Sharing a File Descriptor between Two Processes

Authored by: **Thomas Fletcher**
Updated by: **Thomas Fletcher**

One of the rarely used, but quite handy, features often found in traditional UNIX systems is the ability to pass file descriptors from one process to another through a UNIX domain socket (uds). If you aren't familiar with them, UNIX domain sockets are a combination of pipe and socket services. You have all of the benefits of a bi-directional local pipe (i.e. speed), and you can use the exact same socket function calls to communicate locally as you would to communicate across a regular socket.

UNIX domain sockets were introduced into the Neutrino 6.2 release, but prior to that time you were stuck if you needed the facility of passing a file descriptor between two processes. Using the UNIX domain sockets as a model, this article describes how that functionality can be provided using the standard Neutrino IPC mechanisms. It provides a bit of insight into some of the darker corners of the resource manager and messaging framework.

Understanding How Things Work

The first thing to remember is that the process of obtaining a file descriptor (otherwise known as a connection id in QNX Neutrino) involves the client process connecting to a channel that has been established by some server process. The client then sends a message (generally **_IO_CONNECT_OPEN**) down the channel which the server receives and processes. Part of the processing includes checking access control lists, validating paths, etc. In the case of a successful **open**() function call, the server will bind the calling client's connection to some internal resource manager data structures, and all subsequent standard messages (**_IO_READ, _IO_WRITE** etc.), will resolve through this binding.

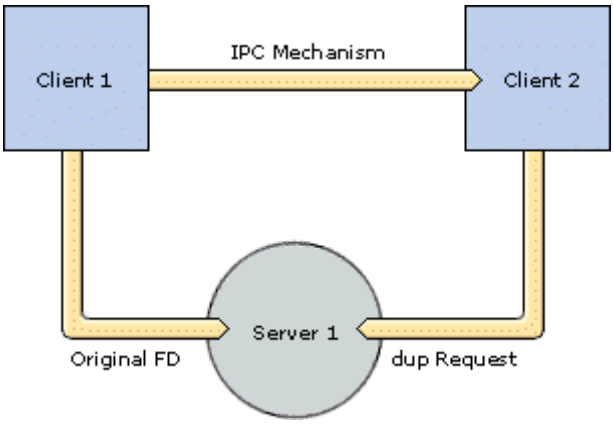
In a monolithic system, where there is usually one central virtual file system (vfs) used by many different file systems, the mapping of file descriptors in one process (such as what happens on a **fork()** system call) to the file descriptors in another process, is quite straightforward.

Due to the distributed nature of the client file descriptor binding operation under the QNX Neutrino microkernel (one process may have many file descriptors that are, in turn, mapped to many different servers, each process with its own unique set of bindings), it is impossible to simply pass numeric integer values between processes. Each client process must build up their file descriptor mappings by talking to each server process individually.

Talking to Strangers

What we will provide in this example is a way that two client applications can use some communication channel (pipe, socket, shared memory, native IPC, carrier pigeon) to pass a packet of information that contains enough information such that the receiver can contact a given server to duplicate a file descriptor that exists in the client's address space.

Graphically, it looks like:



The scenario is as follows:

Client1 has a file descriptor to Server1 (i.e., a file system) as well as a communication channel to Client2 (i.e., a pipe). Client1 would like Client2 to have the ability to use the same file descriptor as it's using.

In this scenario, the packet of information will be a dup message that's described in <*sys/iomsg.h*>. This message is put together by Client1 and contains all of the information that's required for Client2 to establish a connection to the server that mimics the first client's connection.

Here is the code sample that's used for sending the message:

```
/*
 Generate the data packet (message) that will be sent using
 func() to duplicate the file descriptor fdtosend.
*/
int send_fd(int fdtosend, int flags, void *arg,
            int (*func)(void *arg, void *data, int len)) {
 int ret;
 io_dup_t dupmsg;
 memset(&dupmsg, 0, sizeof(dupmsg));

 dupmsg.i.type = _IO_DUP;
 dupmsg.i.combine_len = sizeof(dupmsg.i);
 if((ret = ConnectServerInfo(0, fdtosend, &dupmsg.i.info)) == -1) {
  return ret;
 }
 /* Overload the thread_id to carry this pid's address */
 dupmsg.i.info.tid = getpid();
 printf("SFD: client pid: %dn"
        "     server nd: 0x%x pid 0x%x chid 0x%x scoid 0x%x coid 0x%xn",
  dupmsg.i.info.tid,
  dupmsg.i.info.nd,
  dupmsg.i.info.pid,
  dupmsg.i.info.chid,
  dupmsg.i.info.scoid,
  dupmsg.i.info.coid);
```

```
 /* Actually perform the transmission of the data through the user
callback */
 if(func) {
  ret = func(arg, &dupmsg, sizeof(dupmsg));
 } else {
  ret = 0;
 }
 return ret;
}
```

What's going on here is that we're filling in the **io_dup_t** message structure and sending it as content along the user's communication channel. Given the client file descriptor, we gather all of the information about the channel (and server) using the **ConnectServerInfo**() function. This conveniently returns most of what we need for an **io_dup_t** message. We don't have any extra messages that are being sent, so we can safely fill in the type and the **combine_len** to indicate a single **_IO_DUP** message.

When the receiver gets this message, it's going to need to know which process sent it to be able to properly generate the **_IO_DUP** request. There are a few unused fields in the **msg_info** structure that are meaningless for server information (see **<sys/neutrino.h>**), so we hijack the thread id field (**tid**) and stuff Client1's process id inside it.

Now, with the message readied, it's passed off for transmission by the communication mechanism of choice. This brings us to the code that's used on the receiver side to decode the data and get a new file descriptor:

```
/*
 Given a received packet of data of length len, attempt to extract
 out a DUP message that we can use to generate a file descriptor.
*/
int receive_fd(void *data, int len) {
 io_dup_t *pdupmsg = (io_dup_t *)data;
 pid_t   otherpid;
 int    ret, newfd;

 /* Very basic validity tests on the message */
 if(len < sizeof(*pdupmsg) || pdupmsg->i.type != _IO_DUP) {
  errno = EINVAL;
  return -1;
 }

 /* Extract out our overloaded field */
 otherpid = pdupmsg->i.info.tid;
 pdupmsg->i.info.tid = 0;

 printf("RFD: other pid: %dn"
        "     server nd: 0x%x pid 0x%x chid 0x%x scoid 0x%x coid 0x%xn",
   otherpid,
   pdupmsg->i.info.nd,
   pdupmsg->i.info.pid,
   pdupmsg->i.info.chid,
   pdupmsg->i.info.scoid,
   pdupmsg->i.info.coid);

 /* Establish a connection with the channel first */
 if ((newfd = ConnectAttach(pdupmsg->i.info.nd,
                            pdupmsg->i.info.pid,
                            pdupmsg->i.info.chid, 0, 0)) < 0) {
```

```
 return -1;
 }

 /* Make out like we are the other process */
 pdupmsg->i.info.pid = otherpid;

 /*
 Send the message to the server asking it to duplicate the
 connection that has been described by the client. If
 this request succeeds, then the server resource manager layer
 has bound in our request. If it fails, disconnect from the
 channel.
 */
 if(MsgSendnc(newfd, &pdupmsg->i, sizeof(pdupmsg->i), 0, 0) == -1) {
  ConnectDetach_r(newfd);
  return -1;
 }

 return newfd;
}
```

After some very basic sanity tests to make sure we've received an **_IO_DUP** message, the receiver code attempts to establish a connection to the same channel that the originator of the message had, using the **ConnectAttach()** function. If that succeeds, then the actual **_IO_DUP** message is sent, which will bind the newly obtained connection id to a particular file/device in the server's resource manager layer. Once it's bound in the resource manager, the connection id becomes a proper file descriptor that can then be used with the normal **read()/write()** function calls. If successful, the newly created file descriptor is then returned to the caller.

Conditions and Terms of Agreement

Like most things, there are a couple of conditions that are associated with this code, and assumptions that have been made that, if violated, will most likely cause the process not to work as expected.

Caveats:

> Only works between clients in similar uid groups (or root) Only works using normal file descriptors, not **_NTO_SIDE_CHANNEL** ones The server must support **_IO_DUP** messages (most do by default)

We'll end with a final exercise. Modify the code not to duplicate the file descriptors but instead to use the **_IO_OPENFD** to generate a new file descriptor to a file that has been opened by another client.

```
Full Example
------------
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <mqueue.h>
#include <sys/iomsg.h>
#include <sys/resmgr.h>

#define MY_MQ_NAME "/fdpass"
```

```
/*
 Generate the data packet (message) that will be sent using
 func() to duplicate the file descriptor fdtosend.
*/
int send_fd(int fdtosend, int flags, void *arg,
            int (*func)(void *arg, void *data, int len)) {
 int ret;
 io_dup_t dupmsg;

 memset(&dupmsg, 0, sizeof(dupmsg));

 dupmsg.i.type = _IO_DUP;
 dupmsg.i.combine_len = sizeof(dupmsg.i);

 if((ret = ConnectServerInfo(0, fdtosend, &dupmsg.i.info)) == -1) {
  return ret;
 }
 /* Overload the thread_id to carry this pid's address */
 dupmsg.i.info.tid = getpid();

 printf("SFD: client pid: %dn"
        "     server nd: 0x%x pid 0x%x chid 0x%x scoid 0x%x coid 0x%xn",
  dupmsg.i.info.tid,
  dupmsg.i.info.nd,
  dupmsg.i.info.pid,
  dupmsg.i.info.chid,
  dupmsg.i.info.scoid,
  dupmsg.i.info.coid);

 /* Actually perform the transmission of the
    data through the user callback */

 if(func) {
  ret = func(arg, &dupmsg, sizeof(dupmsg));
 } else {
  ret = 0;
 }

 return ret;
}

/*
 Given a received packet of data of length len, attempt to extract
 out a DUP message which we can then generate a file descriptor with.
*/
int receive_fd(void *data, int len) {
 io_dup_t *pdupmsg = (io_dup_t *)data;
 pid_t   otherpid;
 int    ret, newfd;

 /* Very basic validity tests on the message */
 if(len < sizeof(*pdupmsg) || pdupmsg->i.type != _IO_DUP) {
  errno = EINVAL;
  return -1;
 }

 /* Extract out our overloaded field */
 otherpid = pdupmsg->i.info.tid;
 pdupmsg->i.info.tid = 0;
```

```
 printf("RFD: other pid: %dn"
        "     server nd: 0x%x pid 0x%x chid 0x%x scoid 0x%x coid 0x%xn",
   otherpid,
   pdupmsg->i.info.nd,
   pdupmsg->i.info.pid,
   pdupmsg->i.info.chid,
   pdupmsg->i.info.scoid,
   pdupmsg->i.info.coid);

 /* Establish a connection with the channel first */
 if ((newfd = ConnectAttach(pdupmsg->i.info.nd,
                            pdupmsg->i.info.pid,
                            pdupmsg->i.info.chid, 0, 0)) < 0) {
  return -1;
 }

 /* Make out like we are the other process */
 pdupmsg->i.info.pid = otherpid;

 /*
 Send the message to the server asking it to duplicate the
 connection that has been described by the client.  If
 this request succeeds, then the server resource manager layer
 has bound in our request. If it fails, disconnect from the
 channel.
 */
 if(MsgSendnc(newfd, &pdupmsg->i, sizeof(pdupmsg->i), 0, 0) == -1) {
  ConnectDetach_r(newfd);
  return -1;
 }

 return newfd;
}

int mq_write(void *arg, void *data, int len) {
 return mq_send((int)arg, data, len, 0);
}

int main(int argc, char **argv) {
 mqd_t  mqfd;
 int  fd, ret, sender;
 char    c;

 sender = -1;
 while((ret = getopt(argc, argv, "sr")) != -1) {
  switch(ret) {
  case 's':
   sender = 1;
   break;
  case 'r':
   sender = 0;
   break;
  }
 }

 if(sender < 0) {
  printf("Usage %s -s | -r n", argv[0]);
  printf("Where:n"
         " -s Indicates program should send the fdn"
         " -r Indicates program should receive the fdn");
```

```
  return 1;
 }

 if((mqfd = mq_open(MY_MQ_NAME, O_CREAT | O_RDWR, 0666, NULL))
== -1) {
  perror("Can't open/create mqueue");
  return 1;
 }


 if(sender) {
  //If we are the sender then we are sending an fd
  if((fd = open("/etc/passwd", O_RDONLY)) == -1) {
   perror("Can't open file to send");
   return 1;
  }

  //For kicks, we read the first line and re-position the fd
  printf("SFD: first line [");
  while(read(fd, &c, 1) > 0) {
   if(c == 'n') {
    break;
   }
   printf("%c", c);
  }
  printf("]\n");

  //Send the re-positioned fd to the other side
  ret = send_fd(fd, 0, (void *)mqfd, mq_write);

  //Sleep for a bit to ensure the receiver is run
  sleep(10);
 } else {
  char     *msg;
  struct mq_attr mqstat;

  if(mq_getattr(mqfd, &mqstat) == -1) {
   perror("Can't get mqueue attribute");
   return 1;
  }

  msg = alloca(mqstat.mq_msgsize);

  //If we are the receiver, then we are receiving an fd
  if(!msg || mq_receive(mqfd, msg, mqstat.mq_msgsize, NULL) == -1) {
   perror("Can't receive message");
   return 1;
  }

  //Pass the received buffer off to the interpreter
  if((fd = receive_fd(msg, mqstat.mq_msgsize)) == -1) {
   perror("Can't create file descriptor");
   return 1;
  }

  //Read the next line to make sure we actually have the fd
  printf("RFD: next line [");
  while(read(fd, &c, 1) > 0) {
   if(c == 'n') {
    break;
```

```
    }
    printf("%c", c);
   }
   printf("]\n");
 }

 //Cleanup ...
 printf("%s: cleaning up and exiting n", (sender) ? "SFD" : "RFD");
 close(fd);
 mq_close(mqfd);
 return 0;
}
```

Want more information?

For more information on UNIX domain sockets refer to UNIX Network Programming, Volume I, by W. Richard Stevens.