

pseudo tty stderr capture problem.

Source: <http://unix.derkeiler.com/Newsgroups/comp.unix.programmer/2003-07/0544.html>

From: Min Shei (*minshei_at_hotmail.com*)

Date: 07/16/03

Date: 15 Jul 2003 16:07:29 -0700

Hi, All,

I was trying to make a simple pseudo tty program to capture the stdout and stderr from the child process to avoid the stdout buffering problem. The following is a little program I found on one new group here (written by Richard Tobin, 1990). However, this program captures the child's stdout very well. But I just couldn't make it work for the stderr of the child process. My platform is Solaris 2.7.

Has anyone here had the same problem? The program could be compiled by:

```
cc -o nobuf nobuf.c pseudopipe.c
```

And if you run `./nobuf ls no-such-file`, you should get message like `"no-such-file: No such file or directory"`. But there is not.

I just couldn't figure out what is wrong with this program. :(Please help.

Thanks in advance.

Min

```
----- nobuf.c
```

```
/* A program to defeat buffering by stdio when the output is a pipe.
 * nobuf program args ...
 * runs the program with its standard input, output, and error
connected
 * to a pseudo-terminal. It transfers data between the
pseudo-terminal
 * and the real standard input etc. Since the program sees its output
is
 * a terminal, it won't buffer it.
 *
 * Richard Tobin, 1990. You may redistribute this program freely if
this
 * whole comment remains intact.
```

comp.unix.programmer: pseudo tty stderr capture problem.

```
*/

#include <stdlib.h>
#include <errno.h>
#include <signal.h>
#include <sys/file.h>
#include <sys/ioctl.h>
#include <signal.h>
#include <unistd.h>

int dead_baby(); /* called when child exits */
int pty, ifds0;

main(argc, argv)
    int argc;
    char *argv[];
{
    int ptyfds[2];

    pseudopipe(ptyfds); /* create pty to talk to child */

    ttymodes(ptyfds[0]);

    signal(SIGCHLD, dead_baby); /* so we can exit when child does */

    subprocess(&argv[1], ptyfds); /* start up the subshell */

    close(ptyfds[0]); /* parent doesn't need it */

    pty = ptyfds[1];

    transfer();
}

/* fork and exec the process in argv */

subprocess(argv, ptyfds)
    char **argv;
    int ptyfds[2];
{
    int tty;

    if(fork() != 0) return;

    /* close files child doesn't want */
    close(ptyfds[1]);
    close(0);
    close(1);
    close(2);

    /* get rid of controlling terminal */
```

pseudo tty stderr capture problem.

comp.unix.programmer: pseudo tty stderr capture problem.

```
tty = open("/dev/tty", O_RDWR, 0);
ioctl(tty, TIOCNOTTY, 0);
close(tty);

/* set up pty as std input, output, error */
dup2(ptyfds[0], 0);
dup2(ptyfds[0], 1);
dup2(ptyfds[0], 2);

close(ptyfds[0]);

/* run the process */
execvp(argv[0], argv);

write(2, "exec failed\n", 12);
_exit(1);
}

/* set the pseudo terminal to fairly sane modes */

ttymodes(pty)
    int pty;
{
    static struct sgttyb sgtty = {B9600, B9600, 255, 255, EVENP|ODDP};
    static struct tchars tchars = {255, 255, 255, 255, 4, 255};

    ioctl(pty, TIOCSETP, &sgtty);
    ioctl(pty, TIOCSETC, &tchars);
}

/* transfer characters between standard input/output and pty */

transfer()
{
    int ifds, nread;
    char buf[256], ctld=4;
    extern int errno;

    ifds0 = (1 << 0) | (1 << pty); /* listen to real and pseudo ttys */

    while(ifds0 != 0)
    {
        ifds = ifds0;

        if(select(pty+1, &ifds, (int *)0, (int *)0, (struct timeval *)0)
< 0)
        {
            if(errno == EINTR)
                continue;
            else
                {
```

comp.unix.programmer: pseudo tty stderr capture problem.

```
        perror("select");
        exit(1);
    }
}

if(ifds & (1<<0))
{
    nread = read(0, buf, 256); /* from tty to process */
    if(nread > 0)
        write(pty, buf, nread);
    else if(nread == 0)
        write(pty, &ctld, 1); /* writing zero bytes doesn't work
*/
    else
        ifds0 &= ~(1 << 0);
}

if(ifds & (1 << pty))
{
    nread = read(pty, buf, 256); /* from process to tty */
    if(nread >= 0)
        write(1, buf, nread);
    else
        ifds0 &= ~(1 << pty);
}
}

while(1)
    sigpause(0); /* wait for child to exit */
}

/* child process exited – do any remaining i/o and exit */

dead_baby()
{
    char buf[256];
    int nread;

    while(ioctl(pty, FIONREAD, &nread) == 0 && nread > 0)
    {
        nread = read(pty, buf, 256);
        write(1, buf, nread);
    }

    exit(0);
}

----- pseudopipe.c
-----
```

comp.unix.programmer: pseudo tty stderr capture problem.

```
/* like pipe(2) but uses pseudo-terminals. consequently, the
connection
* is bi-directional, and will appear as a terminal to any forked
process.
* copyright (C) 1986 Richard M Tobin
* you may freely distribute/modify this provided this whole comment
remains
* intact.
*/
```

```
#include <sys/types.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <strings.h>
#include <stdio.h>
```

```
/* pseudopipe puts two file descriptors in its argument array. fd[0]
is the
* slave side of a pseudo-terminal, fd[1] is the master side. to talk
to a
* child process, dup fd[0] onto file descriptors 0, 1 and 2.
* if anything goes wrong (probably no free pty) -1 is returned.
*/
```

```
int pseudopipe(fd)
int fd[2];
{
    char pty_master[128], pty_slave[128];
#define masterfd fd[1]
#define slavefd fd[0]
    FILE *cons;
    int start=0;

    masterfd = findpty(pty_master, pty_slave, &start);
    if(masterfd < 0) return -1;
    slavefd = open(pty_slave, O_RDWR, 0);

    if(slavefd >= 0) return 0;

    return -1;

#undef masterfd
#undef slavefd
}
```

```
/* findpty find a free pseudo-terminal, and opens the master side of
it.
* the arguments are filled in to be the paths of the master and
slave.
*/
```

pseudo tty stderr capture problem.

comp.unix.programmer: pseudo tty stderr capture problem.

```
#define MASTER "/dev/ptyXY"
#define SLAVE "/dev/ttyXY"

findpty(master_path, slave_path, ptyno)
char *master_path, *slave_path;
int *ptyno;
{
    struct stat statbuf;
    char c1;
    int i, master;

    strcpy(master_path, MASTER);
    strcpy(slave_path, SLAVE );

    /* the pseudo-ttys are /dev/ptyXY, where X is p,q,or r and Y is a
     * hex digit. There are usually 16, 32 or 48 so we check for the
     * of /dev/ptyX0 before trying to open them all.
     */

    for(; *ptyno <= ('s'-'p') * 16; ++*ptyno)
    {
        master_path[strlen(MASTER)-2] = *ptyno / 16 + 'p';
        master_path[strlen(MASTER)-1] = "0123456789abcdef"[*ptyno & 15];
        master = open(master_path,O_RDWR);
        if(master < 0) continue;
        slave_path[strlen(SLAVE)-2] = master_path[strlen(MASTER)-2];
        slave_path[strlen(SLAVE)-1] = master_path[strlen(MASTER)-1];
        return master;
    }

    return -1;
}
```