Advanced IPC

Advanced Programming in the UNIX Environment

Chun-Ying Huang <chuang@cs.nctu.edu.tw>

Outline

Introduction

UNIX domain socket

Passing file descriptors

Passing file descriptors and the credentials

Examples

Introduction

We have discussed various forms of IPC, including pipes and sockets

This chapter focuses on UNIX domain sockets

We can pass open file descriptors between processes

Servers can associate names with their file descriptors

Clients can use these names to rendezvous with the servers

NOTE: The textbook introduces STREAMS IPC, but it is not available on recent Linux and BSD operating systems, so we omit the parts relevant to STREAMS IPC

UNIX Domain Socket

UNIX domain sockets are used to communicate with processes running on the same machine

UNIX domain sockets are more efficient than Internet domain sockets

UNIX domain sockets only copy data, no protocols are involved

- No headers, checksums, sequence numbers, and acknowledgements …

UNIX domain sockets provide both stream and datagram interfaces

UNIX domain sockets are reliable

- Messages are neither lost nor delivered out-of-order

Unnamed UNIX Domain Sockets

You can use socketpair function to create unnamed UNIX domain sockets

- ```
  int socketpair(int domain, int type, int protocol,
                      int sockfd[2]);
  ```

- Returns: zero if success, or -1 on error
- It works like a two-way (full-duplex) pipe
- Some BSD-systems implement pipe() using socketpair() function
- The write end of the first descriptor and the read end of the second descriptor are both closed
  ```

The s_pipe Function

Create a full-duplex pipe using UNIX domain socket

```
int s_pipe(int fd[2]) {
    return(socketpair(AF_UNIX, SOCK_STREAM, 0, fd));
}
```

An Example with Unnamed UNIX Domain Socket: Hello, World

The parent sends “hello, ” for the client to print out

The client sends “world!\n” for the parent to print out

See advipc/hellounix.c

```
socketpair(AF_UNIX, SOCK_STREAM, 0, fd);
pid = fork();
if(pid > 0) {           /* parent */
        write(fd[0], MESSAGE1, strlen(MESSAGE1));
        if((len = read(fd[0], buf, sizeof(buf))) < 0)
                err_sys("parent/read");
        write(1, buf, len);
        wait(&status);
} else if(pid == 0) {   /* child */
        if((len = read(fd[1], buf, sizeof(buf))) < 0)
                err_sys("child/read");
        write(1, buf, len);
        write(fd[1], MESSAGE2, strlen(MESSAGE2));
}
...
```

Naming UNIX Domain Sockets

Although the socketpair function creates sockets that are connected to each other, the individual sockets don't have names

This means that they can not be addressed by unrelated processes

The sockaddr_un structure (on Linux and Solaris)

```
struct sockaddr_un {
        sa_family_t sun_family;   /* AF_UNIX */
        char sun_path[108];       /* pathname */
};
```

The sockaddr_un structure (on BSD and Mac OS X)

```
struct sockaddr_un {
        unsigned char sun_len;    /* length including null */
        sa_family_t sun_family;   /* AF_UNIX */
        char sun_path[108];       /* pathname */
};
```

Bind a UNIX Domain Socket

```
int main(void) {
    int fd, size;
    struct sockaddr_un un;

    un.sun_family = AF_UNIX;
    strcpy(un.sun_path, "foo.socket");
    if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        err_sys("socket failed");
    size = offsetof(struct sockaddr_un, sun_path)
         + strlen(un.sun_path);
    if (bind(fd, (struct sockaddr *)&un, size) < 0)
        err_sys("bind failed");
    printf("UNIX domain socket bound\n");
    exit(0);
}
```

Bind a UNIX Domain Socket

```
$ ./fig17.14-bindunix
UNIX domain socket bound
$ ls -l foo.socket
srwxrwxr-x 1 huangant huangant 0 May 10 23:57 foo.socket
$ ./fig17.14-bindunix
bind failed: Address already in use
$ rm foo.socket
$ ./fig17.14-bindunix
UNIX domain socket bound
```

Unique Connections

A server can arrange for unique UNIX domain connections to clients using the standard bind, listen, and accept functions

Clients use connect to contact the server

After the connect request is accepted by the server, a unique connection exists between the client and the server

This style of operation is the same that we illustrated with Internet domain sockets

Examples (see sample codes in advipc)

- Connection-oriented UNIX domain socket – server: advipc/unixsrv1.c
- Connection-oriented UNIX domain socket – client: advipc/unixcli1.c
- Connectionless UNIX domain socket – server: advipc/unixsrv2.c
- Connectionless UNIX domain socket – client: advipc/unixcli2.c

Passing File Descriptors

The ability to pass an open file descriptor between processes is powerful

It allows one process (typically a server) to do everything that is required to open a file, and simply pass back to the calling process a descriptor that can be used with all the I/O functions

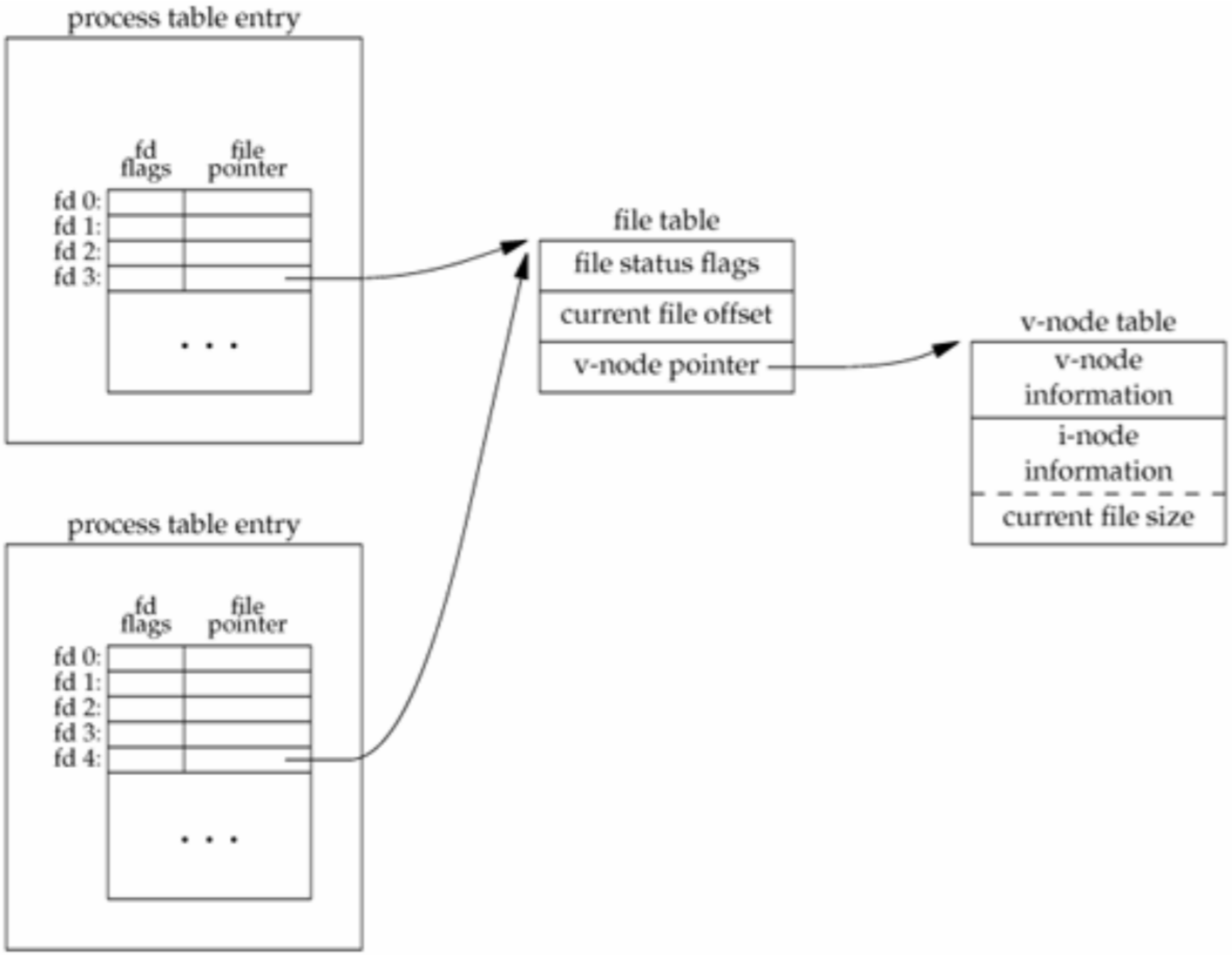
The involved two processes can be irrelevant processes

- They don't have to be in parent and child relationship

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

Passing File Descriptors (Cont'd)



Passing File Descriptors: Simple APIs

The textbook has defined simple APIs to pass descriptors

- `int send_fd(int fd, int fd_to_send);`
- `int send_err(int fd, int status, const char *errmsg);`
- Returns: zero if success, or -1 on error

- ```
  int recv_fd(int fd,
              ssize_t (*userfunc)(int, const void *, size_t));
  ```
- Returns: file descriptor if success, or negative value on error

send_fd sends a descriptor via fd, which is a UNIX domain socket

send_err sends an error message via fd

recv_fd receives a descriptor or an error message via fd

See the implementations in subsequent slides
  ```

Passing File Descriptors: Using UNIX Domain Sockets

The key is to send ancillary data using sendmsg/recvmsg

Procedures

- Create a UNIX domain socket connection between two processes
- The sender creates an ancillary data to store the descriptor to be sent
- The sender sends the ancillary data along with a message
- NOTE: the message MUST have at least one-byte data
- The receiver receives the message and the ancillary data
- The receiver retrieves the descriptor in the ancillary data

sendmsg and recvmsg

Synopsis

- `ssize_t sendmsg(int sockfd, const struct msghdr *msg,`
  `int flags);`
- `ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);`
- Returns: Number of bytes sent/received, or -1 on error

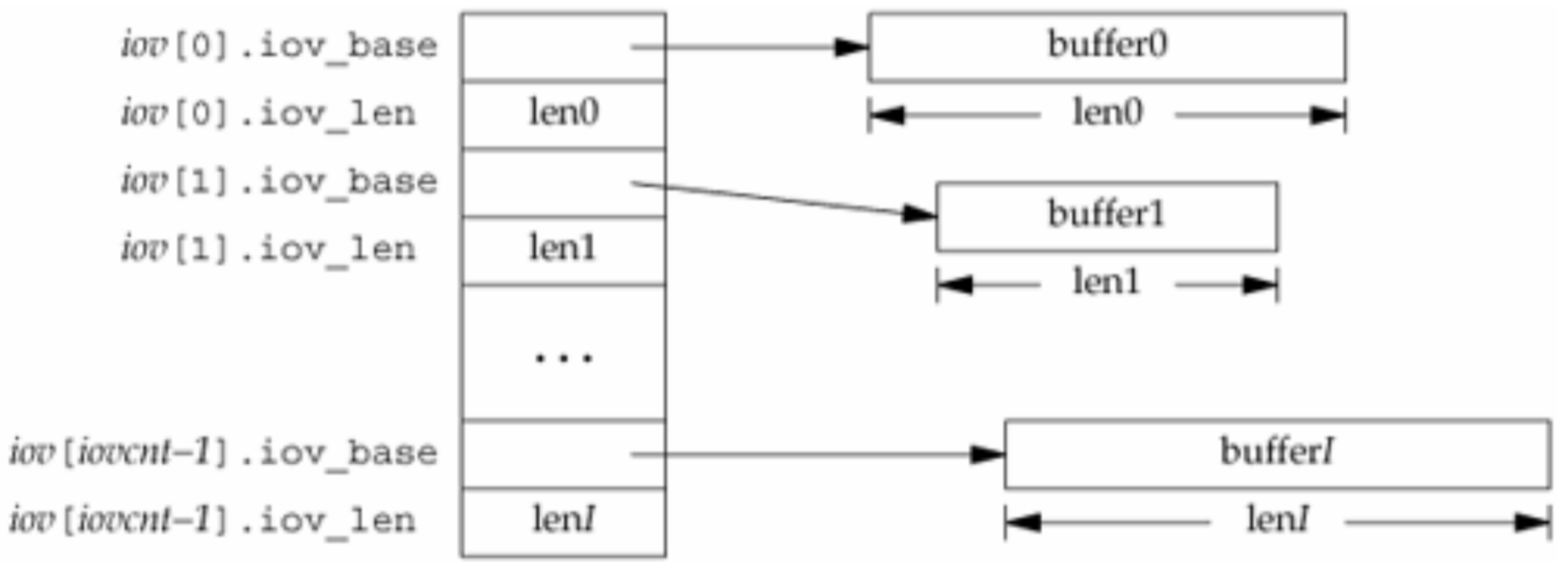
The msghdr structure

```
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 */
};
```

sendmsg and recvmsg (Cont'd)

The iov structure is the same to that used in readv/writev functions

Note that we MUST send at least one-byte data – even if we ONLY want to send the ancillary data



The Ancillary Data

Ancillary data is used to send access control messages (not part of the payloads)

Ancillary data can be used to send messages like header fields, extended error descriptions, file descriptors, or UNIX credentials

Ancillary data is stored in the “msg_control” and “msg_controllen” fields in the msghdr structure

The ancillary data structure

```
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[]; */
};
```

Setup Ancillary Data Structure

We have to properly fill the cmsghdr data structure for passing a file descriptor

- CMSG_SPACE(): The required spaces (including alignment)
- CMSG_LEN(): The actual data length

```
// allocate sufficient spaces: for storing a single descriptor
char cmsgbuf[CMSG_SPACE(sizeof(int))];
struct cmsghdr *pcm = (struct cmsghdr*) cmsgbuf;

// setup the level and type
memset(pcm, 0, sizeof(cmsgbuf));      /* this is important!! */
pcm->cmsg_level = SOL_SOCKET;
pcm->cmsg_type = SCM_RIGHTS;
pcm->cmsg_len = CMSG_LEN(sizeof(int));

// put the descriptor into cmsg data buffer (as an int array)
* (int*) CMSG_DATA(pcm) = desc;
```

Send the Descriptor

Once we have the ancillary data prepared, send it along with a message

```
char data[1] = { 0xab };  /* arbitrary magic number:
              for the receiver to confirm the message */
struct iovec io[1];
struct msghdr m;

memset(&m, 0, sizeof(m));
io[0].iov_base = data;
io[0].iov_len = 1;
m.msg_iov = io;
m.msg_iovlen = 1;
m.msg_control = pcm;
m.msg_controllen = CMSG_SPACE(sizeof(int));
if(sendmsg(c, &m, 0) < 0)
    err_sys("sendmsg");
```

Receive the Descriptor

The receiver prepares and setup the data structure for receiving both the message and the ancillary data

The receiver then calls recvmsg to receive the message and the data

```
char data[1] = { 0 };
char cmsgbuf[CMSG_SPACE(sizeof(int))];
struct iovec io[1];
struct msghdr m;
struct cmsghdr *pcm = (struct cmsghdr*) cmsgbuf;

memset(&m, 0, sizeof(m));
memset(cmsgbuf, 0, sizeof(cmsgbuf));
io[0].iov_base = data;
io[0].iov_len = 1;
m.msg_iov = io;
m.msg_iovlen = 1;
m.msg_control = pcm;
m.msg_controllen = sizeof(cmsgbuf);
```

Receive the Descriptor

The receiver then calls recvmsg to receive the message and the data

See complete examples in advipc/{cmsgsrv.c and cmsgcli.c}

```
if(recvmsg(s, &m, 0) < 0)
    err_sys("recvmsg");

/* optionally check the received message */
if(data[0] != (char) MAGIC) {
    /* something went wrong !? */
}

fd = * (int*) CMSG_DATA(pcm);
```

Who Passes the Descriptors?

We may send the credentials of the sender along with the descriptors

The credentials can also be sent using ancillary data channel

However, the structure to store credentials varies on different OS

FreeBSD

```
#define CMGROUP_MAX 16

struct cmsgcred {
   pid_t cmcred_pid;  /* sender's PID */
   uid_t cmcred_uid;  /* sender's real UID */
   uid_t cmcred_euid; /* sender's EUID */
   gid_t cmcred_gid;  /* sender's real GID */
   short cmcred_ngroups;   /* number of groups */
   gid_t cmcred_groups[CMGROUP_MAX]; /* groups */
};
```

Linux

```
struct ucred {
    uint32_t pid; /* sender’s PID */
    uint32_t uid; /* sender’s user ID */
    uint32_t gid; /* sender’s group ID */
};
```

Pass the Descriptors and the Credentials Together: The Sender

Procedures to pass BOTH the descriptors and credentials

1. Create a large enough spaces
2. IMPORTANT: memory spaces must be filled with zeros
3. Use CMSG_FIRSTHDR() to obtain the first cmsg block, and store the descriptors to be passed
4. Use CMSG_NXGHDR() to obtain the second cmsg block, and store the credentials to be passed (process id, user id, and group id)
5. On Linux, fill the content of ucred structure
6. Send out the message

See code snippets in later slides, and complete slides in advipc/{cmsgsrv2.c and advipc/cmsgcli2.c}

Handle Platform Differences

```
#if defined(__FreeBSD__)
typedef struct cmsgcred cred_t;
#define SCM_CREDTYPE    SCM_CREDS
#elif defined(__linux__)
typedef struct ucred    cred_t;
#define SCM_CREDTYPE    SCM_CREDENTIALS
#else
#error passing credentials is unsupported!
#endif
```

Sender Procedures

Allocate spaces, fill zeros, and fill the data message structure

```
char data[1] = { MAGIC };
char cmsgbuf[CMSG_SPACE(sizeof(int)) + CMSG_SPACE(sizeof(cred_t))];
struct iovec io[1];
struct msghdr m;
struct cmsghdr *pcm = (struct cmsghdr*) cmsgbuf;

memset(&m, 0, sizeof(m));
memset(cmsgbuf, 0, sizeof(cmsgbuf));    /* this is important!! */

io[0].iov_base = data;
io[0].iov_len = 1;
m.msg_iov = io;
m.msg_iovlen = 1;
m.msg_control = pcm;
m.msg_controllen = sizeof(cmsgbuf);
```

Sender Procedures (Cont'd)

Obtain the first block, and fill the descriptor

```
pcm = CMSG_FIRSTHDR(&m);
pcm->cmsg_level = SOL_SOCKET;
pcm->cmsg_type = SCM_RIGHTS;
pcm->cmsg_len = CMSG_LEN(sizeof(int));
* (int*) CMSG_DATA(pcm) = 1; /* descriptor to send */
```

Obtain the second block, fill the credentials, and finally send it out

```
  pcm = CMSG_NXTHDR(&m, pcm);
  pcm->cmsg_level = SOL_SOCKET;
  pcm->cmsg_type = SCM_CREDTYPE;
  pcm->cmsg_len = CMSG_LEN(sizeof(cred_t));
#ifdef __linux__  /* required on Linux */
  pcred = (cred_t*) CMSG_DATA(pcm);
  pcred->uid = getuid();
  pcred->gid = getgid();
  pcred->pid = getpid();
#endif
```

Pass the Descriptors and the Credentials Together: The Receiver

Procedures to receive BOTH the descriptors and credentials

1. Create a large enough spaces
2. It would be better to fill memory spaces with zeros
3. On Linux: enable SO_PASSCRED option
4. Receive the message
5. Write a loop to iteratively check all cmsg blocks
6. Retrieve the descriptor and the credentials

See code snippets in later slides, and complete slides in advipc/{cmsgsrv2.c and advipc/cmsgcli2.c}

Receiver Procedures (1/3)

Allocate spaces, fill zeros, and fill the data message structure

```
char data[1] = { 0 };
char cmsgbuf[CMSG_SPACE(sizeof(int))+CMSG_SPACE(sizeof(cred_t))];
struct iovec io[1];
struct msghdr m;
struct cmsghdr *pcm = (struct cmsghdr*) cmsgbuf;

memset(&m, 0, sizeof(m));
memset(cmsgbuf, 0, sizeof(cmsgbuf));

io[0].iov_base = data;
io[0].iov_len = 1;
m.msg_iov = io;
m.msg_iovlen = 1;
m.msg_control = pcm;
m.msg_controllen = sizeof(cmsgbuf);
```

Receiver Procedures (2/3)

Enable SO_PASSCRED on Linux, and then receive the message

```
#ifdef __linux__
  do {
     int v = 1;
     setsockopt(s, SOL_SOCKET, SO_PASSCRED, &v, sizeof(v));
  } while(0);
#endif
```

Receiver Procedures (3/3)

Iteratively retrieve the ancillary data blocks

```
for (pcm = CMSG_FIRSTHDR(&m); pcm != NULL; pcm = CMSG_NXTHDR(&m, pcm)) {
    if (pcm->cmsg_type == SCM_RIGHTS) {
       newfd = * (int*) CMSG_DATA(pcm);
       fprintf(stderr, "client: descriptor %d received.\n", newfd);
    } else if(pcm->cmsg_type == SCM_CREDTYPE) {
       cred_t *pcred = (cred_t*) CMSG_DATA(pcm);
       fprintf(stderr, "client: received from pid %u, uid %u, gid %u\n",
#ifdef __FreeBSD__
            pcred->cmcred_pid, pcred->cmcred_uid, pcred->cmcred_gid
#elif defined(__linux__)
            pcred->pid, pcred->uid, pcred->gid
#endif
       );
    }
}
```

Running the Example

```
$ ./cmsgsrv2 &
[1] 12811
$ ./cmsgcli2
descriptor sent, pid 12813, uid 1000, gid 1000          (from server)
client: received from pid 12813, uid 1000, gid 1000     (from client)
client: descriptor 4 received.                          (from client)
hello, world!                    (user input: read from client's stdin)
hello, world!                    (output from stdout of the server)
(^D)
$ ./cmsgcli2 > /dev/null 2>/devnull      (disable outputs from the client)
descriptor sent, pid 12823, uid 1000, gid 1000          (from server)
hello, world!                                           (user input)
hello, world!                    (output from stdout of the server)
$
```

Q & A