*NIX

March 10, 2009, 8:04 a.m.
posted by tyler

Python SQL Java php Perl game development web development *nix internet graphics hardware telecommunications C++ Flash Active Directory Windows

Passing File Descriptors

Passing File Descriptors

The ability to pass an open file descriptor between processes is powerful. It can lead to different ways of designing client-server applications. It allows one process (typically a server) to do everything that is required to open a file (involving such details as translating a network name to a network address, dialing a modem, negotiating locks for the file, etc.) and simply pass back to the calling process a descriptor that can be used with all the I/O functions. All the details involved in opening the file or device are hidden from the client.

We must be more specific about what we mean by "passing an open file descriptor" from one process to another. Recall Figure, which showed two processes that have opened the same file. Although they share the same v-node, each process has its own file table entry.

When we pass an open file descriptor from one process to another, we want the passing process and the receiving process to share the same file table entry. Figure shows the desired arrangement.

18. Passing an open file from the top process to the bottom process

[View full size image]

Technically, we are passing a pointer to an open file table entry from one process to another. This pointer is assigned the first available descriptor in the receiving process. (Saying that we are passing an open descriptor mistakenly gives the impression that the descriptor number in the receiving process is the same as in the sending process, which usually isn't true.) Having two processes share an open file table is exactly what happens after a `fork` (recall Figure).

What normally happens when a descriptor is passed from one process to another is that the sending process, after passing the descriptor, then closes the descriptor. Closing the descriptor by the sender doesn't really close the file or device, since the descriptor is still considered open by the receiving process (even if the receiver hasn't specifically received the descriptor yet).

We define the following three functions that we use in this chapter to send and receive file descriptors. Later in this section, we'll show the code for these three functions for both STREAMS and sockets.

```
#include "apue.h"

int send_fd(int fd, int fd_to_send);
int send_err(int fd, int status, const char *errmsg);
```

Both return: 0 if OK, 1 on error

[View full width]

```
int recv_fd(int fd, ssize_t (*userfunc)(int, const
 void *, size_t));
```

Returns: file descriptor if OK, negative value on error

A process (normally a server) that wants to pass a descriptor to another process calls either `send_fd` or `send_err`. The process waiting to receive the descriptor (the client) calls `recv_fd`.

The `send_fd` function sends the descriptor fd_to_send across using the STREAMS pipe or UNIX domain socket represented by fd.

> We'll use the term s-pipe to refer to a bidirectional communication channel that could be implemented as either a STREAMS pipe or a UNIX domain stream socket.

The `send_err` function sends the errmsg using fd, followed by the status byte. The value of status must be in the range 1 through 255.

Clients call `recv_fd` to receive a descriptor. If all is OK (the sender called `send_fd`), the non-negative descriptor is returned as the value of the function. Otherwise, the value returned is the status that was sent by `send_err` (a negative value in the range 1 through -255). Additionally, if an error message was sent by the server, the client's userfunc is called to process the message. The first argument to userfunc is the constant `STDERR_FILENO`, followed by a pointer to the error message and its length. The return value from userfunc is the number of bytes written or a negative number on error. Often, the client specifies the normal `write` function as the userfunc.

We implement our own protocol that is used by these three functions. To send a descriptor, `send_fd` sends two bytes of 0, followed by the actual descriptor. To send an error, `send_err` sends the errmsg, followed by a byte of 0, followed by the absolute value of the status byte (1 through 255). The `recv_fd` function reads everything on the s-pipe until it encounters a null byte. Any characters read up to this point are passed to the caller's userfunc. The next byte read by `recv_fd` is the status byte. If the status byte is 0, a descriptor was passed; otherwise, there is no descriptor to receive.

The function `send_err` calls the `send_fd` function after writing the error message to the s-pipe. This is shown in Figure.

Figure. The `send_err` function

```
#include "apue.h"
/*
 * Used when we had planned to send an fd using send_fd(),
 * but encountered an error instead. We send the error back
 * using the send_fd()/recv_fd() protocol.
 */
int
send_err(int fd, int errcode, const char *msg)
{
    int     n;

    if ((n = strlen(msg)) > 0)
        if (writen(fd, msg, n) != n)    /* send the error message */
            return(-1);

    if (errcode >= 0)
        errcode = -1;   /* must be negative */

    if (send_fd(fd, errcode) < 0)
        return(-1);
```

```
    return(0);
}
```

In the next two sections, we'll look at the implementation of the `send_fd` and `recv_fd` functions.

Passing File Descriptors over STREAMS-Based Pipes

With STREAMS pipes, file descriptors are exchanged using two `ioctl` commands: `I_SENDFD` and `I_RECVFD`. To send a descriptor, we set the third argument for `ioctl` to the actual descriptor. This is shown in Figure.

Figure. The `send_fd` function for STREAMS pipes

```
#include "apue.h"
#include <stropts.h>

/*
 * Pass a file descriptor to another process.
 * If fd<0, then -fd is sent back instead as the error status.
 */
int
send_fd(int fd, int fd_to_send)
{
    char    buf[2];     /* send_fd()/recv_fd() 2-byte protocol */

    buf[0] = 0;         /* null byte flag to recv_fd() */
    if (fd_to_send < 0) {
        buf[1] = -fd_to_send;   /* nonzero status means error */
        if (buf[1] == 0)
            buf[1] = 1; /* -256, etc. would screw up protocol */
    } else {
        buf[1] = 0;     /* zero status means OK */
    }

    if (write(fd, buf, 2) != 2)
        return(-1);
    if (fd_to_send >= 0)
```

```
        if (ioctl(fd, I_SENDFD, fd_to_send) < 0)
            return(-1);
    return(0);
}
```

When we receive a descriptor, the third argument for `ioctl` is a pointer to a `strrecvfd` structure:

```
struct strrecvfd {
    int    fd;       /* new descriptor */
    uid_t  uid;      /* effective user ID of sender */
    gid_t  gid;      /* effective group ID of sender */
    char   fill[8];
};
```

The `recv_fd` function reads the STREAMS pipe until the first byte of the 2-byte protocol (the null byte) is received. When we issue the `I_RECVFD ioctl` command, the next message on the stream head's read queue must be a descriptor from an `I_SENDFD` call, or we get an error. This function is shown in Figure.

Figure. The `recv_fd` function for STREAMS pipes

```
#include "apue.h"
#include <stropts.h>

/*
 * Receive a file descriptor from another process (a server).
 * In addition, any data received from the server is passed
 * to (*userfunc)(STDERR_FILENO, buf, nbytes). We have a
 * 2-byte protocol for receiving the fd from send_fd().
 */
int
recv_fd(int fd, ssize_t (*userfunc)(int, const void *, size_t))
{
    int                 newfd, nread, flag, status;
    char                *ptr;
    char                buf[MAXLINE];
```

```
        struct strbuf       dat;
        struct strrecvfd    recvfd;

        status = -1;
        for ( ; ; ) {
            dat.buf = buf;
            dat.maxlen = MAXLINE;
            flag = 0;
            if (getmsg(fd, NULL, &dat, &flag) < 0)
                err_sys("getmsg error");
            nread = dat.len;
            if (nread == 0) {
                err_ret("connection closed by server");
                return(-1);
            }
            /*
             * See if this is the final data with null & status.
             * Null must be next to last byte of buffer, status
             * byte is last byte. Zero status means there must
             * be a file descriptor to receive.
             */
            for (ptr = buf; ptr < &buf[nread]; ) {
                if (*ptr++ == 0) {
                    if (ptr != &buf[nread-1])
                        err_dump("message format error");
                     status = *ptr & 0xFF;   /* prevent sign extension */
                     if (status == 0) {
                         if (ioctl(fd, I_RECVFD, &recvfd) < 0)
                             return(-1);
                         newfd = recvfd.fd;  /* new descriptor */
                     } else {
                         newfd = -status;
                     }
                     nread -= 2;
                }
            }
            if (nread > 0)
                if ((*userfunc)(STDERR_FILENO, buf, nread) != nread)
                     return(-1);
```

```
        if (status >= 0)    /* final data has arrived */
            return(newfd);  /* descriptor, or -status */
    }
}
```

Passing File Descriptors over UNIX Domain Sockets

To exchange file descriptors using UNIX domain sockets, we call the `sendmsg`(2) and `recvmsg`(2) functions (Section 16.5). Both functions take a pointer to a `msghdr` structure that contains all the information on what to send or receive. The structure on your system might look similar to the following:

```
struct msghdr {
    void         *msg_name;        /* optional address */
    socklen_t     msg_namelen;     /* address size in bytes */
    struct iovec *msg_iov;         /* array of I/O buffers */
    int           msg_iovlen;      /* number of elements in array */
    void         *msg_control;     /* ancillary data */
    socklen_t     msg_controllen;  /* number of ancillary bytes */
    int           msg_flags;       /* flags for received message */
};
```

The first two elements are normally used for sending datagrams on a network connection, where the destination address can be specified with each datagram. The next two elements allow us to specify an array of buffers (scatter read or gather write), as we described for the `readv` and `writev` functions (Section 14.7). The `msg_flags` field contains flags describing the message received, as summarized in Figure.

Two elements deal with the passing or receiving of control information. The `msg_control` field points to a `cmsghdr` (control message header) structure, and the `msg_controllen` field contains the number of bytes of control information.

```
struct cmsghdr  {
    socklen_t   cmsg_len;    /* data byte count, including header */
    int         cmsg_level;  /* originating protocol */
    int         cmsg_type;   /* protocol-specific type */
```

```
        /* followed by the actual control message data */
    };
```

To send a file descriptor, we set `cmsg_len` to the size of the `cmsghdr` structure, plus the size of an integer (the descriptor). The `cmsg_level` field is set to `SOL_SOCKET`, and `cmsg_type` is set to `SCM_RIGHTS`, to indicate that we are passing access rights. (`SCM` stands for socket-level control message.) Access rights can be passed only across a UNIX domain socket. The descriptor is stored right after the `cmsg_type` field, using the macro `CMSG_DATA` to obtain the pointer to this integer.

Three macros are used to access the control data, and one macro is used to help calculate the value to be used for `cmsg_len`.

```
#include <sys/socket.h>

unsigned char *CMSG_DATA(struct cmsghdr *cp);
```

Returns: pointer to data associated with `cmsghdr` structure

```
struct cmsghdr *CMSG_FIRSTHDR(struct msghdr *mp);
```

Returns: pointer to first `cmsghdr` structure associated with the `msghdr` structure, or `NULL` if none exists

```
struct cmsghdr *CMSG_NXTHDR(struct msghdr *mp,
                            struct cmsghdr *cp);
```

Returns: pointer to next `cmsghdr` structure associated with the msghdr structure given the current cmsghdr structure, or NULL if we're at the last one

```
unsigned int CMSG_LEN(unsigned int nbytes);
```

Returns: size to allocate for data object nbytes large

The Single UNIX Specification defines the first three macros, but omits CMSG_LEN.

The CMSG_LEN macro returns the number of bytes needed to store a data object of size nbytes, after adding the size of the cmsghdr structure, adjusting for any alignment constraints required by the processor architecture, and rounding up.

The program in Figure is the send_fd function for UNIX domain sockets.

Figure. The send_fd function for UNIX domain sockets

```
#include "apue.h"
#include <sys/socket.h>

/* size of control buffer to send/recv one file descriptor */
#define CONTROLLEN  CMSG_LEN(sizeof(int))

static struct cmsghdr   *cmptr = NULL;  /* malloc'ed first time */

/*
 * Pass a file descriptor to another process.
 * If fd<0, then -fd is sent back instead as the error status.
 */
int
```

```
send_fd(int fd, int fd_to_send)
{
    struct iovec    iov[1];
    struct msghdr   msg;
    char            buf[2]; /* send_fd()/recv_fd() 2-byte protocol */

    iov[0].iov_base = buf;
    iov[0].iov_len  = 2;
    msg.msg_iov     = iov;
    msg.msg_iovlen  = 1;
    msg.msg_name    = NULL;
    msg.msg_namelen = 0;
    if (fd_to_send < 0) {
        msg.msg_control    = NULL;
        msg.msg_controllen = 0;
        buf[1] = -fd_to_send;   /* nonzero status means error */
        if (buf[1] == 0)
            buf[1] = 1; /* -256, etc. would screw up protocol */
    } else {
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        cmptr->cmsg_level  = SOL_SOCKET;
        cmptr->cmsg_type   = SCM_RIGHTS;
        cmptr->cmsg_len    = CONTROLLEN;
        msg.msg_control    = cmptr;
        msg.msg_controllen = CONTROLLEN;
        *(int *)CMSG_DATA(cmptr) = fd_to_send;     /* the fd to pass */
        buf[1] = 0;          /* zero status means OK */
    }
    buf[0] = 0;              /* null byte flag to recv_fd() */
    if (sendmsg(fd, &msg, 0) != 2)
        return(-1);
    return(0);
}
```

In the `sendmsg` call, we send both the protocol data (the null and the status byte) and the descriptor.

To receive a descriptor (Figure), we allocate enough room for a `cmsghdr` structure and a descriptor, set `msg_control` to point to the allocated area, and call `recvmsg`. We use the `CMSG_LEN` macro to calculate the amount of space needed.

We read from the socket until we read the null byte that precedes the final status byte. Everything up to this null byte is an error message from the sender. This is shown in Figure.

Figure. The `recv_fd` function for UNIX domain sockets

```
#include "apue.h"
#include <sys/socket.h>     /* struct msghdr */

/* size of control buffer to send/recv one file descriptor */
#define CONTROLLEN  CMSG_LEN(sizeof(int))

static struct cmsghdr   *cmptr = NULL;      /* malloc'ed first time */

/*
 * Receive a file descriptor from a server process.  Also, any data
 * received is passed to (*userfunc)(STDERR_FILENO, buf, nbytes).
 * We have a 2-byte protocol for receiving the fd from send_fd().
 */
int
recv_fd(int fd, ssize_t (*userfunc)(int, const void *, size_t))
{
   int              newfd, nr, status;
   char             *ptr;
   char             buf[MAXLINE];
   struct iovec     iov[1];
   struct msghdr    msg;

   status = -1;
   for ( ; ; ) {
       iov[0].iov_base = buf;
       iov[0].iov_len  = sizeof(buf);
       msg.msg_iov     = iov;
       msg.msg_iovlen  = 1;
       msg.msg_name    = NULL;
```

```
        msg.msg_namelen = 0;
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        msg.msg_control    = cmptr;
        msg.msg_controllen = CONTROLLEN;
        if ((nr = recvmsg(fd, &msg, 0)) < 0) {
            err_sys("recvmsg error");
        } else if (nr == 0) {
            err_ret("connection closed by server");
            return(-1);
        }
        /*
         * See if this is the final data with null & status.  Null
         * is next to last byte of buffer; status byte is last byte.
         * Zero status means there is a file descriptor to receive.
         */
        for (ptr = buf; ptr < &buf[nr]; ) {
            if (*ptr++ == 0) {
                if (ptr != &buf[nr-1])
                    err_dump("message format error");
                status = *ptr & 0xFF;  /* prevent sign extension */
                if (status == 0) {
                    if (msg.msg_controllen != CONTROLLEN)
                        err_dump("status = 0 but no fd");
                    newfd = *(int *)CMSG_DATA(cmptr);
                } else {
                    newfd = -status;
                }
                nr -= 2;
            }
         }
         if (nr > 0 && (*userfunc)(STDERR_FILENO, buf, nr) != nr)
             return(-1);
         if (status >= 0)    /* final data has arrived */
             return(newfd);  /* descriptor, or -status */
    }
}
```

Note that we are always prepared to receive a descriptor (we set `msg_control` and `msg_controllen` before each call to `recvmsg`), but only if `msg_controllen` is nonzero on return did we receive a descriptor.

When it comes to passing file descriptors, one difference between UNIX domain sockets and STREAMS pipes is that we get the identity of the sending process with STREAMS pipes. Some versions of UNIX domain sockets provide similar functionality, but their interfaces differ.

> FreeBSD 5.2.1 and Linux 2.4.22 provide support for sending credentials over UNIX domain sockets, but they do it differently. Mac OS X 10.3 is derived in part from FreeBSD, but has credential passing disabled. Solaris 9 doesn't support sending credentials over UNIX domain sockets.

With FreeBSD, credentials are transmitted as a `cmsgcred` structure:

```
#define CMGROUP_MAX 16
struct cmsgcred {
    pid_t cmcred_pid;                  /* sender's process ID */
    uid_t cmcred_uid;                  /* sender's real UID */
    uid_t cmcred_euid;                 /* sender's effective UID */
    gid_t cmcred_gid;                  /* sender's real GID */
    short cmcred_ngroups;              /* number of groups */
    gid_t cmcred_groups[CMGROUP_MAX];  /* groups */
};
```

When we transmit credentials, we need to reserve space only for the `cmsgcred` structure. The kernel will fill it in for us to prevent an application from pretending to have a different identity.

On Linux, credentials are transmitted as a `ucred` structure:

```
struct ucred {
    uint32_t pid;   /* sender's process ID */
    uint32_t uid;   /* sender's user ID */
```

```
        uint32_t gid;   /* sender's group ID */
    };
```

Unlike FreeBSD, Linux requires that we initialize this structure before transmission. The kernel will ensure that applications either use values that correspond to the caller or have the appropriate privilege to use other values.

Figure shows the `send_fd` function updated to include the credentials of the sending process.

24. Sending credentials over UNIX domain sockets

```
#include "apue.h"
#include <sys/socket.h>

#if defined(SCM_CREDS)          /* BSD interface */
#define CREDSTRUCT      cmsgcred
#define SCM_CREDTYPE    SCM_CREDS
#elif defined(SCM_CREDENTIALS)  /* Linux interface */
#define CREDSTRUCT      ucred
#define SCM_CREDTYPE    SCM_CREDENTIALS
#else
#error passing credentials is unsupported!
#endif

/* size of control buffer to send/recv one file descriptor */
#define RIGHTSLEN   CMSG_LEN(sizeof(int))
#define CREDSLEN    CMSG_LEN(sizeof(struct CREDSTRUCT))
#define CONTROLLEN  (RIGHTSLEN + CREDSLEN)

static struct cmsghdr   *cmptr = NULL;  /* malloc'ed first time */

/*
 * Pass a file descriptor to another process.
 * If fd<0, then -fd is sent back instead as the error status.
 */
int
send_fd(int fd, int fd_to_send)
{
```

```
    struct CREDSTRUCT   *credp;
    struct cmsghdr      *cmp;
    struct iovec        iov[1];
    struct msghdr       msg;
    char                buf[2]; /* send_fd/recv_ufd 2-byte protocol */

    iov[0].iov_base = buf;
    iov[0].iov_len =  2;
    msg.msg_iov     = iov;
    msg.msg_iovlen =  1;
    msg.msg_name    = NULL;
    msg.msg_namelen = 0;
    msg.msg_flags = 0;
    if (fd_to_send < 0) {
        msg.msg_control    = NULL;
        msg.msg_controllen = 0;
        buf[1] = -fd_to_send;   /* nonzero status means error */
        if (buf[1] == 0)
            buf[1] = 1; /* -256, etc. would screw up protocol */
    } else {
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        msg.msg_control    = cmptr;
        msg.msg_controllen = CONTROLLEN;
        cmp = cmptr;
        cmp->cmsg_level =  SOL_SOCKET;
        cmp->cmsg_type   = SCM_RIGHTS;
        cmp->cmsg_len    = RIGHTSLEN;
        *(int *)CMSG_DATA(cmp) = fd_to_send;   /* the fd to pass */

        cmp = CMSG_NXTHDR(&msg, cmp);
        cmp->cmsg_level =  SOL_SOCKET;
        cmp->cmsg_type   = SCM_CREDTYPE;
        cmp->cmsg_len    = CREDSLEN;
        credp = (struct CREDSTRUCT *)CMSG_DATA(cmp);
#if defined(SCM_CREDENTIALS)
        credp->uid = geteuid();
        credp->gid = getegid();
        credp->pid = getpid();
```

```
#endif
        buf[1] = 0;     /* zero status means OK */
    }
    buf[0] = 0;         /* null byte flag to recv_ufd() */
    if (sendmsg(fd, &msg, 0) != 2)
        return(-1);
    return(0);
}
```

Note that we need to initialize the credentials structure only on Linux.

The function in [Figure] is a modified version of `recv_fd`, called `recv_ufd`, that returns the user ID of the sender through a reference parameter.

25. Receiving credentials over UNIX domain sockets

```
#include "apue.h"
#include <sys/socket.h>     /* struct msghdr */
#include <sys/un.h>

#if defined(SCM_CREDS)          /* BSD interface */
#define CREDSTRUCT      cmsgcred
#define CR_UID          cmcred_uid
#define CREDOPT         LOCAL_PEERCRED
#define SCM_CREDTYPE    SCM_CREDS
#elif defined(SCM_CREDENTIALS)  /* Linux interface */
#define CREDSTRUCT      ucred
#define CR_UID          uid
#define CREDOPT         SO_PASSCRED
#define SCM_CREDTYPE    SCM_CREDENTIALS
#else
#error passing credentials is unsupported!
#endif

/* size of control buffer to send/recv one file descriptor */
#define RIGHTSLEN   CMSG_LEN(sizeof(int))
#define CREDSLEN    CMSG_LEN(sizeof(struct CREDSTRUCT))
```

```
#define CONTROLLEN  (RIGHTSLEN + CREDSLEN)

static struct cmsghdr   *cmptr = NULL;      /* malloc'ed first time */

/*
 * Receive a file descriptor from a server process.  Also, any data
 * received is passed to (*userfunc)(STDERR_FILENO, buf, nbytes).
 * We have a 2-byte protocol for receiving the fd from send_fd().
 */
int
recv_ufd(int fd, uid_t *uidptr,
         ssize_t (*userfunc)(int, const void *, size_t))
{
    struct cmsghdr      *cmp;
    struct CREDSTRUCT   *credp;
    int                 newfd, nr, status;
    char                *ptr;
    char                buf[MAXLINE];
    struct iovec        iov[1];
    struct msghdr       msg;
    const int           on = 1;

    status = -1;
    newfd = -1;
    if (setsockopt(fd, SOL_SOCKET, CREDOPT, &on, sizeof(int)) < 0) {
        err_ret("setsockopt failed");
        return(-1);
    }
    for ( ; ; ) {
        iov[0].iov_base = buf;
        iov[0].iov_len  = sizeof(buf);
        msg.msg_iov     = iov;
        msg.msg_iovlen  = 1;
        msg.msg_name    = NULL;
        msg.msg_namelen = 0;
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        msg.msg_control    = cmptr;
        msg.msg_controllen = CONTROLLEN;
```

```
            if ((nr = recvmsg(fd, &msg, 0)) < 0) {
                err_sys("recvmsg error");
            } else if (nr == 0) {
                err_ret("connection closed by server");
                return(-1);
            }
            /*
             * See if this is the final data with null & status.  Null
             * is next to last byte of buffer; status byte is last byte.
             * Zero status means there is a file descriptor to receive.
             */
            for (ptr = buf; ptr < &buf[nr]; ) {
                if (*ptr++ == 0) {
                    if (ptr != &buf[nr-1])
                        err_dump("message format error");
                    status = *ptr & 0xFF;   /* prevent sign extension */
                    if (status == 0) {
                        if (msg.msg_controllen != CONTROLLEN)
                            err_dump("status = 0 but no fd");

                        /* process the control data */
                        for (cmp = CMSG_FIRSTHDR(&msg);
                          cmp != NULL; cmp = CMSG_NXTHDR(&msg, cmp)) {
                            if (cmp->cmsg_level != SOL_SOCKET)
                                continue;
                            switch (cmp->cmsg_type) {
                            case SCM_RIGHTS:
                                newfd = *(int *)CMSG_DATA(cmp);
                                break;
                            case SCM_CREDTYPE:
                                credp = (struct CREDSTRUCT *)CMSG_DATA(cmp);
                                *uidptr = credp->CR_UID;
                            }
                        }
                    } else {
                        newfd = -status;
                    }
                    nr -= 2;
                }
```

```
            }
            if (nr > 0 && (*userfunc)(STDERR_FILENO, buf, nr) != nr)
                return(-1);
            if (status >= 0)    /* final data has arrived */
                return(newfd);  /* descriptor, or -status */
        }
    }
```

On FreeBSD, we specify `SCM_CREDS` to transmit credentials; on Linux, we use `SCM_CREDENTIALS`.

Comment