Ignore SIGPIPE without affecting other threads in a process

Content

Tested on

Debian (Etch, Lenny, Squeeze)

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

Objective

To disregard SIGPIPE within a given thread without affect other threads in the same process

Background

If you attempt to write to a broken pipe or a disconnected socket then the signal SIGPIPE will be raised, and its default behaviour is to terminate the process responsible. This behaviour is appropriate for programs that are intended for use in a pipeline, but in most other circumstances it is undesirable and should be prevented.

It is a straightforward matter to ignore SIGPIPE by setting its disposition to SIG_IGN. Historically this would have been done using the signal function, but that is now deprecated in favour of sigaction:

```
struct sigaction sa;
sa.sa_handler = SIG_IGN;
sa.sa_flags = 0;
if (sigaction(SIGPIPE, &sa, 0) == -1) {
  perror("sigaction");
  exit(1);
}
```

There is no corresponding function to change the action for a single thread because all threads in a process share the same set of actions. This makes it inadvisable to change signal dispositions in library code (even temporarily) because the effect cannot be properly encapsulated:

- The library could find itself handling signals it has no interest in.
- The program using the library may fail to see signals that it was expecting.

- If two threads independently attempt to modify then restore the disposition of a signal, there is a race condition which may cause them to interfere with each other.

One possibility is to document the fact that the library might raise `SIGPIPE` and leave it for the caller to decide how to handle it. This lacks elegance, and may be impracticable if the library already has a defined interface (for example in the case of a plugin). It would be useful to have the option of suppressing `SIGPIPE` internally within the library, but only if this can be done in a safe manner.

Scenario

Suppose you are writing a library for fetching the content of a given URL. For schemes such as `http` and `ftp` this will involve opening a TCP connection to a remote server. If the remote server were to close this connection before the library had finished sending its request then in the normal course of events `SIGPIPE` would be raised.

The default action for `SIGPIPE` is to terminate the process in question. This could be prevented by changing its disposition to `SIG_IGN`, but it would be inadvisable to do that within a library for the reasons discussed above. You are therefore looking for a way to catch `SIGPIPE` for an individual thread while inside the library without affecting the behaviour of that or any other thread while outside the library.

Method

Overview

In its simplest form, the method described here has four steps:

1. Add `SIGPIPE` to the signal mask using `pthread_sigmask`.
2. Execute the library code which might raise `SIGPIPE`.
3. Accept any pending `SIGPIPE` using `sigtimedwait`.
4. Restore the signal mask to its original state.

The following header files are used:

Header	Used by
`<signal.h>`	`SIGPIPE`, `SIG_BLOCK`, `SIG_SETMASK`, `sigset_t`, `struct timespec`, `sigemptyset`, `sigaddset`, `pthread_sigmask`, `sigtimedwait`
`<stdlib.h>`	`exit`
`<stdio.h>`	`perror`

A possible concern regarding this method are that it relies on `SIGPIPE` being delivered to the thread which attempted the write operation. It will be on systems that fully conform to POSIX.1-2008, or to POSIX.1-2001 with corrections applied, but this is a matter where there has been some historical variation in behaviour that would be difficult to reliably test for at compile time.

Add SIGPIPE to the signal mask using pthread_sigmask

Signals can be temporarily blocked by adding them to the signal mask using the function `pthread_sigmask`:

```
sigset_t sigpipe_mask;
sigemptyset(&sigpipe_mask);
sigaddset(&sigpipe_mask, SIGPIPE);
sigset_t saved_mask;
if (pthread_sigmask(SIG_BLOCK, &sigpipe_mask, &saved_mask) == -1) {
  perror("pthread_sigmask");
  exit(1);
}
```

Each thread has its own signal mask, so it is safe to do this within a library provided that you restore the mask afterwards. Blocking only defers delivery of the signal, so it remains pending, but for current purposes that will suffice.

Note that the function `sigprocmask` is not a suitable alternative to `pthread_sigmask` because it is not specified for use in multithreaded programs (and there would be no sense in using this method unless you are attempting to be thread-safe).

Accept any pending SIGPIPE using sigtimedwait

If there is a pending `SIGPIPE` then you do not want it to be delivered, because that would invoke the signal action which you are not safely able to change. An alternative is to accept it using one of the functions from the `sigwait` family. Of these, `sigtimedwait` is the safest choice because it can be called in a manner that will not block:

```
struct timespec zerotime = {0};
if (sigtimedwait(&sigpipe_mask, 0, &zerotime) == -1) {
  perror("sigtimedwait");
  exit(1);
}
```

Restore the signal mask to its original state

The signal mask can now be restored by calling `pthread_sigmask` again:

```
if (pthread_sigmask(SIG_SETMASK, &saved_mask, 0) == -1) {
  perror("pthread_sigmask");
  exit(1);
}
```

Variations

Skipping sigtimedwait if SIGPIPE was not raised

A write operation which raises `SIGPIPE` will also return the error `EPIPE`. If this has not happened then you can skip the call to `sigtimedwait` because there is no pending signal to accept (or if there is, it was not caused by the library code you are attempting to encapsulate).

Checking whether SIGPIPE is already pending

Before blocking `SIGPIPE` you may want to check whether it is already pending. This would imply that it is already blocked, and since there cannot be more than one instance queued, raising it again would have no

effect.

Under these circumstances it would be better to skip the call to `sigtimedwait`, because otherwise the library would absorbed a pending `SIGPIPE` which it did not cause. The two calls to `pthread_sigmask` can also be safely skipped over.

Applicability to other signals

This method is only suitable for handling signals which:

- are directed to a specific thread (as opposed to a process), and
- are capable of being safely blocked.

`SIGPIPE` is not the only signal which meets these criteria, but it is the only one you are likely to want to handle in a typical library. Note in particular that:

- `SIGCHLD` is directed to a process, so will not necessarily be handled by the thread responsible for spawning the child process which terminated. It would not therefore be appropriate for different threads to handle this signal in different ways.
- `SIGFPE`, `SIGILL`, `SIGSEGV` and `SIGBUS` are directed to a particular thread, but the effect of blocking them is undefined.

Further reading

- Signal Concepts, Base Specifications, Issue 7, The Open Group, 2008
- Frank Cusack, ‘signals, threads, SIGPIPE, sigpending (fun!)’ <x5yznl6qrfm.fsf@vger.corp.google.com>, *comp.os.programmer*, 29th May 2003
- Tomash Brechko (Kroki), ‘Suppressing SIGPIPE in a library’, 27th February 2010

Tags: c | posix | signal