Cause a process to become a daemon

Content

Objective

To cause a process to become a daemon

Background

A daemon is a type of background process that is not associated with any particular terminal. On POSIX-based systems there are specific steps that a process should perform in order to become a daemon.

These instructions describe what needs to be done in language-independent terms. For specific languages see:

- Cause a process to become a daemon in C

Note that programs invoked by `inetd` need not and should not daemonise themselves because they will inherit the required characteristics from `inetd`.

Scenario

Suppose that you are writing a daemon called `exampled`. When invoked from the command line it should spawn a background process then return control to the user.

Method

Overview

The standard procedure for daemonising a process has six steps:

1. Fork, allowing the parent process to terminate.
2. Start a new session for the daemon by calling `setsid`.
3. Fork again, allowing the parent process to terminate.
4. Change the current working directory to a safe location.
5. Set the `umask` to zero.
6. Close then reopen `stdin`, `stdout` and `stderr`.

Details of each required step are given below. Alternatively, some environments have a library function that does all of the necessary work.

Fork, allowing the parent process to terminate

Calling `fork` briefly splits the program into two processes. Normally the parent then terminates immediately, leaving only the child to continue running. This achieves two things:

- It unblocks any grandparent process that may be waiting for the parent process to terminate. (For example, if the daemon was invoked from the shell then this returns control to the user.)
- It ensures that the child process is not a process group leader (required for the call to `setsid` performed below).

It is not necessary for the child process to wait for the parent to terminate.

Start a new session for the daemon by calling setsid

When a user logs out from a session, all processes associated with that session are killed. For processes that are daemons you do not want this to happen. The solution is to call `setsid`. Provided that the daemon is not already a process group leader (which it will not be following the fork performed above), this will:

- start a new session, with the daemon as session leader and with no controlling terminal, and
- start a new process group, with the daemon as process group leader.

Fork again, allowing the parent process to terminate

If a session leader has no controlling terminal and it opens a terminal device then that device automatically becomes the controlling terminal. Again, this is something that you do not want to happen. The solution is to perform a second fork, again allowing the parent process to terminate. This ensures that the child process is not a session leader.

Before forking it is necessary to ignore `SIGHUP`. This prevents the child from being killed when the parent (which is the session leader) dies.

It is not necessary for the child process to wait for the parent to terminate, however it is very important that the `SIGHUP` handler remain in place until at least one signal has been received. If you wish to use `SIGHUP` for some other purpose once the daemon is running then your options include:

- having two handler functions, one to handle the first `SIGHUP` by switching control to the second

function, or
- having a single handler function, but setting a flag to change its behaviour once the first `SIGHUP` has been ignored.

Change the current working directory to a safe location

If a daemon were to leave its current working directory unchanged then this would prevent the filesystem containing that directory from being unmounted while the daemon was running. It is therefore good practice for daemons to change their working directory to a safe location. This could be:

- the root directory, or
- a directory containing files that are needed by the daemon.

(The latter option turns what would have been a drawback into a feature. If files are needed by the daemon then the filesystem containing them presumably should not be unmounted while the daemon is running. Moving the current working directory to that filesystem prevents this from happening.)

Set the umask to zero

The purpose of the `umask` is to allow users to influence the permissions given to newly created files and directories. Daemons should not allow themselves to be affected by this setting, because what was appropriate for the user will not necessarily be suitable for the daemon.

In some cases it may be more convenient for the `umask` to be set to a non-zero value. This is equally acceptable: the important point is that the daemon has taken control of the value, as opposed to merely accepting what it was given.

Close then reopen stdin, stdout and stderr

Once it is running a daemon should not read from or write to the terminal from which it was launched. The simplest and most effective way to ensure this is to close the file descriptors corresponding to `stdin`, `stdout` and `stderr`. These should then be reopened, either to `/dev/null`, or if preferred to some other location. There are two reasons for not leaving them closed:

- to prevent code that refers to these file descriptors from failing, and
- to prevent the descriptors from being reused for some other purpose.

For example, it is not unusual for code to assume that warning and error messages can be written to `stderr` at any time. If file descriptor 2 were reused to access a database file or a block special device then the outcome could be unfortunate.

To prevent such messages from being lost entirely it is common for `stdout` and `stderr` to be redirected to a log file instead of `/dev/null`. The safest procedure for doing this is as follows:

1. Open the logfile.
2. Close `stdout` and `stderr`.
3. Use `dup2` to redirect `stdout` and `stderr` to the logfile.
4. Close the file descriptor that was initially associated with the logfile.

This ensures that any failure to open the logfile occurs while `stderr` is still connected to the terminal,

allowing the problem to be reported.

As always when performing a fork, you should check that no file descriptors have been left open unnecessarily. It is acceptable for descriptors to remain open for later use: as above, this allows errors that prevent them from being opened to be reported to the terminal.

Testing

Check that the daemon has no controlling terminal and is not a session leader

The controlling terminal and session ID of a process can be inspected using the `ps` command:

```
ps -o pid,sid,tty,cmd -C exampled
```

This should output a table of the form:

```
  PID   SID TT       CMD
29964 29961 ?        /usr/local/bin/exampled
```

A question mark in the `TT` column indicates that there is no controlling terminal. (If there was one then the device name would appear here instead.)

The session ID of a process is (by definition) equal to the process ID of the session leader. Therefore, if the numbers in the PID and SID columns are different (as they are here) then the process is not a session leader.

Check the current working directory

See Inspect the current working directory of a running process.

Check the umask

See Inspect the umask of a running process.

Check file descriptors

The file descriptors of a running process can be inspected by looking in `/proc`. For example, if the PID were 29964:

```
ls -l /proc/29964/fd
```

Each open descriptor is presented as a softlink. If a descriptor is associated with a filesystem object then the target of the softlink is the pathname of the object. For a daemon the output would typically be similar to:

```
lrwx------ 1 root root 64 2011-02-08 06:40 0 -> /dev/null
lrwx------ 1 root root 64 2011-02-08 06:40 1 -> /dev/null
lrwx------ 1 root root 64 2011-02-08 06:40 2 -> /dev/null
```

Check that you see the expected number of file descriptors and that they are associated with the appropriate pathnames.

Tags: posix | process