Cause a process to become a daemon in C

Content

Tested on

Debian (Etch, Lenny, Squeeze)

Fedora (14)

Ubuntu (Hardy, Intrepid, Jaunty, Karmic, Lucid, Maverick, Natty, Precise, Trusty)

Objective

To cause a process to become a daemon in C

Background and Scenario

See Cause a process to become a daemon. That page also gives a more detailed rationale for the method, which is explained only in outline here.

A mechanism is needed for handling errors. The example code shown below assumes that there is a function called `die` provided for this purpose, which takes the same arguments as `printf` and does not return.

Method

Fork, allowing the parent process to terminate

Calling `fork` has three possible types of return value:

- -1 indicates failure (most likely due to lack of memory, although it is possible to run out of other resources such as PIDs).
- 0 indicates that the child is running, in which case execution should continue with the next step of the daemonisation process.

- Any other value indicates that the parent is running, in which case the process should terminate by calling `_exit`.

```
pid_t pid = fork();
if (pid == -1) {
    die("failed to fork while daemonising (errno=%d)",errno);
} else if (pid != 0) {
    _exit(0);
}
```

Start a new session for the daemon by calling setsid

This operation should never fail, because the current process should not now be a process group leader, however we check anyhow as a precaution:

```
if (setsid()==-1) {
    die("failed to become a session leader while daemonising(errno=%d)",errno);
}
```

Fork again, allowing the parent process to terminate

This is a repeat of the first step, except that a handler must be installed for SIGHUP:

```
signal(SIGHUP,SIG_IGN);
pid=fork();
if (pid == -1) {
    die("failed to fork while daemonising (errno=%d)",errno);
} else if (pid != 0) {
    _exit(0);
}
```

The `SIGHUP` handler must remain in place until it has absorbed the `SIGHUP` that the parent is expected to send when it terminates. See below if you wish to install a `SIGHUP` handler for other purposes.

Change the current working directory to a safe location

The root directory is used here, as it is always a safe location and can be changed later if required:

```
if (chdir("/") == -1) {
    die("failed to change working directory while daemonising (errno=%d)",errno);
}
```

Set the umask to zero

Daemons normally operate with a `umask` of zero. Again, this can be changed later if required:

```
umask(0);
```

Close then reopen stdin, stdout and stderr

The POSIX specification requires that `/dev/null` be provided, therefore the daemon can reasonably depend on this device being available provided that they fail gracefully if it is not.

When `stderr` is opened it must be both readable and writable. It is sufficient for `stdin` to be readable and `stdout` to be writable. If `stdout` or `stderr` refer to a regular file then they should be configured to append to it (by means of the O_APPEND flag). Because the `open` function always chooses the lowest unused file descriptor, by reopening the streams in ascending order it is possible to avoid the use of `dup2`:

```
close(STDIN_FILENO);
close(STDOUT_FILENO);
close(STDERR_FILENO);
if (open("/dev/null",O_RDONLY) == -1) {
        die("failed to reopen stdin while daemonising (errno=%d)",errno);
}
if (open("/dev/null",O_WRONLY) == -1) {
        die("failed to reopen stdout while daemonising (errno=%d)",errno);
}
if (open("/dev/null",O_RDWR) == -1) {
        die("failed to reopen stderr while daemonising (errno=%d)",errno);
}
```

See below if you want to direct `stdout` and `stderr` to a logfile.

The complete method as a function

```
#include <errno.h>
#include <signal.h>
#include <fcntl.h>
#include <unistd.h>

void daemonise() {
    // Fork, allowing the parent process to terminate.
    pid_t pid = fork();
    if (pid == -1) {
        die("failed to fork while daemonising (errno=%d)",errno);
    } else if (pid != 0) {
        _exit(0);
    }

    // Start a new session for the daemon.
    if (setsid()==-1) {
        die("failed to become a session leader while daemonising(errno=%d)",errno);
    }

    // Fork again, allowing the parent process to terminate.
    signal(SIGHUP,SIG_IGN);
    pid=fork();
    if (pid == -1) {
        die("failed to fork while daemonising (errno=%d)",errno);
    } else if (pid != 0) {
        _exit(0);
    }

    // Set the current working directory to the root directory.
    if (chdir("/") == -1) {
        die("failed to change working directory while daemonising (errno=%d)",errno);
    }

    // Set the user file creation mask to zero.
    umask(0);

    // Close then reopen standard file descriptors.
    close(STDIN_FILENO);
    close(STDOUT_FILENO);
    close(STDERR_FILENO);
    if (open("/dev/null",O_RDONLY) == -1) {
        die("failed to reopen stdin while daemonising (errno=%d)",errno);
    }
    if (open("/dev/null",O_WRONLY) == -1) {
        die("failed to reopen stdout while daemonising (errno=%d)",errno);
    }
    if (open("/dev/null",O_RDWR) == -1) {
        die("failed to reopen stderr while daemonising (errno=%d)",errno);
    }
}
```

Testing

See Cause a process to become a daemon.

Variations

Redirect stdout and stderr to a logfile

When directing output to a logfile, it is best to open the file before closing `stderr` to ensure that the daemon is not left with no means of reporting errors:

```
close(STDIN_FILENO);
if (open("/dev/null",O_RDONLY) == -1) {
    die("failed to reopen stdin while daemonising (errno=%d)",errno);
}
int logfile_fileno = open(logfile_pathname,O_RDWR|O_CREAT|O_APPEND,S_IRUSR|S_IWUSR|S_IRGRP);
if (logfile_fileno == -1) {
    die("failed to open logfile (errno=%d)",errno);
}
dup2(logfile_fileno,STDOUT_FILENO);
dup2(logfile_fileno,STDERR_FILENO);
close(logfile_fileno);
```

Note that `dup2` will close the target file descriptor if necessary, so there is no need to do this explicitly.

Using SIGHUP for other purposes

Daemons often interpret `SIGHUP` as a request to reread the configuration file. A signal handler must be installed to perform this function, however it must not become fully active until after the parent process of the second fork operation has terminated (as that event will generate a `SIGHUP`).

One solution is to use a flag within the handler function to treat the first call differently:

```
void handle_sighup(int signum) {
    static bool first=true;
    if (first) {
        first=false;
        return;
    }
    // Insert remainder of handler here.
}
```

When installing the signal handler, it is better to use `sigaction` in preference to the `signal` function because that allows the `SA_RESTART` flag to be used. Without this, it is necessary to place a loop around any system function that is capable of returning `EINTR`:

```
struct sigaction sa;
sa.sa_handler = handle_sighup;
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_RESTART;
if (sigaction(SIGHUP,&sa,0) == -1) {
    die("failed to install SIGHUP handler (errno=%d)",errno);
}
```

Methods to avoid

Use the daemon function

Many POSIX-based operating systems provide a function called `daemon` which performs some or all of the steps listed above. Unfortunately it has three significant drawbacks:

- It is not available on all systems.
- Its behaviour is not standardised (or necessarily well-documented).
- Its behaviour is more difficult to customise.

For these reasons, any benefit gained by using the `daemon` function is likely to be a short-term one at best.

Tags: c | posix | process