Lecture #15 – Pseudo Terminals (Chapter 19)

- Overview

Pseudo-terminal is an imaginary terminal to be used by applications that need to talk to a terminal.

Note: See Figure 19.1 on page 676 for typical pseudo-terminal arrangement

1. Normally, a process opens the pseudo-terminal master, and calls fork.

2. The child establishes a new session, opens the corresponding pseudo-terminal slave, duplicates it to be stdin, stdout, and stderr, and then calls exec.

3. The pseudo-terminal slave becomes the controlling terminal for the child process.

4. It now appears to the user process above the slave that its stdin, stdout, and stderr are connected to a terminal device.

5. Anything written to the master appears as input to the slave, and vice versa.

In other words, pseudo-terminals allow us to drive applications that want to be connected to a terminal, without having an actual terminal involved.

Typical uses of pseudo-terminals:

1. network login servers

   See Figure 19.3 on page 678.

   Basically, rlogind is driving the shell via a pseudo terminal.

   This allows you to have a command login that emulates a serial terminal over the network.

2. script program

   See Figure 19.4 on page 679

3. expect program

   Pseudo-terminals can be used to drive interactive programs in non-interactive modes. Expect provides a general way to write shell scripts to drive interactive programs.

4. running co-processes

   In chapter 14, we noticed the running a co-process that used stdio could lead to a deadlock situation since I/O across a pipe is fully buffered. If we do not have access to the source code for the co-process, then we cannot fix this problem.

   Another solution is to place a pseudo-terminal between our process and the co-process, and the stdio library uses line buffering instead of full buffering since the pseudo-terminal looks like a terminal.

5. watching output of long running programs

   See Figure 19.6 on page 681

- Opening Pseudo-Terminal Devices

Opening pseudo-terminal devices differs between SVR4 and 4.3+ BSD. We will provide "ptym_open" and "ptys_open" to handle the differences.

```
int     ptym_open(char *pts_name);
int     ptys_open(int fdm, char *pts_name);
```

"ptym_open" determines the next available pty master and opens the device.
The slave is returned through pts_name, and the file descriptor as the return value.

"ptys_open" uses the filedes and name from "ptym_open" to open the slave.

SVR4:

```
int ptym_open(char *pts_name)
{
        char    *ptr;
        int     fdm;

        strcpy(pts_name, "/dev/ptmx");

        if ((fdm = open(pts_name, O_RDWR)) < 0)
                return(-1);

        if (grantpt(fdm) < 0)                   /*  grant access to slave */
        {
                close(fdm);
                return(-2);
        }
```

```
        if (unlockpt(fdm) < 0)                  /*  clear slave's lock flag */
        {
                close(fdm);
                return(-3);
        }

        if ((ptr = ptsname(fdm)) == NULL)       /*  slave's name */
        {
                close(fdm);
                return(-4);
        }

        strcpy(pts_name, ptr);                  /*  name of slave */
        return(fdm);                            /*  fd of master */
}

int ptys_open(int fdm, char *pts_name)
{
        int     fds;

        if ((fds = open(pts_name, O_RDWR)) < 0)
        {
                close(fdm);
                return(-5);
        }

        if (ioctl(fds, I_PUSH, "ptem") < 0)
        {
                close(fdm);
                close(fds);
                return(-6);
        }

        if (ioctl(fds, I_PUSH, "ldterm") < 0)
        {
                close(fdm);
                close(fds);
                return(-7);
        }

        if (ioctl(fds, I_PUSH, "ttcompat") < 0)
        {
                close(fdm);     close(fds);     return(-8);
        }
        return(fds);
}
```

4.3+ BSD:

```
int ptym_open(char *pts_name)
{
        int     fdm;
        char    *ptr1, *ptr2;

        strcpy(pts_name, "/dev/ptyXY");

        for (ptr1 = "pqrstuvwxyzPQRST"; *ptr1 != 0; ptr1++)
        {
                pts_name[8] = *ptr1;

                for (ptr2 = "0123456789abcdef"; *ptr2 != 0; ptr2++)
                {
                        pts_name[9] = *ptr2;

                        /*  attempt the open of the master */

                        if ((fdm = open(pts_name, O_RDWR)) < 0)
                        {
                                if (errno == ENOENT)
                                        return(-1);
                                else
                                        continue;
                        }

                        pts_name[5] = 't';      /* change from "pty" to "tty" */
                        return(fdm);
                }
        }

        return(-1);
}

int ptys_open(int fdm, char *pts_name)
{
        struct group    *grptr;
        int             gid, fds;

        if ((grptr = getgrnam("tty")) != NULL)
                gid = grptr->gr_gid;
        else
                gid = -1;

        chown (pts_name, getuid(), gid);
```

```
        chmod(pts_name, S_IRUSR | S_IWUSR | S_IWGRP);

        if ((fds = open(pts_name, O_RDWR)) < 0)
        {
                close(fdm);
                return(-1);
        }

        return(fds);
}
```

- pty_fork

  pid_t pty_fork(int *fdm, char *slave, const struct termios *slave_termios,
  const struct winsize *slave_winsize);

  Return values are same as "fork"

  Code structure is as follows:

```
call ptym_open

fork
        P:
                Save filedes of master
                Return pid of child
        C:
                Call setsid
                Call ptys_open
                Call dup2 to set stdin, stdout, and stderr to slave
```

  Code:

```
#include <sys/types.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <termios.h>
#ifndef TIOCGWINSZ
#include <sys/ioctl.h>  /* 44BSD requires this too */
#endif

pid_t pty_fork(int *ptrfdm, char *slave_name,
        const struct termios *slave_termios,
        const struct winsize *slave_winsize)
{
```

```
        int     fdm, fds;
        pid_t   pid;
        char    pts_name[20];

        if ( (fdm = ptym_open(pts_name)) < 0)
        {
                fprintf(stderr, "can't open master pty: %s", pts_name);
                exit(1);
        }

        if (slave_name != NULL)
                strcpy(slave_name, pts_name);   /* return name of slave */

        if ( (pid = fork()) < 0)
                return(-1);

        else if (pid == 0) {            /* child */
                if (setsid() < 0)
                {
                        fprintf(stderr, "setsid error");
                        exit(1);
                }

                /* SVR4 acquires controlling terminal on open() */
                if ( (fds = ptys_open(fdm, pts_name)) < 0)
                {
                        fprintf(stderr, "can't open slave pty");
                        exit(1);
                }

                close(fdm);             /* all done with master in child */

#if     defined(TIOCSCTTY) && !defined(CIBAUD)
                                /* 44BSD way to acquire controlling terminal */
                                /* !CIBAUD to avoid doing this under SunOS */
                if (ioctl(fds, TIOCSCTTY, (char *) 0) < 0)
                {
                        fprintf(stderr, "TIOCSCTTY error");
                        exit(1);
                }
#endif
                                /* set slave's termios and window size */
                if (slave_termios != NULL)
                {
                        if (tcsetattr(fds, TCSANOW, slave_termios) < 0)
                        {
```

```
                fprintf(stderr, "tcsetattr error on slave pty");
                exit(1);
            }
        }

        if (slave_winsize != NULL)
        {
            if (ioctl(fds, TIOCSWINSZ, slave_winsize) < 0)
            {
                fprintf(stderr, "TIOCSWINSZ error on slave pty");
                exit(1);
            }
        }

        /* slave becomes stdin/stdout/stderr of child */
        if (dup2(fds, 0) != 0)
        {
            fprintf(stderr, "dup2 error to stdin");
            exit(1);
        }

        if (dup2(fds, 1) != 1)
        {
            fprintf(stderr, "dup2 error to stdout");
            exit(1);
        }

        if (dup2(fds, 2) != 2)
        {
            fprintf(stderr, "dup2 error to stderr");
            exit(1);
        }

        if (fds > 2)
            close(fds);

        return(0);          /* child returns 0 just like fork() */

    }
    else
    {               /* parent */
        *ptrfdm = fdm;  /* return fd of master */
        return(pid);    /* parent returns pid of child */
    }
}
```

- The pty program

The goal of this program is to be able to type:

Pty prog arg1 arg2

instead of:

Prog arg1 arg2

When we use pty to execute another program, that program is executed in a session by itself, and connected to a pseudo terminal.

Note: See uses of this program on pages 698-705

Code (pty.c):

```
int main(int argc, char *argv[])
{
        int             fdm, c, ignoreeof, interactive, noecho, verbose;
        pid_t           pid;
        char            *driver, slave_name[20];
        struct termios  orig_termios;
        struct winsize  size;

        interactive = isatty(0);
        ignoreeof = 0;
        noecho = 0;
        verbose = 0;
        driver = NULL;

        opterr = 0;             /* don't want getopt() writing to stderr */
        while ( (c = getopt(argc, argv, "d:einv")) != EOF)
        {
                switch (c) {
                case 'd':               /* driver for stdin/stdout */
                        driver = optarg;
                        break;

                case 'e':               /* noecho for slave pty's line discipline */
                        noecho = 1;
                        break;

                case 'i':               /* ignore EOF on standard input */
                        ignoreeof = 1;
                        break;
```

```
        case 'n':               /* not interactive */
                interactive = 0;
                break;

        case 'v':               /* verbose */
                verbose = 1;
                break;

        case '?':
                fprintf(stderr, "unrecognized option: -%c", optopt);
                exit(1);
        }
}

if (optind >= argc)
{
        fprintf(stderr, "usage: pty [ -d driver -einv ] program [ arg ... ]");
        exit(1);
}

if (interactive)
{       /* fetch current termios and window size */
        if (tcgetattr(0, &orig_termios) < 0)
        {
                fprintf(stderr, "tcgetattr error on stdin");
                exit(1);
        }

        if (ioctl(0, TIOCGWINSZ, (char *) &size) < 0)
        {
                fprintf(stderr, "TIOCGWINSZ error");
                exit(1);
        }

        pid = pty_fork(&fdm, slave_name, &orig_termios, &size);

}
else
        pid = pty_fork(&fdm, slave_name, NULL, NULL);

if (pid < 0)
{
        fprintf(stderr, "fork error");
        exit(1);
}
```

```
        else if (pid == 0)
        {               /* child */
                if (noecho)
                        set_noecho(0);  /* stdin is slave pty */

                if (execvp(argv[optind], &argv[optind]) < 0)
                {
                        fprintf(stderr, "can't execute: %s", argv[optind]);
                        exit(1);
                }
        }

        if (verbose)
        {
                fprintf(stderr, "slave name = %s\n", slave_name);

                if (driver != NULL)
                        fprintf(stderr, "driver = %s\n", driver);
        }

        if (interactive && driver == NULL)
        {
                if (tty_raw(0) < 0)     /* user's tty to raw mode */
                {
                        fprintf(stderr, "tty_raw error");
                        exit(1);
                }

                if (atexit(tty_atexit) < 0)             /* reset user's tty on exit */
                {
                        fprintf(stderr, "atexit error");
                        exit(1);
                }
        }

        if (driver)
                do_driver(driver);      /* changes our stdin/stdout */

        loop(fdm, ignoreeof);   /* copies stdin -> ptym, ptym -> stdout */

        exit(0);
}

static void set_noecho(int fd)          /* turn off echo (for slave pty) */
{
        struct termios  stermios;
```

```
        if (tcgetattr(fd, &stermios) < 0)
        {
            fprintf(stderr, "tcgetattr error");
            exit(1);
        }

        stermios.c_lflag &= ~(ECHO | ECHOE | ECHOK | ECHONL);
        stermios.c_oflag &= ~(ONLCR);

        /* also turn off NL to CR/NL mapping on output */

        if (tcsetattr(fd, TCSANOW, &stermios) < 0)
        {
            fprintf(stderr, "tcsetattr error");
            exit(1);
        }
    }
```

Code (loop.c):

```
    static void     sig_term(int);
    static volatile sig_atomic_t    sigcaught;      /* set by signal handler */

    void loop(int ptym, int ignoreeof)
    {
        pid_t   child;
        int     nread;
        char    buff[BUFFSIZE];

        if ( (child = fork()) < 0)
        {
            fprintf(stderr, "fork error");
            exit(1);
        }
        else if (child == 0)
        {       /* child copies stdin to ptym */
            for ( ; ; )
            {
                if ( (nread = read(0, buff, BUFFSIZE)) < 0)
                {
                    fprintf(stderr, "read error from stdin");
                    exit(1);
                }
                else if (nread == 0)
                    break;          /* EOF on stdin means we're done */
```

```
                if (writen(ptym, buff, nread) != nread)
                {
                        fprintf(stderr, "writen error to master pty");
                        exit(1);
                }
        }

        /* We always terminate when we encounter an EOF on stdin,
           but we only notify the parent if ignoreeof is 0. */

        if (ignoreeof == 0)
                kill(getppid(), SIGTERM);       /* notify parent */

        exit(0);        /* and terminate; child can't return */
}

/* parent copies ptym to stdout */

if (signal(SIGTERM, sig_term) == SIG_ERR)
{
        fprintf(stderr, "signal_intr error for SIGTERM");
        exit(1);
}

for ( ; ; )
{
        if ( (nread = read(ptym, buff, BUFFSIZE)) <= 0)
                break;          /* signal caught, error, or EOF */

        if (writen(1, buff, nread) != nread)
        {
                fprintf(stderr, "writen error to stdout");
                exit(1);
        }
}

/* There are three ways to get here: sig_term() below caught the
 * SIGTERM from the child, we read an EOF on the pty master (which
 * means we have to signal the child to stop), or an error. */

if (sigcaught == 0)     /* tell child if it didn't send us the signal */
        kill(child, SIGTERM);

return;         /* parent returns to caller */
}
```

```
/* The child sends us a SIGTERM when it receives an EOF on
 * the pty slave or encounters a read() error. */

static void sig_term(int signo)
{
        sigcaught = 1;          /* just set flag and return */
        return;                 /* probably interrupts read() of ptym */
}
```

Code (driver.c):

```
void do_driver(char *driver)
{
        pid_t   child;
        int             pipe[2];

        /* create a stream pipe to communicate with the driver */

        if (s_pipe(pipe) < 0)
        {
                fprintf(stderr, "can't create stream pipe");
                exit(1);
        }

        if ( (child = fork()) < 0)
        {
                fprintf(stderr, "fork error");
                exit(1);
        }

        else if (child == 0)
        {       /* child */

                close(pipe[1]);

                /* stdin for driver */

                if (dup2(pipe[0], 0) != 0)
                {
                        fprintf(stderr, "dup2 error to stdin");
                        exit(1);
                }

                /* stdout for driver */
                if (dup2(pipe[0], 1) != 1)
```

```
                {
                        fprintf(stderr, "dup2 error to stdout");
                        exit(1);
                }

                close(pipe[0]);

                /* leave stderr for driver alone */

                execlp(driver, driver, (char *) 0);
                fprintf(stderr, "execlp error for: %s", driver);
                exit(1);
        }

        close(pipe[0]);         /* parent */

        if (dup2(pipe[1], 0) != 1)
        {
                fprintf(stderr, "dup2 error to stdin");
                exit(1);
        }

        if (dup2(pipe[1], 1) != 1)
        {
                fprintf(stderr, "dup2 error to stdout");
                exit(1);
        }

        close(pipe[1]);

        /* Parent returns, but with stdin and stdout connected
           to the driver. */
}
```