SoByte

- Home
- Archives
- Tags
- Categories
- About

Passing file descriptors between processes via UNIX domain sockets

2022-01-04
tutorials
1209 words 6 min read

Table of Contents

Linux provides a series of system calls that allow us to pass file descriptors between processes. Instead of simply passing the file descriptor, which is a 32-bit integer, we actually pass the file handle to the target process so that it can read and write to the file. Now suppose process B wants to send a file descriptor to process A. Let's see how to do that.

1. UNIX domain socket

To pass file descriptors, you need to establish inter-process communication. UNIX domain socket is a way to communicate between processes, it is similar to a normal socket, except that it does not use an ip address, but a socket file. We want to use it to send control messages and thus pass file descriptors. First we will have process A create a socket:

```
/*code of process A*/
int socket_fd = socket(AF_UNIX, SOCK_DGRAM, 0);
```

Note the arguments to the `socket` function. `AF_UNIX` specifies the protocol family as a UNIX domain socket. Here we use the datagram socket `SOCK_DGRAM` , which is similar to UDP and does not require a link to be established, but is sent directly through the address. Of course, you can also use `SOCK_STREAM`, which is similar to TCP. We won't list them here.

Next we bind the address:

```
/*code of process A*/
struct sockaddr_un un;
un.sun_family = AF_UNIX;
unlink("process_a");
strcpy(un.sun_path, "process_a");

if (bind(socket_fd, (struct sockaddr*)&un, sizeof(un)) < 0) {
    printf("bind failed\n");
    return 1;
}
```

Note that the address here is no longer an ip address, but a file path. Here we specify the address as `process_a`, and call `bind` to bind the address, which creates a socket file called `process_a`.

This way, other processes can send messages to this process through a special address like `process_a`. It is the same as sending UDP messages, except that `struct sockaddr_un` is used to define the address instead of `struct sockaddr_in` or `struct sockaddr_in6`. In addition, it is important to note that other processes can pass file descriptors to this process by sending control messages.

2. sendmsg

Linux provides a pair of system calls: `sendmsg` and `recvmsg` . Unlike our usual `send` and `recv`, they can be used to send or receive control messages in addition to regular data, which is the key to passing file descriptors; they can also be used to send or receive a discrete piece of data.

Let's look at the declarations of these two system calls

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

ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);
ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);
```

`sendmsg` and `recvmsg` use a structure `struct msghdr` to describe the data sent. The structure is defined as follows:

```
struct iovec {                    /* Scatter/gather array items */
    void  *iov_base;              /* Starting address */
```

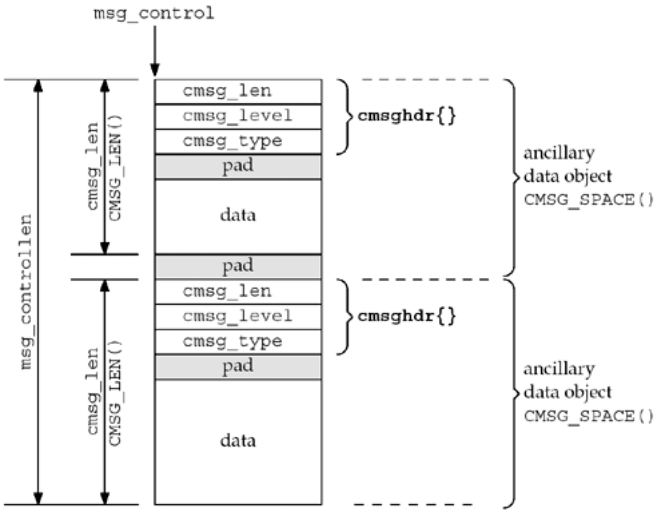
```
    size_t iov_len;              /* Number of bytes to transfer */
};

struct msghdr {
    void         *msg_name;       /* optional address */
    socklen_t     msg_namelen;    /* size of address */
    struct iovec *msg_iov;        /* scatter/gather array */
    size_t        msg_iovlen;     /* # elements in msg_iov */
    void         *msg_control;    /* ancillary data, see below */
    size_t        msg_controllen; /* ancillary data buffer len */
    int           msg_flags;      /* flags on received message */
};
```

- `msg_name` Destination address. This is optional, if the protocol is connection-oriented, you don't need to specify an address; otherwise, you need to specify an address. This is similar to `send()` and `sendto()` .
- `msg_iov` the data to be sent. This is an array, the length of which is specified by `msg_iovlen`, and the elements of the array are a `struct iovec` structure that specifies the starting address (`iov_base`) and length (`iov_len`) of a contiguous piece of data. In other words, it can send more than one continuous piece of data; or, it can send a discontinuous piece of data.
- `msg_control` controls the message. This is what we are talking about today. We can't set it directly, we have to use a series of macros to set it.

`msg_control` points to a sequence consisting of a `struct cmsghdr` structure and its additional data. The definition of `struct cmsghdr` is as follows:

```
struct cmsghdr {
    socklen_t cmsg_len;    /* data byte count, including header */
    int       cmsg_level;  /* originating protocol */
    int       cmsg_type;   /* protocol-specific type */
/* followed by
    unsigned char cmsg_data[]; */
};
```

`struct cmsghdr` actually defines the header of the data, which should be followed by an `unsigned char` array that holds the actual data of the control information. This is often referred to as a `variable-length structure`. `msg_control` points to a sequence of such variable-length structures. The memory structure is shown in the figure below:



We need to use the following macros:

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

struct cmsghdr *CMSG_FIRSTHDR(struct msghdr *msgh);
size_t CMSG_SPACE(size_t length);
size_t CMSG_LEN(size_t length);
unsigned char *CMSG_DATA(struct cmsghdr *cmsg);
```

- `CMSG_FIRSTHDR()` returns the first element of the sequence pointed to by `msg_control`
- `CMSG_SPACE()` The length of the actual data of the incoming control message, returning the space required for the variable-length structure
- `CMSG_LEN()` The length of the actual data of the control message, and the length of the variable-length structure.
- `CMSG_DATA()` Returns the first address of the actual data that holds the control information.

  Note the difference between `CMSG_SPACE()` and `CMSG_LEN()`: the former contains the length of the padding, which is the actual space occupied; the latter does not contain the length of the padding, and is used to assign the value to `cmsg_len` .

Next, let's have process B pass a file descriptor to process A. First set the address of process A, which is "process_a":

```
/*code of process B*/
struct sockaddr_un ad;
ad.sun_family = AF_UNIX;
strcpy(ad.sun_path, "process_a");
```

We only need to send control messages, not regular data, so we leave the regular data empty:

```
/*code of process B*/
struct iovec e = {NULL, 0};
```

Next, we allocate space for the control data, since we only pass one file descriptor, so the length is `sizeof(int)` :

```c
/*code of process B*/
char cmsg[CMSG_SPACE(sizeof(int))];
```

Then you can set the `struct msghdr` structure:

```c
/*code of process B*/
struct msghdr m = {(void*)&ad, sizeof(ad), &e, 1, cmsg, sizeof(cmsg), 0};
```

Next we get `struct cmsghdr` and set it:

```c
/*code of process B*/
struct cmsghdr *c = CMSG_FIRSTHDR(&m);
c->cmsg_level = SOL_SOCKET;
c->cmsg_type = SCM_RIGHTS;
c->cmsg_len = CMSG_LEN(sizeof(int));
*(int*)CMSG_DATA(c) = cfd; // set file descriptor
```

Finally, just send it out:

```c
/*code of process B*/
sendmsg(mfd, &m, 0)
```

3. recvmsg

Now we ask process A to receive the file descriptor passed to it. Here we call `recvmsg` .

```c
/*code of process A*/
char buf[512];
struct iovec e = {buf, 512};
char cmsg[CMSG_SPACE(sizeof(int))];
struct msghdr m = {NULL, 0, &e, 1, cmsg, sizeof(cmsg), 0};

int n = recvmsg(socket_fd, &m, 0);
printf("Receive: %d\n", n);

struct cmsghdr *c = CMSG_FIRSTHDR(&m);
int cfd = *(int*)CMSG_DATA(c); // receive file descriptor
```

Thus, process A receives the file descriptor from process B, and process B opens the file and can perform read and write operations on it.
unix socket

Enable Tab Search on your own blog Prev Promise and Deferred Next