Capture the output of a child process in C

Content

Tested on

Debian (Etch, Lenny, Squeeze)

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

Objective

To capture the standard output of a child process in C

Scenario

Suppose that you are writing a program which executes a command as a child process using `fork` and `exec`:

```
pid_t pid = fork();
if (pid == -1) {
  perror("fork");
  exit(1);
} else if (pid == 0) {
  execl(cmdpath, cmdname, (char*)0);
  perror("execl");
  _exit(1);
}
```

The command is expected to write some text to `stdout` and you wish to capture this output for use by the parent process.

Method

Overview

The method described here has four steps:

1. Create a new pipe using the `pipe` function.
2. Connect the entrance of the pipe to `STDOUT_FILENO` within the child process.
3. Close the entrance of the pipe within the parent process.
4. Close the exit from the pipe within the child process.

The parent process will then be able to read the output of the child process from the exit of the pipe.

The following header files are used:

Header	Used by
`<errno.h>`	`errno`, `EINTR`
`<stdio.h>`	`perror`
`<stdlib.h>`	`exit`
`<unistd.h>`	`_exit`, `close`, `dup2`, `execl`, `fork`, `pipe`, `STDOUT_FILENO`
`<sys/wait.h>`	`wait`, `pid_t`

Create a new pipe using the pipe function

A pipe is an anonymous first-in, first-out (FIFO) buffer with endpoints presented as file descriptors. Because these can be owned by different processes, it provides a convenient means for transporting the output of the child process to the parent process:

```
int filedes[2];
if (pipe(filedes) == -1) {
  perror("pipe");
  exit(1);
}
```

The file descriptor for the entrance to the pipe is written to `filedes[1]` and the exit to `filedes[0]`. The former must be transferred to the child process, the latter retained by the parent process. The simplest way to arrange this is to create the pipe before the child process is forked (thus ensuring that each process receives a copy of both descriptors).

Connect the entrance of the pipe to STDOUT_FILENO within the child process

When a process forks, the child inherits a set of file descriptors that are copies of those owned by the parent process. Consequently, if the standard output of the parent process is routed to a particular terminal device then the same will be true of the child process (in the first instance).

To capture the output of the child process, its standard output must instead be routed into the pipe. This can be arranged using the `dup2` command:

```
while ((dup2(filedes[1], STDOUT_FILENO) == -1) && (errno == EINTR)) {}
```

The effect is to close the file descriptor `STDOUT_FILENO` if it was previously open, then (re)open it as a copy of `filedes[1]`. A loop is needed to allow for the possibility of `dup2` being interrupted by a signal. Once this has been done, `filedes[1]` can be closed:

```
close(filedes[1]);
```

It would be equally acceptable to copy the descriptor onto `STDERR_FILENO` in order to capture the standard error stream. To capture both `stdout` and `stderr` you can either create two separate pipes, or if it is acceptable for the streams to be mixed, copy the same file descriptor onto both `STDOUT_FILENO` and `STDERR_FILENO` by calling `dup2` twice.

Close the entrance of the pipe within the parent process

The parent process has no need to access the entrance to the pipe, so `filedes[1]` should be closed within that process too:

```
close(filedes[1]);
```

Close the exit from the pipe within the child process

Similarly, the child process has no need to access the exit from the pipe:

```
close(filedes[0]);
```

(You should also have made arrangements to close any other file descriptors not needed by the child process, regardless of whether you want to capture its output.)

Sample code

The code for managing the pipe can be integrated into the existing program as follows:

```
int filedes[2];
if (pipe(filedes) == -1) {
  perror("pipe");
  exit(1);
}

pid_t pid = fork();
if (pid == -1) {
  perror("fork");
  exit(1);
} else if (pid == 0) {
  while ((dup2(filedes[1], STDOUT_FILENO) == -1) && (errno == EINTR)) {}
  close(filedes[1]);
  close(filedes[0]);
  execl(cmdpath, cmdname, (char*)0);
  perror("execl");
  _exit(1);
}
close(filedes[1]);
```

It is then possible for the parent process to read the output of the child process from file descriptor `filedes[0]`:

```
char buffer[4096];
while (1) {
  ssize_t count = read(filedes[0], buffer, sizeof(buffer));
  if (count == -1) {
    if (errno == EINTR) {
      continue;
    } else {
```

```
      perror("read");
      exit(1);
    }
  } else if (count == 0) {
    break;
  } else {
    handle_child_process_output(buffer, count);
  }
}
close(filedes[0]);
wait(0);
```

If you need to avoid blocking while waiting for output from the child then this can be arranged using `select`, `O_NONBLOCK` or similar.

Alternatives

Using O_CLOEXEC to close file descriptors

If you want to capture its output then it is quite likely that (as in this example) the child process will be calling a function from the `exec` family to transfer control to another program. An alternative method is then available for closing the pipe exit within the child process, by setting the `O_CLOEXEC` flag:

```
if (fcntl(filedes[0], F_SETFD, FD_CLOEXEC) == -1) {
  perror("fcntl");
  exit(1);
}
```

This should be done in the parent process prior to forking. It avoids the need to take any explicit action within the child process to close the file descriptor, provided that `exec` is called. This makes little difference if there is only one file descriptor to close, but when there are many child processes executing in parallel the benefits are more noticable: one system call is needed instead of many, and because the flag can be set immediately when the pipe is created there is less risk of file descriptors being missed.

Using popen

The `popen` function provides most of the functionality described above in the form of a single function call:

```
FILE* fp = popen("pwd", "r");
// ...
int status = pclose(fp);
```

This is undeniably simpler than constructing the pipework explicitly, but `popen` can also be quite limiting:

- It returns a `stdio` stream as opposed to a raw file descriptor, which is unsuitable for handling the output asynchronously.
- Rather than executing the command directly, `popen` typically spawns an instance of the shell first. This can adversely affect performance, and may have other undesirable side effects.
- It is possible to attach to the standard output of the child process or the standard input, but not both at the same time.
- It does not provide access to the process ID of the child process.
- There is no opportunity to modify the context of the child process before `exec` is called.

Workarounds are possible for some of these issues, but in the author's experience it is generally better to accept the minor inconvenience of calling `pipe`, `fork` and `exec` explicitly rather than attempting a `popen`-based solution and taking the risk of it later needing to be rewritten.

See also

- Reap zombie processes using a SIGCHLD handler

Further reading

- pipe, Base Specifications Issue 7, The Open Group, 2008
- dup, Base Specifications Issue 7, The Open Group, 2008

Tags: c | posix | process