation on Boot

Old-style daemons are usually activated exclusively on boot (and manually by the administrator) via SysV init scripts, as detailed in the LSB Linux Standard Base Core Specification[1]. This method of activation is supported ubiquitously on Linux init systems, both old-style and new-style systems. Among other issues, SysV init scripts have the disadvantage of involving shell scripts in the boot process.

New-style init systems generally employ updated versions of activation, both during boot-up and during runtime and using more minimal service description files.

In systemd, if the developer or administrator wants to make sure that a service or other unit is activated automatically on boot, it is recommended to place a symlink to the unit file in the .wants/ directory of either multi-user.target or

graphical.target, which are normally used as boot targets at system startup. See systemd.unit(5) for details about the .wants/ directories, and systemd.special(7) for details about the two boot targets.

Socket-Based Activation

In order to maximize the possible parallelization and robustness and simplify configuration and development, it is recommended for all new-style daemons that communicate via listening sockets to employ socket-based activation. In a socket-based activation scheme, the creation and binding of the listening socket as primary communication channel of daemons to local (and sometimes remote) clients is moved out of the daemon code and into the init system. Based on per-daemon configuration, the init system installs the sockets and then hands them off to the spawned process as soon as the respective daemon is to be started. Optionally, activation of the service can be delayed until the first inbound traffic arrives at the socket to implement on-demand activation of daemons. However, the primary advantage of this scheme is that all providers and all consumers of the sockets can be started in parallel as soon as all sockets are established. In addition to that, daemons can be restarted with losing only a minimal number of client transactions, or even any client request at all (the latter is particularly true for state-less protocols, such as DNS or syslog), because the socket stays bound and accessible during the restart, and all requests are queued while the daemon cannot process them.

New-style daemons which support socket activation must be able to receive their sockets from the init system instead of creating and binding them themselves. For details about the programming interfaces for this scheme provided by systemd, see sd_listen_fds(3) and sd-daemon(3). For details about porting existing daemons to socket-based activation, see below. With minimal effort, it is possible to implement socket-based activation in addition to traditional internal socket creation in the same codebase in order to support both new-style and old-style init systems from the same daemon binary.

systemd implements socket-based activation via .socket units, which are described in systemd.socket(5). When configuring socket units for socket-based activation, it is essential that all listening sockets are pulled in by the special target unit sockets.target. It is recommended to place a WantedBy=sockets.target directive in

the "[Install]" section to automatically add such a dependency on installation of a socket unit. Unless DefaultDependencies=no is set, the necessary ordering dependencies are implicitly created for all socket units. For more information about sockets.target, see systemd.special(7). It is not necessary or recommended to place any additional dependencies on socket units (for example from multi-user.target or suchlike) when one is installed in sockets.target.

Bus-Based Activation

When the D-Bus IPC system is used for communication with clients, new-style daemons should employ bus activation so that they are automatically activated when a client application accesses their IPC interfaces. This is configured in D-Bus service files (not to be confused with systemd service unit files!). To ensure that D-Bus uses systemd to start-up and maintain the daemon, use the SystemdService= directive in these service files to configure the matching systemd service for a D-Bus service. e.g.: For a D-Bus service whose D-Bus activation file is named org.freedesktop.RealtimeKit.service, make sure to set SystemdService=rtkit-daemon.service in that file to bind it to the systemd service rtkit-daemon.service. This is needed to make sure that the daemon is started in a race-free fashion when activated via multiple mechanisms simultaneously.

Device-Based Activation

Often, daemons that manage a particular type of hardware should be activated only when the hardware of the respective kind is plugged in or otherwise becomes available. In a new-style init system, it is possible to bind activation to hardware plug/unplug events. In systemd, kernel devices appearing in the sysfs/udev device tree can be exposed as units if they are tagged with the string "systemd". Like any other kind of unit, they may then pull in other units when activated (i.e. plugged in) and thus implement device-based activation. systemd dependencies may be encoded in the udev database via the SYSTEMD_WANTS= property. See systemd.device(5) for details. Often, it is nicer to pull in services from devices only indirectly via dedicated targets. Example: Instead of pulling in bluetoothd.service from all the various bluetooth dongles and other hardware available, pull in bluetooth.target from them and bluetoothd.service from that target. This provides for nicer abstraction and gives administrators the option to enable bluetoothd.service via controlling a bluetooth.target.wants/ symlink uniformly with

a command like enable of systemctl(1) instead of manipulating the udev ruleset.

Path-Based Activation

Often, runtime of daemons processing spool files or directories (such as a printing system) can be delayed until these file system objects change state, or become non-empty. New-style init systems provide a way to bind service activation to file system changes. systemd implements this scheme via path-based activation configured in .path units, as outlined in systemd.path(5).

Timer-Based Activation

Some daemons that implement clean-up jobs that are intended to be executed in regular intervals benefit from timer-based activation. In systemd, this is implemented via .timer units, as described in systemd.timer(5).

Other Forms of Activation

Other forms of activation have been suggested and implemented in some systems. However, there are often simpler or better alternatives, or they can be put together of combinations of the schemes above. Example: Sometimes, it appears useful to start daemons or .socket units when a specific IP address is configured on a network interface, because network sockets shall be bound to the address. However, an alternative to implement this is by utilizing the Linux IP_FREEBIND socket option, as accessible via FreeBind=yes in systemd socket files (see systemd.socket(5) for details). This option, when enabled, allows sockets to be bound to a non-local, not configured IP address, and hence allows bindings to a particular IP address before it actually becomes available, making such an explicit dependency to the configured address redundant. Another often suggested trigger for service activation is low system load. However, here too, a more convincing approach might be to make proper use of features of the operating system, in particular, the CPU or I/O scheduler of Linux. Instead of scheduling jobs from userspace based on monitoring the OS scheduler, it is advisable to leave the scheduling of processes to the OS scheduler itself. systemd provides fine-grained access to the CPU and I/O schedulers. If a process executed by the init system shall not negatively impact the amount of CPU or I/O bandwidth available to other processes, it should be configured with CPUSchedulingPolicy=idle and/or IOSchedulingClass=idle. Optionally, this may be combined with timer-based activation to schedule background jobs during runtime and with minimal impact on

the system, and remove it from the boot phase itself.

INTEGRATION WITH SYSTEMD

Writing systemd Unit Files

When writing systemd unit files, it is recommended to consider the following suggestions:

1. If possible, do not use the Type=forking setting in service files. But if you do, make sure to set the PID file path using PIDFile=. See systemd.service(5) for details.
2. If your daemon registers a D-Bus name on the bus, make sure to use Type=dbus in the service file if possible.
3. Make sure to set a good human-readable description string with Description=.
4. Do not disable DefaultDependencies=, unless you really know what you do and your unit is involved in early boot or late system shutdown.
5. Normally, little if any dependencies should need to be defined explicitly. However, if you do configure explicit dependencies, only refer to unit names listed on systemd.special(7) or names introduced by your own package to keep the unit file operating system-independent.
6. Make sure to include an "[Install]" section including installation information for the unit file. See systemd.unit(5) for details. To activate your service on boot, make sure to add a WantedBy=multi-user.target or WantedBy=graphical.target directive. To activate your socket on boot, make sure to add WantedBy=sockets.target. Usually, you also want to make sure that when your service is installed, your socket is installed too, hence add Also=foo.socket in your service file foo.service, for a hypothetical program foo.

Installing systemd Service Files

At the build installation time (e.g. make install during package build), packages are recommended to install their systemd unit files in the directory returned by pkg-config systemd --variable=systemdsystemunitdir (for system services) or pkg-config systemd --variable=systemduserunitdir (for user services). This will make the services available in the system on explicit request but not activate them automatically during boot. Optionally, during package installation (e.g. rpm -i by the administrator), symlinks should be created in the systemd configuration

directories via the enable command of the systemctl(1) tool to activate them
automatically on boot.
Packages using autoconf(1) are recommended to use a configure script excerpt like
the following to determine the unit installation path during source configuration:

```
PKG_PROG_PKG_CONFIG
AC_ARG_WITH([systemdsystemunitdir],
     [AS_HELP_STRING([--with-systemdsystemunitdir=DIR], [Directory for systemd service files])],,
     [with_systemdsystemunitdir=auto])
AS_IF([test "x$with_systemdsystemunitdir" = "xyes" -o "x$with_systemdsystemunitdir" = "xauto"], [
     def_systemdsystemunitdir=$($PKG_CONFIG --variable=systemdsystemunitdir systemd)
     AS_IF([test "x$def_systemdsystemunitdir" = "x"],
    [AS_IF([test "x$with_systemdsystemunitdir" = "xyes"],
     [AC_MSG_ERROR([systemd support requested but pkg-config unable to query systemd package])])
     with_systemdsystemunitdir=no],
    [with_systemdsystemunitdir="$def_systemdsystemunitdir"])])
AS_IF([test "x$with_systemdsystemunitdir" != "xno"],
      [AC_SUBST([systemdsystemunitdir], [$with_systemdsystemunitdir])])
AM_CONDITIONAL([HAVE_SYSTEMD], [test "x$with_systemdsystemunitdir" != "xno"])
```

This snippet allows automatic installation of the unit files on systemd machines,
and optionally allows their installation even on machines lacking systemd.
(Modification of this snippet for the user unit directory is left as an exercise
for the reader.)
Additionally, to ensure that make distcheck continues to work, it is recommended to
add the following to the top-level Makefile.am file in automake(1)-based projects:

```
AM_DISTCHECK_CONFIGURE_FLAGS = \
   --with-systemdsystemunitdir=$$dc_install_base/$(systemdsystemunitdir)
```

Finally, unit files should be installed in the system with an automake excerpt like
the following:

```
if HAVE_SYSTEMD
systemdsystemunit_DATA = \
  foobar.socket \
  foobar.service
endif
```

In the rpm(8) .spec file, use snippets like the following to enable/disable the service during installation/deinstallation. This makes use of the RPM macros shipped along systemd. Consult the packaging guidelines of your distribution for details and the equivalent for other package managers.

At the top of the file:

```
BuildRequires: systemd
%{?systemd_requires}
```

And as scriptlets, further down:

```
%post
%systemd_post foobar.service foobar.socket
%preun
%systemd_preun foobar.service foobar.socket
%postun
%systemd_postun
```

If the service shall be restarted during upgrades, replace the "%postun" scriptlet above with the following:

```
%postun
%systemd_postun_with_restart foobar.service
```

Note that "%systemd_post" and "%systemd_preun" expect the names of all units that are installed/removed as arguments, separated by spaces. "%systemd_postun" expects no arguments. "%systemd_postun_with_restart" expects the units to restart as arguments.

To facilitate upgrades from a package version that shipped only SysV init scripts to a package version that ships both a SysV init script and a native systemd service file, use a fragment like the following:

```
%triggerun -- foobar < 0.47.11-1
if /sbin/chkconfig --level 5 foobar ; then
  /bin/systemctl --no-reload enable foobar.service foobar.socket >/dev/null 2>&1 || :
fi
```

Where 0.47.11-1 is the first package version that includes the native unit file. This fragment will ensure that the first time the unit file is installed, it will be enabled if and only if the SysV init script is enabled, thus making sure that the enable status is not changed. Note that chkconfig is a command specific to

Fedora which can be used to check whether a SysV init script is enabled. Other operating systems will have to use different commands here.

PORTING EXISTING DAEMONS

Since new-style init systems such as systemd are compatible with traditional SysV init systems, it is not strictly necessary to port existing daemons to the new style. However, doing so offers additional functionality to the daemons as well as simplifying integration into new-style init systems.

To port an existing SysV compatible daemon, the following steps are recommended:

1. If not already implemented, add an optional command line switch to the daemon to disable daemonization. This is useful not only for using the daemon in new-style init systems, but also to ease debugging.
2. If the daemon offers interfaces to other software running on the local system via local AF_UNIX sockets, consider implementing socket-based activation (see above). Usually, a minimal patch is sufficient to implement this: Extend the socket creation in the daemon code so that sd_listen_fds(3) is checked for already passed sockets first. If sockets are passed (i.e. when sd_listen_fds() returns a positive value), skip the socket creation step and use the passed sockets. Secondly, ensure that the file system socket nodes for local AF_UNIX sockets used in the socket-based activation are not removed when the daemon shuts down, if sockets have been passed. Third, if the daemon normally closes all remaining open file descriptors as part of its initialization, the sockets passed from the init system must be spared. Since new-style init systems guarantee that no left-over file descriptors are passed to executed processes, it might be a good choice to simply skip the closing of all remaining open file descriptors if sockets are passed.
3. Write and install a systemd unit file for the service (and the sockets if socket-based activation is used, as well as a path unit file, if the daemon processes a spool directory), see above for details.
4. If the daemon exposes interfaces via D-Bus, write and install a D-Bus activation file for the service, see above for details.

PLACING DAEMON DATA

It is recommended to follow the general guidelines for placing package files, as discussed in file-hierarchy(7).

SEE ALSO

systemd(1), sd-daemon(3), sd_listen_fds(3), sd_notify(3), daemon(3),

systemd.service(5), file-hierarchy(7)

NOTES

1. LSB recommendations for SysV init scripts

   http://refspecs.linuxbase.org/LSB_3.1.1/LSB-Core-generic/LSB-Core-generic/iniscrptact.html

2. Apple MacOS X Daemon Requirements

https://developer.apple.com/library/mac/documentation/MacOSX/Conceptual/BPSystemStartup/Chapters/CreatingLaunchdJobs.html